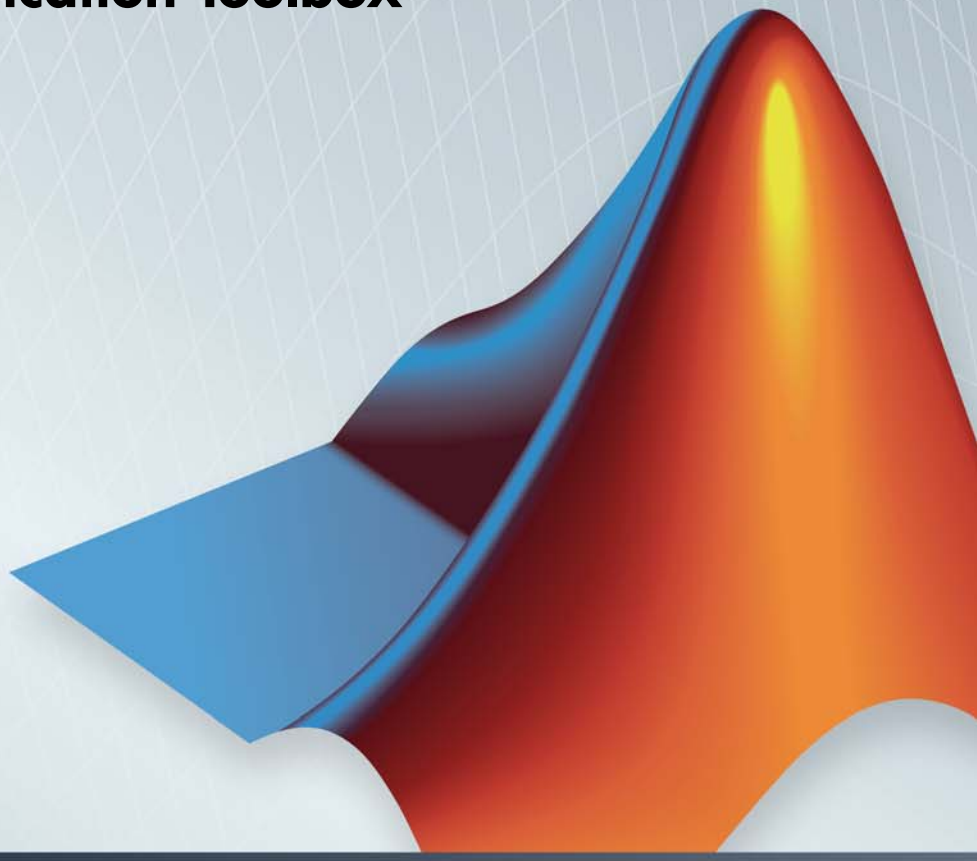
System Identification Toolbox™

User's Guide

R2013a

Lennart Ljung



MATLAB® & SIMULINK®

MathWorks®

How to Contact MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

System Identification Toolbox™ User's Guide

Revision History

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

Contents

1 Choosing Your System Identification Approach

2 Data Import and Processing

3 Linear Model Identification

4 Nonlinear Black-Box Model Identification

ODE Parameter Estimation (Grey-Box Modeling)

5

6 Time Series Identification

7 Recursive Model Identification

8 Model Analysis

9 Setting Toolbox Preferences

10 Control Design Applications

11 System Identification Toolbox Blocks

12 System Identification Tool GUI

Index

1

Choosing Your System Identification Approach

Acknowledgments

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

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

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

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

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

What Are Model Objects?

In this section...

Model Objects Represent Linear Systems

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

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

The main families of model objects are:

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

About Model Data

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

- Transfer functions store the numerator and denominator coefficients
- State-space models store the A, B, C, and D matrices that describe the dynamics of the system

- PID controller models store the proportional, integral, and derivative gains

Other model attributes stored as model data include time units, names for the model inputs or outputs, and time delays.

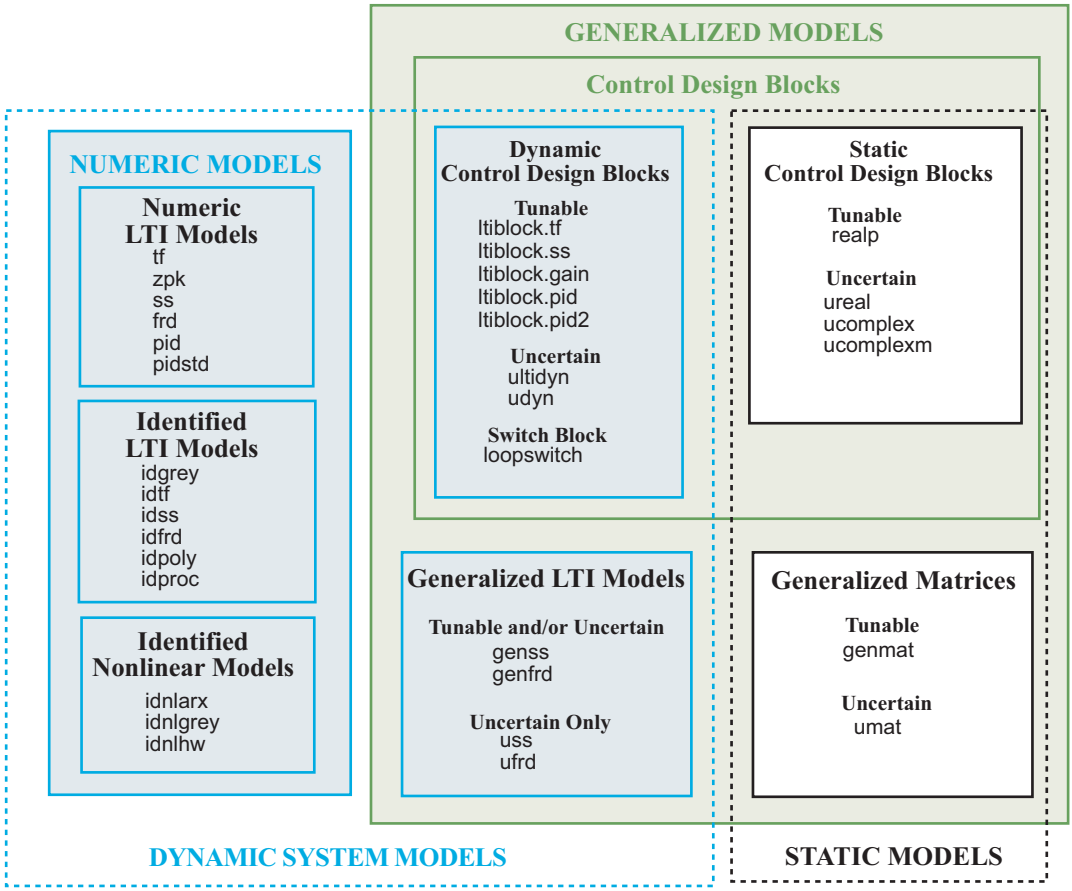
Note All model objects are MATLAB® objects, but working with them does not require a background in object-oriented programming. To learn more about objects and object syntax, see "Classes in the MATLAB Language" in the MATLAB documentation.

Concepts

- "Types of Model Objects" on page 1-5

Types of Model Objects

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



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

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

Concepts

- "What Are Model Objects?" on page 1-3
- "Dynamic System Models" on page 1-7
- "Numeric Models" on page 1-9

Dynamic System Models

Dynamic System Models generally represent systems that have internal dynamics or memory of past states (such as integrators).

Most commands for analyzing linear systems, such as `bode`, `margin`, and `ltiview`, work on most Dynamic System Model objects. For Generalized Models, analysis commands use the current value of tunable parameters and the nominal value of uncertain parameters.

The following table lists the Dynamic System Models.

Model Family	Model Types
Numeric LTI models — Basic numeric representation of linear systems	`tf`
	`zpk`
	`ss`
	`frd`
	`pid`
	`pidstd`
Identified LTI models — Representations of linear systems with tunable coefficients, whose values can be identified using measured input/output data.	`idtf`
	`idss`
	`idfrd`
	`idgrey`
	`idpoly`
	`idproc`
Identified nonlinear models — Representations of nonlinear systems with tunable coefficients, whose values can be identified using input/output data. Limited support for commands that analyze linear systems.	`idnlarx`
	`idnlhw`
	`idnlgrey`

Model Family	Model Types
Generalized LTI models — Representations of systems that include tunable or uncertain coefficients	`genss`
	`genfrd`
	`uss`
	`ufrd`
Dynamic Control Design Blocks — Tunable, uncertain, or switch components for constructing models of control systems	`ltiblock.gain`
	`ltiblock.tf`
	`ltiblock.ss`
	`ltiblock.pid`
	`ltiblock.pid2`
	`ultidyn`
	`udyn`
	`loopswitch`

Concepts

- "Numeric Linear Time Invariant (LTI) Models" on page 1-9
- "Identified LTI Models" on page 1-9
- "Identified Nonlinear Models" on page 1-10

Numeric Models

Numeric Linear Time Invariant (LTI) Models

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

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

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

Identified LTI Models

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

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

Model Type	Description
`idtf`	Transfer function model in polynomial form, with identifiable parameters
`idss`	State-space model, with identifiable parameters
`idpoly`	Polynomial input-output model, with identifiable parameters

Model Type	Description
`idproc`	Continuous-time process model, with identifiable parameters
`idfrd`	Frequency-response model, with identifiable parameters
`idgrey`	Linear ODE (grey-box) model, with identifiable parameters

Identified Nonlinear Models

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

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

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

About Identified Linear Models

In this section...

What are IDLTI Models?

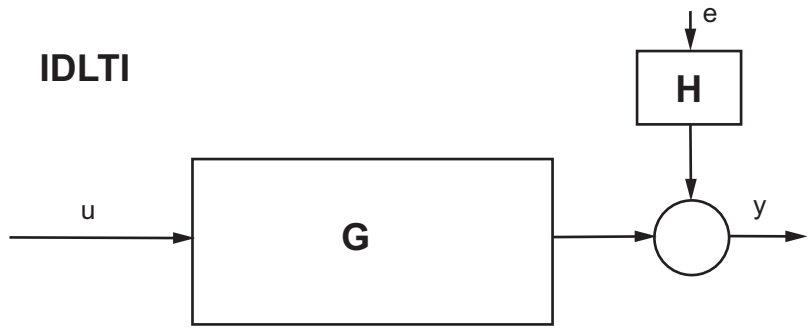
System Identification Toolbox software uses objects to represent a variety of linear and nonlinear model structures. These linear model objects are collectively known as *Identified Linear Time-Invariant* (IDLTI) models.

IDLTI models contain two distinct dynamic components:

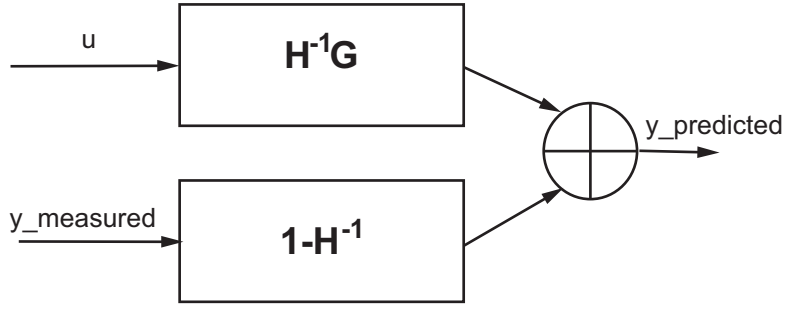
- **Measured component** — Describes the relationship between the measured inputs and the measured output (G)
- **Noise component** — Describes the relationship between the disturbances at the output and the measured output (H)

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

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



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



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

Measured and Noise Component Parameterizations

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

Model Type	Transfer Functions G and H	Configuration Method
State space model (`idss`)	Represents an identified state-space model structure, governed by the equations: $\dot{x} = Ax + Bu + Ke$ $y = Cx + Du + e$ where the transfer function between the measured input u and output y is $G(s) = C(sI - A)^{-1}B + D$ and the noise transfer function is $H(s) = C(sI - A)^{-1}K + I$.	Construction: Use `idss` to create a model, specifying values of state-space matrices A, B, C, D and K as input arguments (using `NaNs` to denote unknown entries). Estimation: Use `ssest` or `n4sid`, specifying name-value pairs for various configurations, such as, canonical parameterization of the measured dynamics (`'Form'/'canonical'`), denoting absence of feedthrough by fixing D to zero (`'Feedthrough'/false`), and absence of noise dynamics by fixing K to zero (`'DisturbanceModel'/'none'`).
Polynomial model (`idpoly`)	Represents a polynomial model such as ARX, ARMAX and BJ. An ARMAX model, for example, uses the input-output equation Ay(t) = Bu(t)+Ce(t), so that the measured transfer function G is $G(s) = A^{-1}B$, while the noise transfer function is $H(s) = A^{-1}C$. The ARMAX model is a special configuration of the general polynomial model whose governing equation is:	Construction: Use `idpoly` to create a model using values of active polynomials as input arguments. For example, to create an Output-Error model which uses `G = B/F` as the measured component and has a trivial noise component (`H = 1`). enter: `y = idpoly([],B,[],[],F)` Estimation: Use the `armax`, `arx`, or `bj`, specifying the orders

Model Type	Transfer Functions G and H	Configuration Method
	$$Ay(t) = \frac{B}{F}u(t) + \frac{C}{D}e(t)$$ The autoregressive component, A, is common between the measured and noise components. The polynomials B and F constitute the measured component while the polynomials C and D constitute the noise component.	of the polynomials as input arguments. For example, `bj` requires you to specify the orders of the B, C, D, and F polynomials to construct a model with governing equation $$y(t) = \frac{B}{F}u(t) + \frac{C}{D}e(t)$$
Transfer function model (`idtf`)	Represents an identified transfer function model, which has no dynamic elements to model noise behavior. This object uses the trivial noise model $H(s) = I$. The governing equation is $$y(t) = \frac{num}{den}u(t) + e(t)$$	Construction: Use `idtf` to create a model, specifying values of the numerator and denominator coefficients as input arguments. The numerator and denominator vectors constitute the measured component `G = num(s)/den(s)`. The noise component is fixed to `H = 1`. Estimation: Use `tfest`, specifying the number of poles and zeros of the measured component `G`.
Process model (`idproc`)	Represents a process model, which provides options to represent the noise dynamics as either first- or second-order ARMA process (that is, $H(s) = C(s)/A(s)$, where $C(s)$ and $A(s)$ are monic polynomials of equal degree). The measured component, $G(s)$, is represented	For process (and grey-box) models, the noise component is often treated as an on-demand extension to an otherwise measured component-centric representation. For these models, you can add a noise component by using the

Model Type	Transfer Functions G and H	Configuration Method
	by a transfer function expressed in pole-zero form.	`DisturbanceModel` estimation option. For example: `model = procest(data,'P1D')` estimates a model whose equation is: $$y(s) = K_p \frac{1}{(T_{p1}s+1)} e^{-sTd} u(s) + e(s).$$ To add a second order noise component to the model, use: `Options = procestOptions(`Disturbance`model = procest(data, `P1D', Options)` This model has the equation: $$y(s) = K_p \frac{1}{(T_{p1}s+1)} e^{-sTd} u(s) + \frac{1+c_1 s}{1+d_1 s} e(s)$$ where the coefficients *c1* and *d1* parameterize the noise component of the model. If you are constructing a process model using the `idproc` command, specify the structure of the measured component using the `Type` input argument and the noise component by using the `NoiseTF` name-value pair. For example, `model=idproc('P1','Kp',1,'Tp1',1,'Noi`

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

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

Linear Model Estimation

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

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

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

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

Black Box ("Cold Start") Estimation

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

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

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

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

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

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

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

Structured Estimations

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

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

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

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

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

Estimation Options

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

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

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

For information about how to view the list of options used to create an estimated model, see “Estimation Report” on page 1-19.

Estimation Report

The model returned by the estimation command contains information about the estimation operation in the `Report` property. Information includes:

- Information on initial conditions used for estimation
- Termination conditions for iterative estimation algorithms under a category called Termination
- Final prediction error (FPE), fit % and mean-square error (MSE) under a category called Fit
- The nature of estimation data under a category called DataUsed
- All estimated quantities - parameter values and their covariance, initial state values for state-space models and values of input level for process models under a category called Parameters
- The option set used for estimation under a category called OptionsUsed

To illustrate, this example displays the contents of the `Report.OptionsUsed` property of the model estimated in “Estimation Options” on page 1-18.

```
sys.Report

Status: 'Estimated using POLYEST with Focus = "prediction"'
               Method: 'OE'
    InitialCondition: 'zero'
                 Fit: [1x1 struct]
          Parameters: [1x1 struct]
         OptionsUsed: [1x1 idoptions.polyest]
           RandState: [1x1 struct]
            DataUsed: [1x1 struct]
         Termination: [1x1 struct]
```

Exploring one of the categories provides additional data:

```
sys.Report.Fit shows:

  FitPercent: 70.5666
         FPE: 1.7558
     LossFcn: 1.7094
         MSE: 1.6865
```

Linear Model Structures

In this section...

About System Identification Toolbox Model Objects

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

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

This toolbox includes nine classes for representing models. For example, `idpoly` represents linear input-output polynomial models, and `idss` represents linear state-space models. For a complete list of available model objects, see "Available Linear Models" on page 1-22 and "Available Nonlinear Models" on page 1-32.

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

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

linear model objects. However, other methods are unique to a specific model object. For example, the estimation method `canon` is unique to the state-space `idss` model.

Every class has a special method for creating objects of that class, called the *constructor*. Using a constructor creates an instance of the corresponding class or *instantiates the object*. The constructor name is the same as the class name. For example, `idpoly` is both the name of the class representing linear black-box polynomial models and the name of the constructor for instantiating the model object.

Available Linear Models

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

Model Type	Usage	Learn More
Transfer function (`idtf`)	Use this structure to represent transfer functions: $$y = \frac{num}{den}u + e$$ where *num* and *den* are polynomials of arbitrary lengths. You can specify initial guesses for, and estimate, *num*, *den*, and transport delays.	"Identifying Transfer Function Models" on page 3-113
Process model (`idproc`)	Use this structure to represent process models that are low order transfer functions expressed in pole-zero form. They include	"Identifying Process Models" on page 3-26

Model Type	Usage	Learn More
	integrator, delay, zero, and up to 3 poles.	
State-space model (`idss`)	Use this structure to represent known state-space structures and black-box structures. You can fix certain parameters to known values and estimate the remaining parameters. You can also prescribe minimum/maximum bounds on the values of the free parameters. If you need to specify parameter dependencies or parameterize the state-space matrices using your own parameters, use a grey-box model.	"Identifying State-Space Models" on page 3-79
Polynomial models (`idpoly`)	Use to represent linear transfer functions based on the general form input-output polynomial form: $$Ay = \frac{B}{F}u + \frac{C}{D}e$$ where A, B, C, D and F are polynomials with coefficients that the toolbox estimates from data.	"Identifying Input-Output Polynomial Models" on page 3-45

Model Type	Usage	Learn More
	Typically, you begin modeling using simpler forms of this generalized structure (such as ARX: $Ay = Bu + e$ and OE: $y = \frac{B}{F}u + e$) and, if necessary, increase the model complexity.	
Grey-box model (`idgrey`)	Use to represent arbitrary parameterizations of state-space models. For example, you can use this structure to represent your ordinary differential or difference equation (ODE) and to define parameter dependencies.	"Estimating Linear Grey-Box Models" on page 5-6

When to Construct a Model Structure Independently of Estimation

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

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

- Simulate or analyze the effect of model's parameters on its response, independent of estimation.

- Specify an initial guess for specific model parameter values before estimation. Specify bounds on parameter values, or set up the auxiliary model information in advance, or both. Auxiliary model information includes specifying input/output names, units, Notes, user data, etc.

In most cases, you can use the estimation commands to both construct and estimate the model—without having to construct the model object independently. For example, the estimation command `tfest` creates a transfer function model using data and some information on the order of the model - the number of poles and zeros. For information about how to both construct and estimate models with a single command, see "Model Estimation Commands" on page 1-40.

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

Commands for Constructing Model Structures

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

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

For information about how to both construct and estimate models with a single command, see "Model Estimation Commands" on page 1-40.

Summary of Model Constructors

Model Constructor	Resulting Model Class
`idfrd`	Nonparametric frequency-response model.
`idproc`	Continuous-time, low-order transfer functions (process models).

Summary of Model Constructors (Continued)

Model Constructor	Resulting Model Class
`idpoly`	Linear input-output polynomial models: • ARX • ARMAX • Output-Error • Box-Jenkins
`idss`	Linear state-space models.
`idtf`	Linear transfer function models.
`idgrey`	Linear ordinary differential or difference equations (grey-box models). You write a function that translates user parameters to state-space matrices. Can also be viewed as state-space models with user-specified parameterization.
`idnlgrey`	Nonlinear ordinary differential or difference equation (grey-box models). You write a function or MEX-file to represent the governing equations.
`idnlarx`	Nonlinear ARX models, which define the predicted output as a nonlinear function of past inputs and outputs.
`idnlhw`	Nonlinear Hammerstein-Wiener models, which include a linear dynamic system with nonlinear static transformations of inputs and outputs.

For more information about when to use these commands, see "When to Construct a Model Structure Independently of Estimation" on page 1-24.

Model Properties

Categories of Model Properties

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

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

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

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

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

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

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

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

Viewing Model Properties and Estimated Parameters

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

Task	Command	Example

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

Access a specific model property	Use dot notation	View the *A* matrix containing the estimated parameters in the previous model: `m_arx.a`
	For properties, such as `Report`, that are configured like structures, use dot notation of the form `model.PropertyName.FieldName`. `FieldName` is the name of any field of the property.	View the method used in ARX model estimation: `m_arx.Report.Method`
Access model uncertainty information	`getpvec` and `polydata`	View the *A* polynomial and 1 standard uncertainty of an ARX model: `[a,~,~,~,~,da] = polydata(m_arx)`

Task	Command	Example

Change model property values	`set` or dot notation	Change the input delays for all three input channels to `[1 1 1]` for an ARX model:
		
		`set(m_arx,'InputDelay',[1 1 1])`
		
		or
		
		`m_arx.InputDelay = [1 1 1]`

See Also

Validate each model directly after estimation to help fine-tune your modeling strategy. When you do not achieve a satisfactory model, you can try a different model structure and order, or try another identification algorithm. For more information about validating and troubleshooting models, see "Validating Models After Estimation" on page 8-3.

Imposing Constraints on Model Parameter Values

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

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

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

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

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

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

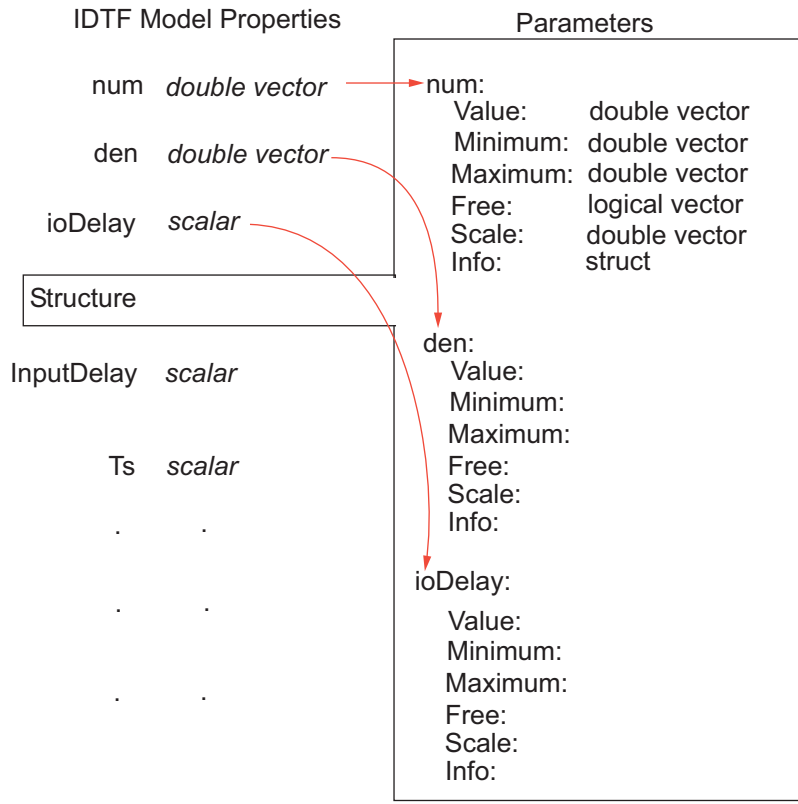
You can update the value of the `num` and `den` properties after you create the object as follows:

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

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

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

For a transfer function model, the `num`, `den`, and `ioDelay` model properties are simply pointers to the `Value` attribute of the corresponding parameter in the `Structure` property.



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

Available Nonlinear Models

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

Model Type	Usage	Learn More
Nonlinear ARX model (`idnlarx` object)	Use to represent nonlinear extensions of linear models. This structure allows you to model complex nonlinear behavior using flexible nonlinear functions, such as wavelet and sigmoid networks.	"Identifying Nonlinear ARX Models" on page 4-8
Linear models with input/output nonlinearities, or Hammerstein-Wiener model (`idnlhw` object)	Use to represent linear models with static nonlinearities.	"Identifying Hammerstein-Wiener Models" on page 4-48
Nonlinear grey-box model (`idnlgrey` object)	Use to represent nonlinear ODEs with unknown parameters.	"Estimating Nonlinear Grey-Box Models" on page 5-17

Recommended Model Estimation Sequence

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

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

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

Estimate impulse-response and frequency-response models first to gain insight into the system dynamics and assess whether a linear model is sufficient. Then, estimate parametric models in the following order:

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

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

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

   For more information, see "Identifying Input-Output Polynomial Models" on page 3-45 and "Identifying State-Space Models" on page 3-79.

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

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

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

For more information, see "Identifying Input-Output Polynomial Models" on page 3-45.

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

For general information about choosing you model strategy, see "Overview". For information about validating models, see "Validating Models After Estimation" on page 8-3.

Supported Models for Time- and Frequency-Domain Data

In this section...

Supported Models for Time-Domain Data

Continuous-Time Models

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

- Transfer function models. See "Identifying Transfer Function Models" on page 3-113.
- Process models. See "Identifying Process Models" on page 3-26.
- State-space models. See "Identifying State-Space Models" on page 3-79.

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

Discrete-Time Models

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

ODEs (Grey-Box Models)

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

Nonlinear Models

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

You can also estimate nonlinear grey-box models from time-domain data. See "Estimating Nonlinear Grey-Box Models" on page 5-17.

Supported Models for Frequency-Domain Data

There are two types of frequency-domain data:

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

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

Continuous-Time Models

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

- Transfer function models using continuous- or discrete-time data. See "Identifying Transfer Function Models" on page 3-113.
- Process models using continuous- or discrete-time data. See "Identifying Process Models" on page 3-26.
- Input-output polynomial models of output-error structure using continuous time data. See "Identifying Input-Output Polynomial Models" on page 3-45.
- State-space models using continuous- or discrete-time data.

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

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

Discrete-Time Models

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

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

ODEs (Grey-Box Models)

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

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

Nonlinear Black-Box Models

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

See Also

"Supported Continuous- and Discrete-Time Models" on page 1-38

Supported Continuous- and Discrete-Time Models

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

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

Supported Continuous-Time Models

Model Type	Description
"Identifying Transfer Function Models" on page 3-113	Estimate continuous-time transfer function models directly using `tfest` from either time- and frequency-domain data. If you estimated a discrete-time transfer function model from time-domain data, then use `d2c` to transform it into a continuous-time model.
Low-order transfer functions (process models)	Estimate low-order process models for up to three free poles from either time- or frequency-domain data.
Linear input-output polynomial models	To get a linear, continuous-time model of arbitrary structure from time-domain data, you can estimate a discrete-time model, and then use `d2c` to transform it into a continuous-time model. You can estimate only polynomial models of Output Error structure using continuous-time frequency domain data.. Other structures that include noise models, such as Box-Jenkins (BJ) and ARMAX, are not supported for frequency-domain data.

Supported Continuous-Time Models (Continued)

Model Type	Description
State-space models	Estimate continuous-time state-space models directly using the estimation commands from either time- and frequency-domain data. If you estimated a discrete-time state-space model from time-domain data, then use `d2c` to transform it into a continuous-time model.
Linear ODEs (grey-box) models	If the MATLAB file returns continuous-time model matrices, then estimate the ordinary differential equation (ODE) coefficients using either time- or frequency-domain data.
Nonlinear ODEs (grey-box) models	If the MATLAB file returns continuous-time output and state derivative values, estimate arbitrary differential equations (ODEs) from time-domain data.

Supported Discrete-Time Models

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

Model Estimation Commands

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

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

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

Commands for Constructing and Estimating Models

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

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

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

Modeling Multiple-Output Systems

In this section...

About Modeling Multiple-Output Systems

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

- "Modeling Multiple Outputs Directly" on page 1-42
- "Modeling Multiple Outputs as a Combination of Single-Output Models" on page 1-42

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

If you know that some of the outputs have poor accuracy and should be less important during estimation, you can control how much each output is weighed in the estimation. For more information, see "Improving Multiple-Output Estimation Results by Weighing Outputs During Estimation" on page 1-43.

Modeling Multiple Outputs Directly

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

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

Modeling Multiple Outputs as a Combination of Single-Output Models

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

Use this approach when no feedback is present in the dynamic system and there are no couplings between the outputs. If you are unsure about the presence of feedback, see "How to Analyze Data Using the advice Command" on page 2-94.

To construct partial models, use subreferencing to create partial data sets, such that each data set contains all inputs and one output. For more information about creating partial data sets, see the following sections in the *System Identification Toolbox User's Guide*:

- For working in the System Identification Tool GUI, see "Creating Data Sets from a Subset of Signal Channels" on page 2-37.
- For working at the command line, see the "Select Data Channels, I/O Data and Experiments in iddata Objects" on page 2-63.

After validating the single-output models, use vertical concatenation to combine these partial models into a single multiple-output model. For more information about concatenation, see "Increasing Number of Channels or Data Points of iddata Objects" on page 2-67 or "Adding Input or Output Channels in idfrd Objects" on page 2-80.

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

Improving Multiple-Output Estimation Results by Weighing Outputs During Estimation

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

For linear models, use the `OutputWeight` estimation to indicate the desired output weighting. If you set this option to `'noise'`, an automatic weighting, equal to the inverse of the estimated noise variance, is used for model estimation. You can also specify a custom weighting matrix, which must be a positive semi-definite matrix.

Note

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

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

For more information about the OutputWeight estimation option, see `arxOptions`, `ssestOptions`, `tfestOptions`, `procestOptions`, etc. For more information about the `Algorithm` fields for nonlinear estimation, see the `idnlarx` and `idnlhw` reference pages.

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

2

Data Import and Processing

Supported Data

System Identification Toolbox software supports estimation of linear models from both time- and frequency-domain data. For nonlinear models, this toolbox supports only time-domain data. For more information, see "Supported Models for Time- and Frequency-Domain Data" on page 1-35.

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

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

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

and a corresponding output sequence

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

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

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

Supported Data	Description
Time-domain I/O data	One or more input variables *u(t)* and one or more output variables *y(t)*, sampled as a function of time. Time-domain data can be either real or complex
Time-series data	Contains one or more outputs *y(t)* and no measured input. Can be time-domain or frequency-domain data.

Supported Data	Description
Frequency-domain data	Fourier transform of the input and output time-domain signals. The data is the set of input and output signals in frequency domain; the frequency grid need not be uniform.
Frequency-response data	Complex frequency-response values for a linear system characterized by its transfer function *G*, measurable directly using a spectrum analyzer. Also called *frequency function data*. Represented by `frd` or `idfrd` objects. The data sample time may be zero or nonzero. The frequency vector need not be uniformly spaced.

Note If your data is complex valued, see "Manipulating Complex-Valued Data" on page 2-144 for information about supported operations for complex data.

Ways to Obtain Identification Data

You can obtain identification data by:

- Measuring input and output signals from a physical system.

  Your data must capture the important system dynamics, such as dominant time constants. After measuring the signals, organize the data into variables, as described in "Representing Data in MATLAB Workspace" on page 2-9. Then, import it in the System Identification Tool GUI or represent it as a data object for estimating models at the command line.

- Generating an input signal with desired characteristics, such as a random Gaussian or binary signal or a sinusoid, using `idinput`. Then, generate an output signal using this input to simulate a model with known coefficients. For more information, see "Generating Data Using Simulation" on page 2-130.

  Using input/output data thus generated helps you study the impact of input signal characteristics and noise on estimation.

- Logging signals from Simulink® models.

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

Ways to Prepare Data for System Identification

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

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

Represent data for system identification

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

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

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

- For working at the command line, create an `iddata` or `idfrd` object.

  For time-domain or frequency-domain data, see “Representing Time- and Frequency-Domain Data Using iddata Objects” on page 2-55.

  For frequency-response data, see “Representing Frequency-Response Data Using idfrd Objects” on page 2-76.

- To simulate data with and without noise, see “Generating Data Using Simulation” on page 2-130.

Analyze data quality

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

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

  See “Analyzing Data Quality” on page 2-85.

- Using the `advice` command to analyze the data for the presence of constant offsets and trends, delay, possible feedback, and signal excitation levels.

  See “How to Analyze Data Using the advice Command” on page 2-94.

Preprocess data

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

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

  See "Handling Missing Data and Outliers" on page 2-100.

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

  See "Handling Offsets and Trends in Data" on page 2-104 for information about subtracting means and linear trends, and "Filtering Data" on page 2-120 for information about filtering.

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

  See "Resampling Data" on page 2-111 for information about decimating and interpolating values, and "Filtering Data" on page 2-120 for information about filtering.

Select a subset of your data

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

To learn more about selecting data, see "Selecting Subsets of Data" on page 2-96.

Combine data from multiple experiments

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

To learn more about creating multiple-experiment data sets, see "Creating Multiexperiment Data Sets in the GUI" on page 2-39 or "Creating Multiexperiment Data at the Command Line" on page 2-61.

Requirements on Data Sampling

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

The System Identification Tool GUI only supports uniformly sampled data.

The System Identification Toolbox product provides limited support for nonuniformly sampled data. For more information about specifying uniform and nonuniform time vectors, see "Constructing an iddata Object for Time-Domain Data" on page 2-56.

Representing Data in MATLAB Workspace

In this section...

Time-Domain Data Representation

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

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

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

To use time-domain data for identification, you must know the sampling interval. If you are working with uniformly sampled data, use the actual sampling interval from your experiment. Each data value is assigned a time instant, which is calculated from the start time and sampling interval. You can work with nonuniformly sampled data only at the command line by specifying a vector of time instants using the `SamplingInstants` property of `iddata`, as described in "Constructing an iddata Object for Time-Domain Data" on page 2-56.

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

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

After you have the variables in the MATLAB workspace, import them into the System Identification Tool GUI or create a data object for working at the command line. For more information, see "Importing Time-Domain Data into the GUI" on page 2-18 and "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-55.

Time-Series Data Representation

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

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

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

To use time-series data for identification, you also need the sampling interval. If you are working with uniformly sampled data, use the actual sampling interval from your experiment. Each data value is assigned a sample time, which is calculated from the start time and the sampling interval. If you are working with nonuniformly sampled data at the command line, you can specify a vector of time instants using the `iddata SamplingInstants` property, as described in "Constructing an iddata Object for Time-Domain Data" on page 2-56. Note that model estimation cannot be performed using non-uniformly sampled data.

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

After you have the variables in the MATLAB workspace, import them into the System Identification Tool GUI or create a data object for working at the command line. For more information, see "Importing Time-Domain Data into the GUI" on page 2-18 and "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-55.

For information about estimating time-series model parameters, see "Time-Series Model Identification".

Frequency-Domain Data Representation

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

Frequency-Domain Input/Output Signal Representation

What Is Frequency-Domain Input/Output Signal?. *Frequency-domain data* is the Fourier transform of the input and output time-domain signals. For continuous-time signals, the Fourier transform over the entire time axis is defined as follows:

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

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

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

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

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

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

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

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

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

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

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

How to Represent Frequency-Domain Data in MATLAB. You must organize frequency-domain data in the following format:

- Input and output
  - For single-input/single-output (SISO) data:

  - The input data must be a column vector containing the values

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

  - The output data must be a column vector containing the values

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

    k=1, 2, ..., N_f, where N_f is the number of frequencies.

- For multi-input/multi-output data with N_u inputs, N_y outputs and N_f frequency measurements:
  - The input data must be an N_f-by-N_u matrix
  - The output data must be an N_f-by-N_y matrix

- Frequencies
  - Must be a column vector.

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

After you have the variables in the MATLAB workspace, import them into the System Identification Tool GUI or create a data object for working at the command line. For more information, see "Importing Frequency-Domain Input/Output Signals into the GUI" on page 2-23 and "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-55.

Frequency-Response Data Representation

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

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

$$y = Gu$$

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

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

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

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

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

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

When the input to the system is a sinusoid of a specific frequency, the output is also a sinusoid with the same frequency. The amplitude of the output is $|G|$ times the amplitude of the input. The phase of the shifted from the input by $\varphi = \arg G$. G is evaluated at the frequency of the input sinusoid.

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

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

How to Represent Frequency-Response Data in MATLAB. You can represent frequency-response data in two ways:

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

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

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

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

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

After you have the variables in the MATLAB workspace, import them into the System Identification Tool GUI or create a data object for working at the command line. For more information about importing data into the GUI, see "Importing Frequency-Response Data into the GUI" on page 2-26. To learn more about creating a data object, see "Representing Frequency-Response Data Using idfrd Objects" on page 2-76.

Importing Data into the GUI

In this section...

Types of Data You Can Import into the GUI

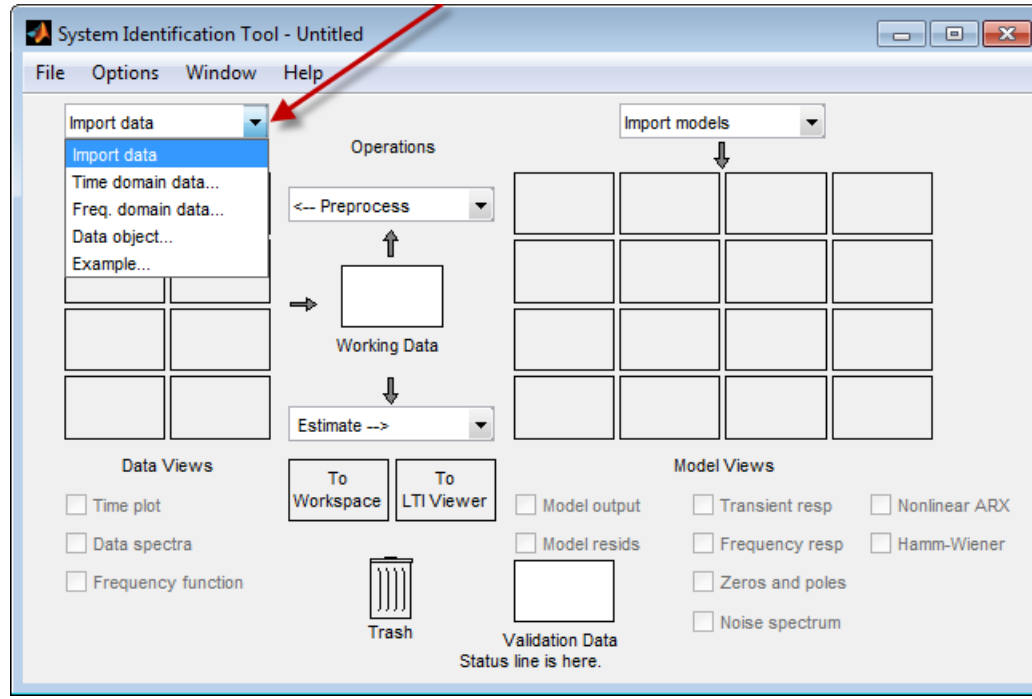
You can import the following types of data from the MATLAB workspace into the System Identification Tool GUI:

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

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

```
ident
```

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



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

Importing Time-Domain Data into the GUI

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

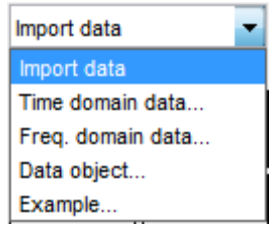
Note Your time-domain data must be sampled at equal time intervals. The input and output signals must have the same number of data samples.

To import data into the GUI:

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

   ```
   ident
   ```

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



3 Specify the following options:

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

- **Input** — Enter the MATLAB variable name (column vector or matrix) or a MATLAB expression that represents the input data. The expression must evaluate to a column vector or matrix.
- **Output** — Enter the MATLAB variable name (column vector or matrix) or a MATLAB expression that represents the output data. The expression must evaluate to a column vector or matrix.
- **Data name** — Enter the name of the data set, which appears in the System Identification Tool window after the import operation is completed.
- **Starting time** — Enter the starting value of the time axis for time plots.
- **Sampling interval** — Enter the actual sampling interval in the experiment. For more information about this setting, see "Specifying the Data Sampling Interval" on page 2-34.

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

---

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

Input Properties

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

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

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

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

Channel Names

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

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

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

Physical Units of Variables

- **Input** — Enter a string to specify the input units.

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

- **Output** — Enter a string to specify the output units.

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

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

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

Importing Frequency-Domain Data into the GUI

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

Importing Frequency-Domain Input/Output Signals into the GUI

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

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

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

To import data into the GUI:

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

   ```
   ident
   ```

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

3 Specify the following options:

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

  The frequency vector must have the same number of rows as the input and output signals.
- **Data name** — Enter the name of the data set, which appears in the System Identification Tool window after the import operation is completed.
- **Frequency unit** — Enter `Hz` for Hertz or keep the `rad/s` default value.
- **Sampling interval** — Enter the actual sampling interval in the experiment. For continuous-time data, enter `0`. For more information about this setting, see "Specifying the Data Sampling Interval" on page 2-34.

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

Input Properties

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

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

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

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

Channel Names

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

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

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

Physical Units of Variables

- **Input** — Enter a string to specify the input units.

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

- **Output** — Enter a string to specify the output units.

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

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

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

Importing Frequency-Response Data into the GUI

Prerequisite. Before you can import frequency-response data into the System Identification Tool GUI, you must import the data into the MATLAB workspace, as described in "Frequency-Response Data Representation" on page 2-13.

Importing Complex-Valued Frequency-Response Data. To import frequency-response data consisting of complex-valued frequency values at specified frequencies:

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

```
ident
```

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

3. In the **Data Format for Signals** list, select **Freq. Function (Complex)**.

4. Specify the following options:

   - **Freq. Func.** — Enter the MATLAB variable name or a MATLAB expression that represents the complex frequency-response data $G(e^{iw})$.
   - **Frequency** — Enter the MATLAB variable name of a vector or a MATLAB expression that represents the frequencies. The expression must evaluate to a column vector.
   - **Data name** — Enter the name of the data set, which appears in the System Identification Tool window after the import operation is completed.
   - **Frequency unit** — Enter `Hz` for Hertz or keep the `rad/s` default value.
   - **Sampling interval** — Enter the actual sampling interval in the experiment. For continuous-time data, enter `0`. For more information about this setting, see "Specifying the Data Sampling Interval" on page 2-34.

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

   Channel Names

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

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

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

Physical Units of Variables

- **Input** — Enter a string to specify the input units.

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

- **Output** — Enter a string to specify the output units.

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

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

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

Importing Amplitude and Phase Frequency-Response Data. To import frequency-response data consisting of amplitude and phase values at specified frequencies:

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

```
ident
```

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

3 In the **Data Format for Signals** list, select **Freq. Function (Amp/Phase)**.

4 Specify the following options:

- **Amplitude** — Enter the MATLAB variable name or a MATLAB expression that represents the amplitude $|G|$.
- **Phase (deg)** — Enter the MATLAB variable name or a MATLAB expression that represents the phase $\varphi = \arg G$.
- **Frequency** — Enter the MATLAB variable name of a vector or a MATLAB expression that represents the frequencies. The expression must evaluate to a column vector.
- **Data name** — Enter the name of the data set, which appears in the System Identification Tool window after the import operation is completed.
- **Frequency unit** — Enter `Hz` for Hertz or keep the `rad/s` default value.
- **Sampling interval** — Enter the actual sampling interval in the experiment. For continuous-time data, enter `0`. For more information about this setting, see "Specifying the Data Sampling Interval" on page 2-34.

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

Channel Names

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

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

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

Physical Units of Variables

- **Input** — Enter a string to specify the input units.

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

- **Output** — Enter a string to specify the output units.

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

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

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

Importing Data Objects into the GUI

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

Before you can import a data object into the System Identification Tool GUI, you must create the data object in the MATLAB workspace, as described in "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-55 or "Representing Frequency-Response Data Using idfrd Objects" on page 2-76.

Note You can also import a Control System Toolbox `frd` object. Importing an `frd` object converts it to an `idfrd` object.

Select **Import data > Data object** to open the Import Data dialog box.

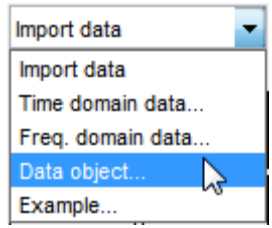
Import `iddata`, `idfrd`, or `frd` data object in the MATLAB workspace.

To import a data object into the GUI:

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

```
ident
```

2 In the System Identification Tool window, select **Import data > Data object**.



This action opens the Import Data dialog box. **IDDATA or IDFRD/FRD** is already selected in the **Data Format for Signals** list.

3 Specify the following options:

- **Object** — Enter the name of the MATLAB variable that represents the data object in the MATLAB workspace. Press **Enter**.
- **Data name** — Enter the name of the data set, which appears in the System Identification Tool window after the import operation is completed.
- (Only for time-domain `iddata` object) **Starting time** — Enter the starting value of the time axis for time plots.
- (Only for frequency domain `iddata` or `idfrd` object) **Frequency unit** — Enter the frequency unit for response plots.
- **Sampling interval** — Enter the actual sampling interval in the experiment. For more information about this setting, see "Specifying the Data Sampling Interval" on page 2-34.

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

---

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

(Only for `iddata` object) **Input Properties**

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

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

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

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

Channel Names

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

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

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

Physical Units of Variables

- **Input** — Enter a string to specify the input units.

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

- **Output** — Enter a string to specify the output units.

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

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

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

Specifying the Data Sampling Interval

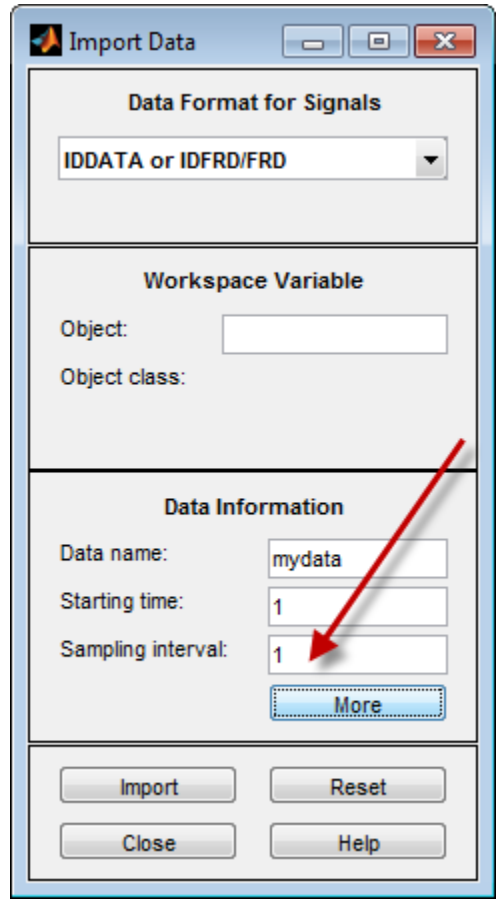
When you import data into the GUI, you must specify the data sampling interval.

The *sampling interval* is the time between successive data samples in your experiment and must be the numerical time interval at which your data is sampled in any units. For example, enter `0.5` if your data was sampled every `0.5 s`, and enter `1` if your data was sampled every `1 s`.

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

The sampling interval is used during model estimation. For time-domain data, the sampling interval is used together with the start time to calculate the sampling time instants. When you transform time-domain signals to frequency-domain signals (see the `fft` reference page), the Fourier transforms are computed as discrete Fourier transforms (DFTs) for this sampling

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



Sampling Interval in the Import Data dialog box

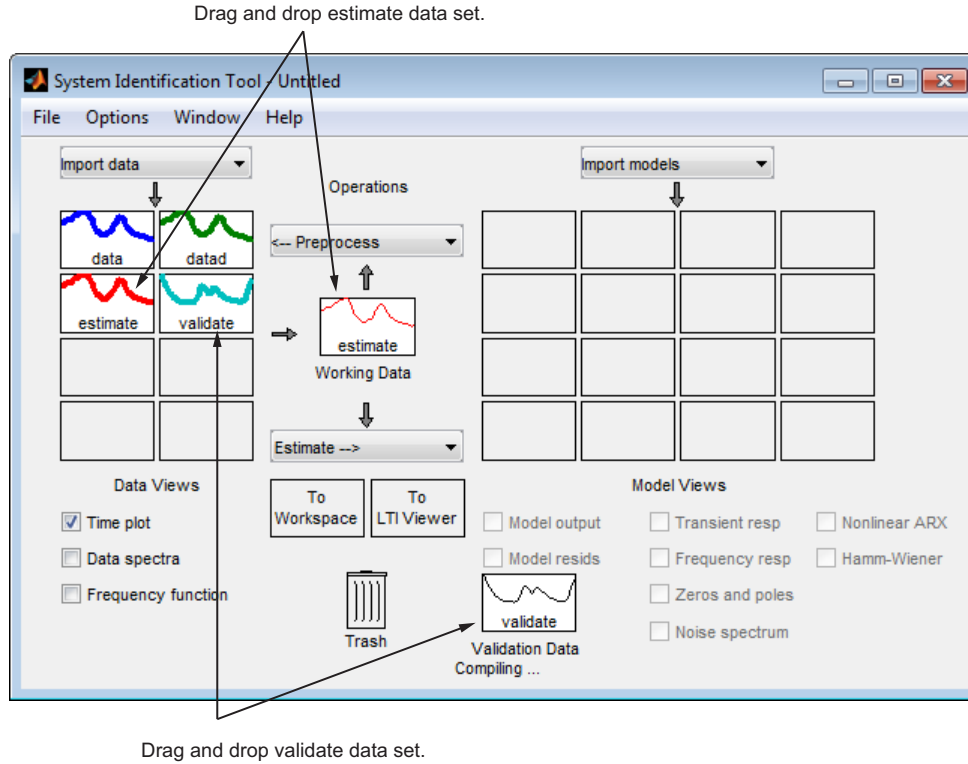
Specifying Estimation and Validation Data

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

In the System Identification Tool GUI, **Working Data** refers to estimation data. Similarly, **Validation Data** refers to the data set you use to validate a

model. For example, when you plot the model output, the input to the model is the input signal from the validation data set. This plot compares model output to the measured output in the validation data set. Selecting **Model resids** performs residual analysis using the validation data.

To specify **Working Data**, drag and drop the corresponding data icon into the **Working Data** rectangle, as shown in the following figure. Similarly, to specify **Validation Data**, drag and drop the corresponding data icon into the **Validation Data** rectangle.



Preprocessing Data Using Quick Start

As a preprocessing shortcut for time-domain data, select **Preprocess > Quick start** to simultaneously perform the following four actions:

- Subtract the mean value from each channel.

  Note For information about when to subtract mean values from the data, see "Handling Offsets and Trends in Data" on page 2-104.

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

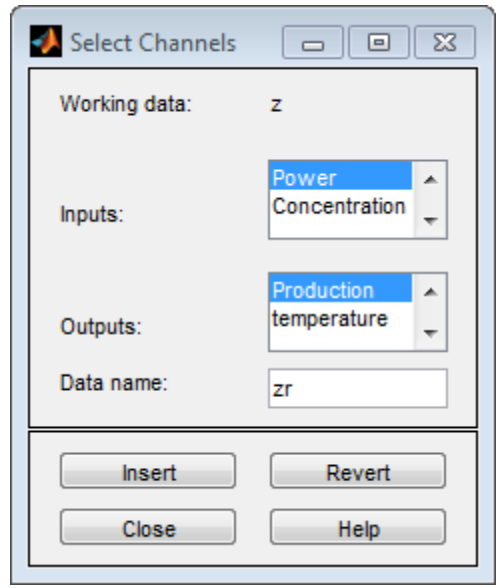
Creating Data Sets from a Subset of Signal Channels

You can create a new data set in the System Identification Tool GUI by extracting subsets of input and output channels from an existing data set.

To create a new data set from selected channels:

1. In the System Identification Tool GUI, drag the icon of the data from which you want to select channels to the **Working Data** rectangle.

2 Select **Preprocess > Select channels** to open the Select Channels dialog box.



The **Inputs** list displays the input channels and the **Outputs** list displays the output channels in the selected data set.

3 In the **Inputs** list, select one or more channels in any of following ways:

- Select one channel by clicking its name.
- Select adjacent channels by pressing the **Shift** key while clicking the first and last channel names.
- Select nonadjacent channels by pressing the **Ctrl** key while clicking each channel name.

Tip To exclude input channels and create time-series data, clear all selections by holding down the **Ctrl** key and clicking each selection. To reset selections, click **Revert**.

---

4 In the **Outputs** list, select one or more channels in any of following ways:

- Select one channel by clicking its name.
- Select adjacent channels by pressing the **Shift** key while clicking the first and last channel names.
- Select nonadjacent channels by pressing the **Ctrl** key while clicking each channel name.

Tip To reset selections, click **Revert**.

5 In the **Data name** field, type the name of the new data set. Use a name that is unique in the Data Board.

6 Click **Insert** to add the new data set to the Data Board in the System Identification Tool GUI.

7 Click **Close**.

Creating Multiexperiment Data Sets in the GUI

Why Create Multiexperiment Data?

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

Experiments can mean data that was collected during different sessions, or portions of the data collected during a single session. In the latter situation, you can create multiexperiment data by splitting a single data set into

multiple segments that exclude corrupt data, and then merge the good data segments.

Limitations on Data Sets

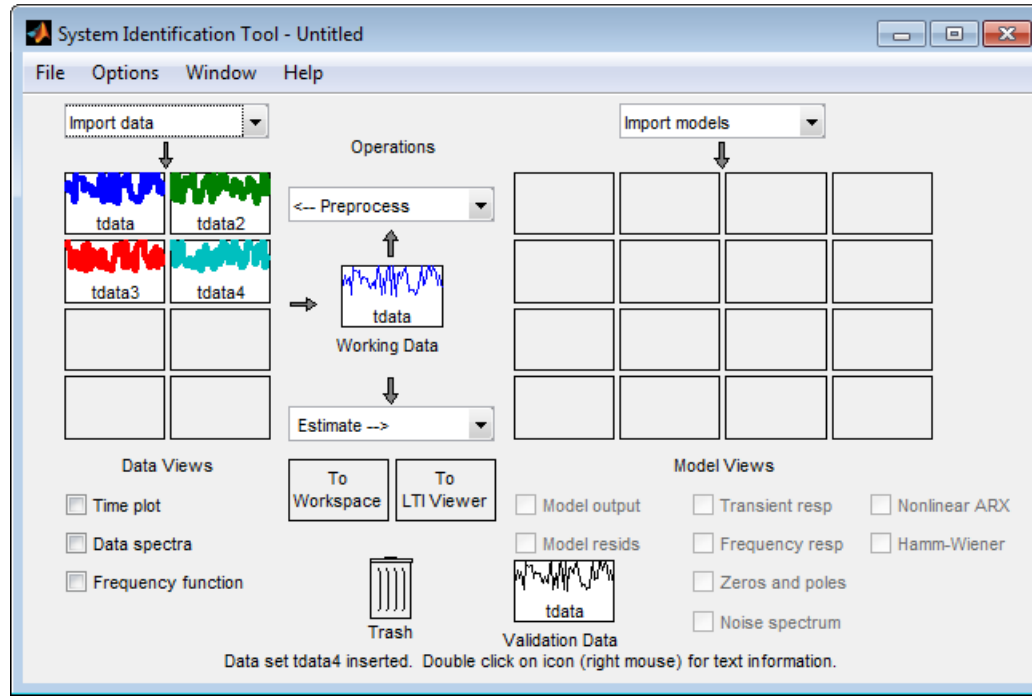
You can only merge data sets that have *all* of the following characteristics:

- Same number of input and output channels.
- Different names. The name of each data set becomes the experiment name in the merged data set.
- Same input and output channel names.
- Same data domain (that is, time-domain data or frequency-domain data only).

Merging Data Sets

You can merge data sets using the System Identification Tool GUI.

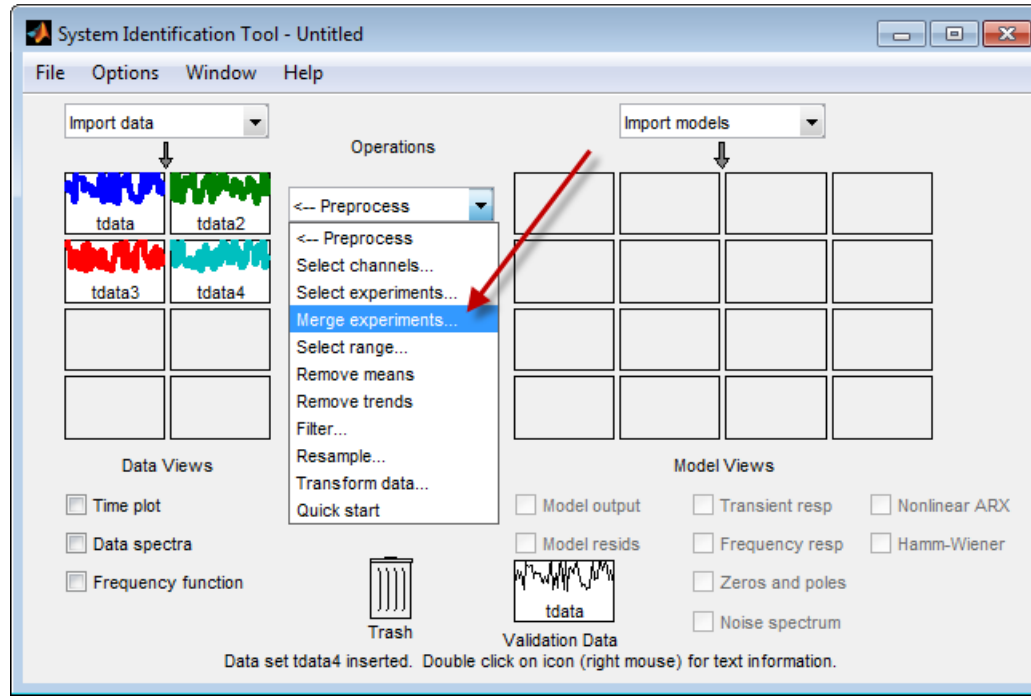
For example, suppose that you want to combine the data sets `tdata`, `tdata2`, `tdata3`, `tdata4` shown in the following figure.



GUI Contains Four Data Sets to Merge

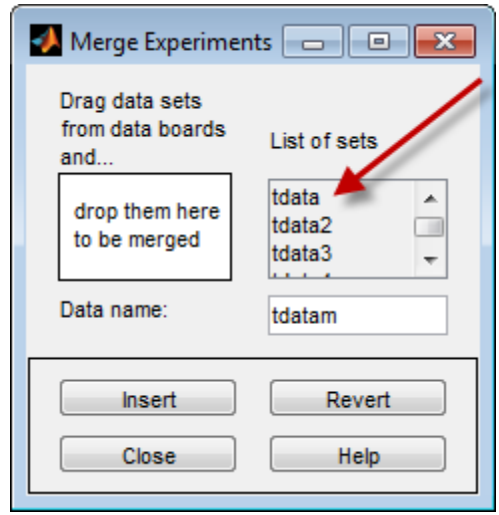
To merge data sets in the GUI:

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



2 In the System Identification Tool window, drag a data set icon to the Merge Experiments dialog box,to the **drop them here to be merged** rectangle.

The name of the data set is added to the **List of sets**. Repeat for each data set you want to merge.



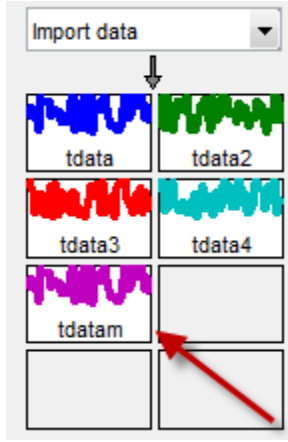
tdata and tdata2 to Be Merged

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

---

3 In the **Data name** field, type the name of the new data set. This name must be unique in the Data Board.

4 Click **Insert** to add the new data set to the Data Board in the System Identification Tool window.



Data Board Now Contains tdatam with Merged Experiments

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

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

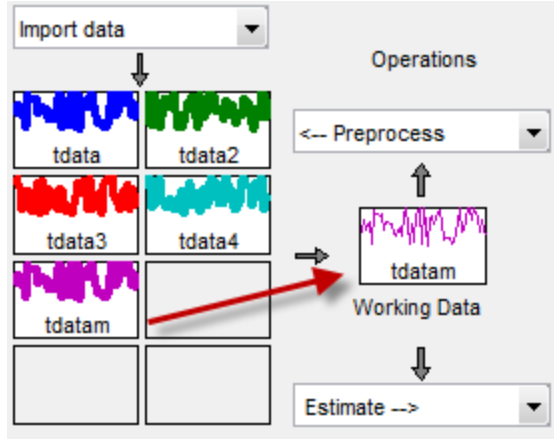
Extracting Specific Experiments from a Multiexperiment Data Set into a New Data Set

When a data set already consists of several experiments, you can extract one or more of these experiments into a new data set, using the System Identification Tool GUI.

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

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

1 In the System Identification Tool window, drag and drop the `tdatam` data icon to the **Working Data** rectangle.

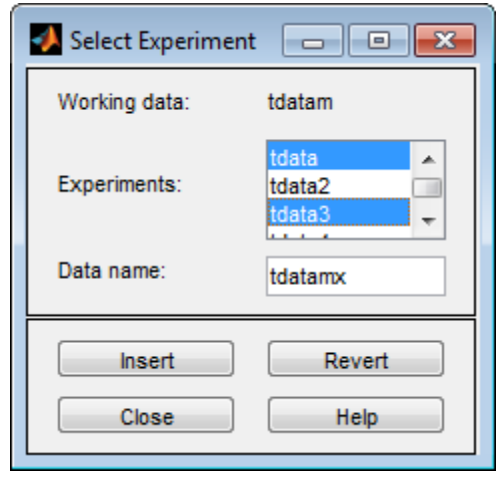


tdatam Is Set to Working Data

2 In the **Operations** area, select **Preprocess > Select experiments** from the drop-down menu to open the Select Experiment dialog box.

3 In the **Experiments** list, select one or more data sets in either of the following ways:

- Select one data set by clicking its name.
- Select adjacent data sets by pressing the **Shift** key while clicking the first and last names.
- Select nonadjacent data sets by pressing the **Ctrl** key while clicking each name.



4 In the **Data name** field, type the name of the new data set. This name must be unique in the Data Board.

5 Click **Insert** to add the new data set to the Data Board in the System Identification Tool GUI.

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

Managing Data in the GUI

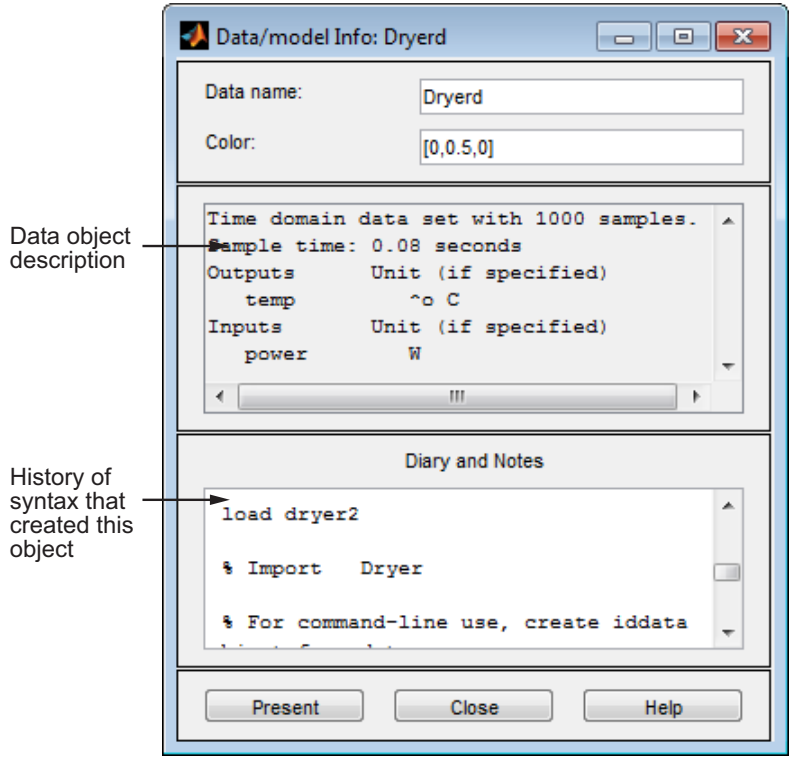
- "Viewing Data Properties" on page 2-48
- "Renaming Data and Changing Display Color" on page 2-49
- "Distinguishing Data Types" on page 2-51

Viewing Data Properties

You can get information about each data set in the System Identification Tool GUI by right-clicking the corresponding data icon.

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

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



To displays the data properties in the MATLAB Command Window, click **Present**.

Renaming Data and Changing Display Color

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

The Data/model Info dialog box opens. This dialog box describes both the contents and the properties of the data. The object description area displays the syntax of the operations you used to create the data in the GUI.

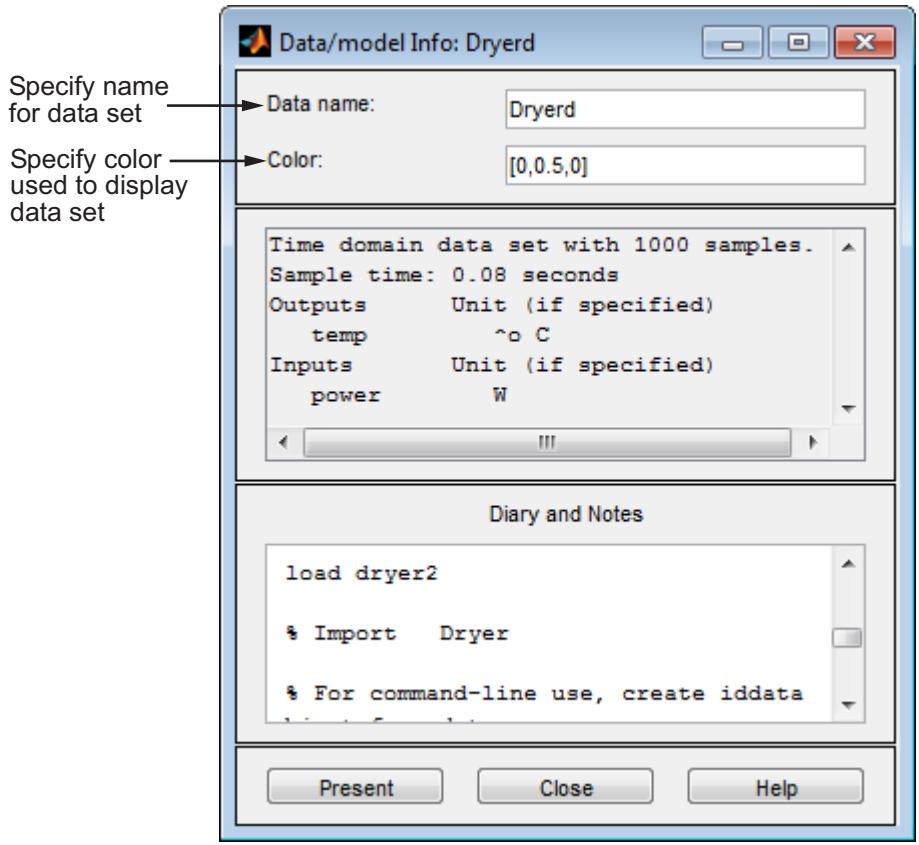
The Data/model Info dialog box also lets you rename the data by entering a new name in the **Data name** field.

You can also specify a new display color using three RGB values in the **Color** field. Each value is between `0` to `1` and indicates the relative presence of red, green, and blue, respectively. For more information about specifying default data color, see "Customizing the System Identification Tool GUI" on page 12-17.

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

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

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



Information About the Data

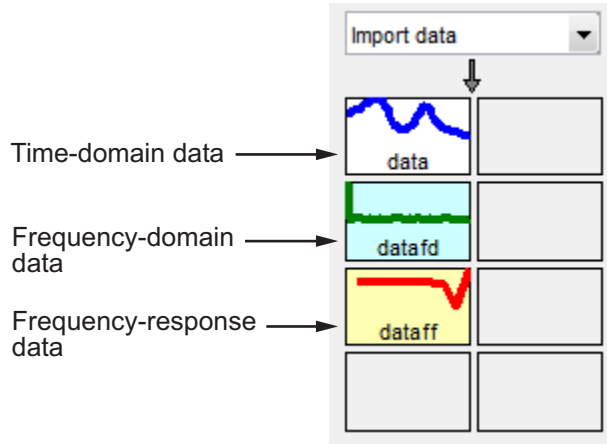
You can enter comments about the origin and state of the data in the **Diary And Notes** area. For example, you might want to include the experiment name, date, and the description of experimental conditions. When you estimate models from this data, these notes are associated with the models.

Clicking **Present** display portions of this information in the MATLAB Command Window.

Distinguishing Data Types

The background color of a data icon is color-coded, as follows:

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



Colors Representing Type of Data

Organizing Data Icons

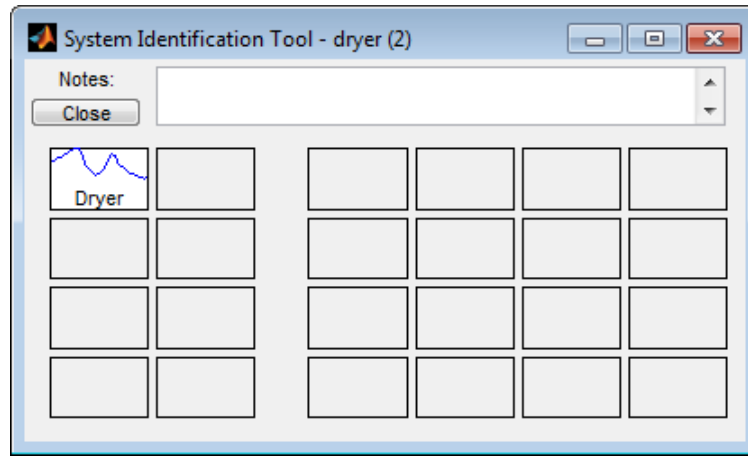
You can rearrange data icons in the System Identification Tool GUI by dragging and dropping the icons to empty Data Board rectangles in the GUI.

Note You cannot drag and drop a data icon into the model area on the right.

When you need additional space for organizing data or model icons, select **Options > Extra model/data board** in the System Identification Tool GUI. This action opens an extra session window with blank rectangles for data and models. The new window is an extension of the current session and does not represent a new session.

Tip When you import or create data sets and there is insufficient space for the icons, an additional session window opens automatically.

You can drag and drop data between the main System Identification Tool GUI and any extra session windows.



Type comments in the **Notes** field to describe the data sets. When you save a session, as described in "Saving, Merging, and Closing Sessions" on page 12-6, all additional windows and notes are also saved.

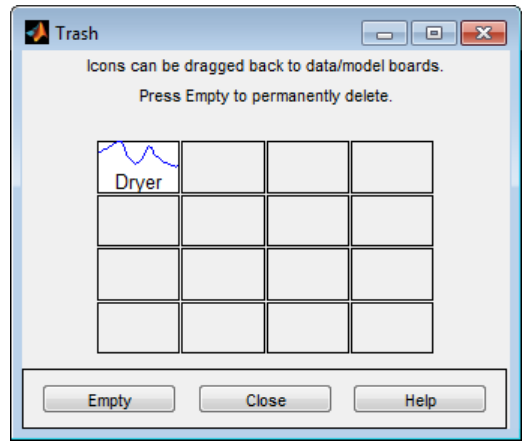
Deleting Data Sets

To delete data sets in the System Identification Tool GUI, drag and drop the corresponding icon into **Trash**. Moving items to **Trash** does not permanently delete these items.

Note You cannot delete a data set that is currently designated as **Working Data** or **Validation Data**. You must first specify a different data set in the System Identification Tool GUI to be **Working Data** or **Validation Data**, as described in "Specifying Estimation and Validation Data" on page 2-35.

To restore a data set from **Trash**, drag its icon from **Trash** to the Data or Model Board in the System Identification Tool window. You can view the **Trash** contents by double-clicking the **Trash** icon.

Note You must restore data to the Data Board; you cannot drag data icons to the Model Board.



To permanently delete all items in **Trash**, select **Options > Empty trash**.

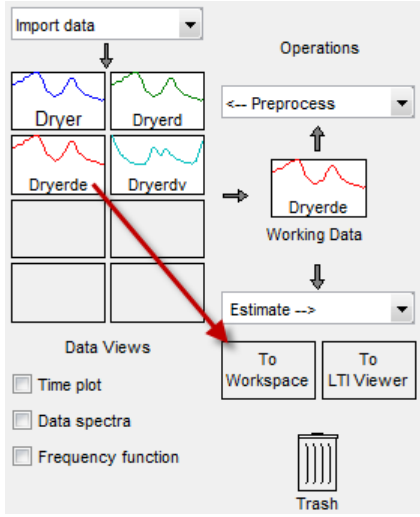
Exiting a session empties the **Trash** automatically.

Exporting Data to the MATLAB Workspace

The data you create in the System Identification Tool GUI is not available in the MATLAB workspace until you export the data set. Exporting to the MATLAB workspace is necessary when you need to perform an operation on the data that is only available at the command line.

To export a data set to the MATLAB workspace, drag and drop the corresponding icon to the **To Workspace** rectangle.

When you export data to the MATLAB workspace, the resulting variables have the same name as in the System Identification Tool GUI. For example, the following figure shows how to export the time-domain data object `datad`.



Exporting Data to the MATLAB® Workspace

In this example, the MATLAB workspace contains a variable named `data` after export.

Representing Time- and Frequency-Domain Data Using iddata Objects

In this section...

iddata Constructor

Requirements for Constructing an iddata Object

To construct an `iddata` object, you must have already imported data into the MATLAB workspace, as described in "Representing Data in MATLAB Workspace" on page 2-9.

Constructing an iddata Object for Time-Domain Data

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

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

You can also specify additional properties, as follows:

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

For more information about accessing object properties, see “Properties”.

In this example, `Ts` is the sampling time, or the time interval, between successive data samples. For uniformly sampled data, `Ts` is a scalar value equal to the sampling interval of your experiment. The default time unit is seconds, but you can set it to a new value using the `TimeUnit` property. For more information about `iddata` time properties, see “Modifying Time and Frequency Vectors” on page 2-69.

For nonuniformly sampled data, specify `Ts` as `[]`, and set the value of the `SamplingInstants` property as a column vector containing individual time values. For example:

```
data = iddata(y,u,Ts,[],'SamplingInstants',TimeVector)
```

Where `TimeVector` represents a vector of time values.

Note You can modify the property `SamplingInstants` by setting it to a new vector with the length equal to the number of data samples.

To represent time-series data, use the following syntax:

```
ts_data = iddata(y,[],Ts)
```

where `y` is the output data, `[]` indicates empty input data, and `Ts` is the sampling interval.

The following example shows how to create an `iddata` object using single-input/single-output (SISO) data from `dryer2.mat`. The input and output each contain 1000 samples with the sampling interval of `0.08` second.

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

MATLAB returns the following output:

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

Outputs     Unit (if specified)
   y1

Inputs      Unit (if specified)
   u1
```

The default channel name `'y1'` is assigned to the first and only output channel. When `y2` contains several channels, the channels are assigned default names `'y1','y2','y2',...,'yn'`. Similarly, the default channel name `'u1'` is assigned to the first and only input channel. For more information about naming channels, see "Naming, Adding, and Removing Data Channels" on page 2-73.

Constructing an iddata Object for Frequency-Domain Data

Frequency-domain data is the Fourier transform of the input and output signals at specific frequency values. To represent frequency-domain data, use the following syntax to create the `iddata` object:

```
data = iddata(y,u,Ts,'Frequency',w)
```

`'Frequency'` is an `iddata` property that specifies the frequency values `w`, where `w` is the frequency column vector that defines the frequencies at which the Fourier transform values of `y` and `u` are computed. `Ts` is the time interval between successive data samples in seconds for the original time-domain data. `w`, `y`, and `u` have the same number of rows.

Note You must specify the frequency vector for frequency-domain data.

For more information about `iddata` time and frequency properties, see "Modifying Time and Frequency Vectors" on page 2-69.

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

You can specify additional properties when you create the `iddata` object, as follows:

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

For more information about accessing object properties, see "Properties".

iddata Properties

To view the properties of the `iddata` object, use the `get` command. For example, type the following commands at the prompt:

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

MATLAB returns the following object properties and values:

```
          Domain: 'Time'
            Name: []
      OutputData: [1000x1 double]
               y: 'Same as OutputData'
      OutputName: {'y1'}
      OutputUnit: {''}
       InputData: [1000x1 double]
               u: 'Same as InputData'
       InputName: {'u1'}
       InputUnit: {''}
          Period: Inf
     InterSample: 'zoh'
              Ts: 0.0800
          Tstart: []
SamplingInstants: [1000x0 double]
        TimeUnit: ''
  ExperimentName: 'Exp1'
           Notes: []
        UserData: []
```

For a complete description of all properties, see the `iddata` reference page.

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

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

To change property values for an existing `iddata` object, use the `set` command or dot notation. For example, to change the sampling interval to `0.05`, type the following at the prompt:

```
set(data,'Ts',0.05)
```

or equivalently:

```
data.ts = 0.05
```

Property names are not case sensitive. You do not need to type the entire property name if the first few letters uniquely identify the property.

Tip You can use `data.y` as an alternative to `data.OutputData` to access the output values, or use `data.u` as an alternative to `data.InputData` to access the input values.

An `iddata` object containing frequency-domain data includes frequency-specific properties, such as `Frequency` for the frequency vector and `Units` for frequency units (instead of `Tstart` and `SamplingIntervals`).

To view the property list, type the following command sequence at the prompt:

```
% Load input u2 and output y2
  load dryer2;
% Create iddata object
  data = iddata(y2,u2,0.08);
% Take the Fourier transform of the data
% transforming it to frequency domain
  data = fft(data)
% Get property values of data
  get(data)
```

MATLAB returns the following object properties and values:

```
            Domain: 'Frequency'
              Name: []
        OutputData: [501x1 double]
                 y: 'Same as OutputData'
        OutputName: {'y1'}
        OutputUnit: {''}
         InputData: [501x1 double]
                 u: 'Same as InputData'
         InputName: {'u1'}
         InputUnit: {''}
            Period: Inf
       InterSample: 'zoh'
                Ts: 0.0800
             Units: 'rad/s'
         Frequency: [501x1 double]
          TimeUnit: ''
    ExperimentName: 'Exp1'
             Notes: []
          UserData: []
```

Creating Multiexperiment Data at the Command Line

Why Create Multiexperiment Data Sets?

You can create `iddata` objects that contain several experiments. Identifying models for an `iddata` object with multiple experiments results in an *average* model.

In the System Identification Toolbox product, *experiments* can either mean data collected during different sessions, or portions of the data

collected during a single session. In the latter situation, you can create a multiexperiment `iddata` object by splitting the data from a single session into multiple segments to exclude bad data, and merge the good data portions.

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

Limitations on Data Sets

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

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

Entering Multiexperiment Data Directly

To construct an `iddata` object that includes *N* data sets, you can use this syntax:

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

where `y`, `u`, and `Ts` are 1-by-*N* cell arrays containing data from the different experiments. Similarly, when you specify `Tstart`, `Period`, `InterSample`, and `SamplingInstants` properties of the `iddata` object, you must assign their values as 1-by-*N* cell arrays.

Merging Data Sets

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

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

These commands create an `iddata` object that conatains two experiments, where the experiments are assigned default names `'Exp1'` and `'Exp2'`, respectively.

Adding Experiments to an Existing iddata Object

You can add experiments individually to an `iddata` object as an alternative approach to merging data sets.

For example, to add the experiments in the `iddata` object `dat4` to `data`, use the following syntax:

```
data(:,:,:,'Run4') = dat4
```

This syntax explicitly assigns the experiment name `'Run4'` to the new experiment. The `Experiment` property of the `iddata` object stores experiment names.

For more information about subreferencing experiments in a multiexperiment data set, see “Subreferencing Experiments” on page 2-66.

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

Subreferencing Input and Output Data

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

Use the following general syntax to subreference specific data values in `iddata` objects:

```
data(samples,outputchannels,inputchannels,experimentname)
```

In this syntax, `samples` specify one or more sample indexes, `outputchannels` and `inputchannels` specify channel indexes or channel names, and `experimentname` specifies experiment indexes or names.

For example, to retrieve samples `5` through `30` in the `iddata` object `data` and store them in a new `iddata` object `data_sub`, use the following syntax:

```
data_sub = data(5:30)
```

You can also use logical expressions to subreference data. For example, to retrieve all data values from a single-experiment data set that fall between sample instants `1.27` and `9.3` in the `iddata` object `data` and assign them to `data_sub`, use the following syntax:

```
data_sub = data(data.sa>1.27&data.sa<9.3)
```

Note You do not need to type the entire property name. In this example, `sa` in `data.sa` uniquely identifies the `SamplingInstants` property.

You can retrieve the input signal from an `iddata` object using the following commands:

```
u = get(data,'InputData')
```

or

```
data.InputData
```

or

```
data.u    % u is the abbreviation for InputData
```

Similarly, you can retrieve the output data using

```
data.OutputData
```

or

```
data.y    % y is the abbreviation for OutputData
```

Subreferencing Data Channels

Use the following general syntax to subreference specific data channels in `iddata` objects:

```
data(samples,outputchannels,inputchannels,experiment)
```

In this syntax, `samples` specify one or more sample indexes, `outputchannels` and `inputchannels` specify channel indexes or channel names, and `experimentname` specifies experiment indexes or names.

To specify several channel names, you must use a cell array of name strings.

For example, suppose the `iddata` object `data` contains three output channels (named `y1`, `y2`, and `y3`), and four input channels (named `u1`, `u2`, `u3`, and `u4`). To select all data samples in `y3`, `u1`, and `u4`, type the following command at the prompt:

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

or equivalently

```
% Use channel indexes 1 and 4
% to reference the input channels
  data_sub = data(:,3,[1 4])
```

Tip Use a colon (`:`) to specify all samples or all channels, and the empty matrix (`[]`) to specify no samples or no channels.

If you want to create a time-series object by extracting only the output data from an `iddata` object, type the following command:

```
data_ts = data(:,:,[])
```

You can assign new values to subreferenced variables. For example, the following command assigns the first 10 values of output channel `1` of `data` to values in samples `101` through `110` in the output channel `2` of `data1`. It also assigns the values in samples `101` through `110` in the input channel `3` of `data1` to the first 10 values of input channel `1` of `data`.

```
data(1:10,1,1) = data1(101:110,2,3)
```

Subreferencing Experiments

Use the following general syntax to subreference specific experiments in `iddata` objects:

```
data(samples,outputchannels,inputchannels,experimentname)
```

In this syntax, `samples` specify one or more sample indexes, `outputchannels` and `inputchannels` specify channel indexes or channel names, and `experimentname` specifies experiment indexes or names.

When specifying several experiment names, you must use a cell array of name strings. The `iddata` object stores experiments name in the `ExperimentName` property.

For example, suppose the `iddata` object `data` contains five experiments with default names, `Exp1`, `Exp2`, `Exp3`, `Exp4`, and `Exp5`. Use the following syntax to subreference the first and fifth experiment in `data`:

```
data_sub = data(:,:,:,{'Exp1','Exp5'}) % Using experiment name
```

or

```
data_sub = data(:,:,:,[1 5])           % Using experiment index
```

Tip Use a colon (`:`) to denote all samples and all channels, and the empty matrix (`[]`) to specify no samples and no channels.

Alternatively, you can use the `getexp` command. The following example shows how to subreference the first and fifth experiment in `data`:

```
data_sub = getexp(data,{'Exp1','Exp5'}) % Using experiment name
```

or

```
data_sub = getexp(data,[1 5])           % Using experiment index
```

The following example shows how to retrieve the first 100 samples of output channels 2 and 3 and input channels 4 to 8 of Experiment 3:

```
dat(1:100,[2,3],[4:8],3)
```

Increasing Number of Channels or Data Points of iddata Objects

iddata Properties Storing Input and Output Data

The `InputData` `iddata` property stores column-wise input data, and the `OutputData` property stores column-wise output data. For more information about accessing `iddata` properties, see "iddata Properties" on page 2-58.

Horizontal Concatenation

Horizontal concatenation of `iddata` objects creates a new `iddata` object that appends all `InputData` information and all `OutputData`. This type of concatenation produces a single object with more input and output channels. For example, the following syntax performs horizontal concatenation on the `iddata` objects `data1,data2,...,dataN`:

```
data = [data1,data2,...,dataN]
```

This syntax is equivalent to the following longer syntax:

```
data.InputData =
     [data1.InputData,data2.InputData,...,dataN.InputData]
data.OutputData =
     [data1.OutputData,data2.OutputData,...,dataN.OutputData]
```

For horizontal concatenation, `data1,data2,...,dataN` must have the same number of samples and experiments , and the same`Ts` and `Tstart` values.

The channels in the concatenated `iddata` object are named according to the following rules:

- **Combining default channel names.** If you concatenate `iddata` objects with default channel names, such as `u1` and `y1`, channels in the new `iddata` object are automatically renamed to avoid name duplication.
- **Combining duplicate input channels.** If `data1,data2,...,dataN` have input channels with duplicate user-defined names, such that `dataK` contains channel names that are already present in `dataJ` with `J < K`, the `dataK` channels are ignored.
- **Combining duplicate output channels.** If `data1,data2,...,dataN` have input channels with duplicate user-defined names, only the output channels with unique names are added during the concatenation.

Vertical Concatenation

Vertical concatenation of `iddata` objects creates a new `iddata` object that vertically stacks the input and output data values in the corresponding data channels. The resulting object has the same number of channels, but each channel contains more data points. For example, the following syntax creates a `data` object such that its total number of samples is the sum of the samples in `data1,data2,...,dataN`.

```
data = [data1;data2;... ;dataN]
```

This syntax is equivalent to the following longer syntax:

```
data.InputData =
     [data1.InputData;data2.InputData;...;dataN.InputData]
data.OutputData =
     [data1.OutputData;data2.OutputData;...;dataN.OutputData]
```

For vertical concatenation, `data1,data2,...,dataN` must have the same number of input channels, output channels, and experiments.

Managing iddata Objects

Modifying Time and Frequency Vectors

The `iddata` object stores time-domain data or frequency-domain data and has several properties that specify the time or frequency values. To modify the time or frequency values, you must change the corresponding property values.

Note You can modify the property `SamplingInstants` by setting it to a new vector with the length equal to the number of data samples. For more information, see "Constructing an iddata Object for Time-Domain Data" on page 2-56.

The following tables summarize time-vector and frequency-vector properties, respectively, and provides usage examples. In each example, `data` is an `iddata` object.

Note Property names are not case sensitive. You do not need to type the entire property name if the first few letters uniquely identify the property.

---

iddata Time-Vector Properties

Property	Description	Syntax Example

`Ts`	Sampling time interval.
	• For a single experiment, `Ts` is a scalar value.
	• For multiexperiement data with `Ne` experiments, `Ts` is a `1`-by-`Ne` cell array, and each cell contains the sampling interval of the corresponding experiment.
		To set the sampling interval to `0.05`:
		`set(data,'ts',0.05)`
		or
		`data.ts = 0.05`

`Tstart`	Starting time of the experiment. • For a single experiment, `Ts` is a scalar value. • For multiexperiement data with `Ne` experiments, `Ts` is a `1`-by-`Ne` cell array, and each cell contains the sampling interval of the corresponding experiment.	To change starting time of the first data sample to `24`: `data.Tstart = 24` Time units are set by the property `TimeUnit`.

iddata Time-Vector Properties (Continued)

Property	Description	Syntax Example

`SamplingInstants`	Time values in the time vector, computed from the properties `Tstart` and `Ts`.
	• For a single experiment, `SamplingInstants` is an `N`-by-`1` vector.
	• For multiexperiement data with `Ne` experiments, this property is a `1`-by-`Ne` cell array, and each cell contains the sampling instants of the corresponding experiment.
		To retrieve the time vector for `iddata` object `data`, use:
		`get(data,'sa')`
		To plot the input data as a function of time:
		`plot(data.sa,data.u)`
		Note `sa` is the first two letters of the `SamplingInstants` property that uniquely identifies this property.

`TimeUnit`	Unit of time. Specify as one of the following: `'nanoseconds'`, `'microseconds'`, `'milliseconds'`, `'seconds'`, `'minutes'`, `'hours'`, `'days'`, `'weeks'`, `'months'`, and `'years'`.	To change the unit of the time vector to `milliseconds`: `data.ti = 'milliseconds'`

iddata Frequency-Vector Properties

Property	Description	Syntax Example
`Frequency`	Frequency values at which the Fourier transforms of the signals are defined. • For a single experiment, `Frequency` is a scalar value. • For 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)`
`FrequencyUnit`	Unit of Frequency. Specify as one of the following: be one of the following: `'rad/TimeUnit'`, `'cycles/TimeUnit'`, `'rad/s'`, `'Hz'`, `'kHz'`, `'MHz'`, `'GHz'`, and, `'rpm'`. Default: `` `rad/TimeUnit' `` For multi-experiement data with `Ne` experiments, `Units` is a `1-by-Ne` cell array, and each cell contains the frequency unit for each experiment.	Set the frequency unit to Hz: `data.FrequencyUnit = 'Hz'` Note that changing the frequency unit does not scale the frequency vector. For a proper translation of units, use `chgFreqUnit`.

Naming, Adding, and Removing Data Channels

What Are Input and Output Channels?. A multivariate system might contain several input variables or several output variables, or both. When an input or output signal includes several measured variables, these variables are called *channels*.

Naming Channels. The `iddata` properties `InputName` and `OutputName` store the channel names for the input and output signals. When you plot the data, you use channel names to select the variable displayed on the plot. If you have multivariate data, it is helpful to assign a name to each channel that describes the measured variable. For more information about selecting channels on a plot, see "Selecting Measured and Noise Channels in Plots" on page 12-16.

You can use the `set` command to specify the names of individual channels. For example, suppose `data` contains two input channels (voltage and current) and one output channel (temperature). To set these channel names, use the following syntax:

```
set(data,'InputName',{'Voltage','Current'},
         'OutputName','Temperature')
```

Tip You can also specify channel names as follows:

```
data.una = {'Voltage','Current')
data.yna = 'Temperature'
```

`una` is equivalent to the property `InputName`, and `yna` is equivalent to `OutputName`.

If you do not specify channel names when you create the `iddata` object, the toolbox assigns default names. By default, the output channels

are named 'y1','y2',...,'yn', and the input channels are named 'u1','u2',...,'un'.

Adding Channels. You can add data channels to an iddata object.

For example, consider an iddata object named data that contains an input signal with four channels. To add a fifth input channel, stored as the vector Input5, use the following syntax:

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

Input5 must have the same number of rows as the other input channels. In this example, data.u(:,5) references all samples as (indicated by :) of the input signal u and sets the values of the fifth channel. This channel is created when assigning its value to Input5.

You can also combine input channels and output channels of several iddata objects into one iddata object using concatenation. For more information, see "Increasing Number of Channels or Data Points of iddata Objects" on page 2-67.

Modifying Channel Data. After you create an iddata object, you can modify or remove specific input and output channels, if needed. You can accomplish this by subreferencing the input and output matrices and assigning new values.

For example, suppose the iddata object data contains three output channels (named y1, y2, and y3), and four input channels (named u1, u2, u3, and u4). To replace data such that it only contains samples in y3, u1, and u4, type the following at the prompt:

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

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

Subreferencing iddata Objects

See "Select Data Channels, I/O Data and Experiments in iddata Objects" on page 2-63.

Concatenating iddata Objects

See “Increasing Number of Channels or Data Points of iddata Objects” on page 2-67.

Representing Frequency-Response Data Using idfrd Objects

In this section...

idfrd Constructor

The `idfrd` represents complex frequency-response data. Before you can create an `idfrd` object, you must import your data as described in "Frequency-Response Data Representation" on page 2-13.

Note The `idfrd` object can only encapsulate one frequency-response data set. It does not support the `iddata` equivalent of multiexperiment data.

Use the following syntax to create the data object `fr_data`:

```
fr_data = idfrd(response,f,Ts)
```

Suppose that `ny` is the number of output channels, `nu` is the number of input channels, and `nf` is a vector of frequency values. `response` is an `ny`-by-`nu`-by-`nf` 3-D array. `f` is the frequency vector that contains the frequencies of the response.`Ts` is the sampling time, which is used when measuring or computing the frequency response. If you are working with a continuous-time system, set `Ts` to `0`.

`response(ky,ku,kf)`, where `ky`, `ku`, and `kf` reference the `k`th output, input, and frequency value, respectively, is interpreted as the complex-valued frequency response from input `ku` to output `ky` at frequency `f(kf)`.

Note When you work at the command line, you can only create `idfrd` objects from complex values of $G(e^{iw})$. For a SISO system, `response` can be a vector.

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

```
fr_data = idfrd(response,f,Ts,
                'Property1',Value1,...,'PropertyN',ValueN)
```

idfrd Properties

To view the properties of the `idfrd` object, you can use the `get` command. The following example shows how to create an `idfrd` object that contains 100 frequency-response values with a sampling time interval of 0.08 s and get its properties:

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

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

MATLAB returns the following object properties and values:

```
        ans =

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

For a complete description of all `idfrd` object properties, see the `idfrd` reference page.

To change property values for an existing `idfrd` object, use the `set` command or dot notation. For example, to change the name of the `idfrd` object, type the following command sequence at the prompt:

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

Property names are not case sensitive. You do not need to type the entire property name if the first few letters uniquely identify the property.

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

MATLAB returns the following object properties and values:

```
ans =

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

Select I/O Channels and Data in idfrd Objects

You can reference specific data values in the `idfrd` object using the following syntax:

```
fr_data(outputchannels,inputchannels)
```

Reference specific channels by name or by channel index.

Tip Use a colon (`:`) to specify all channels, and use the empty matrix (`[]`) to specify no channels.

For example, the following command references frequency-response data from input channel 3 to output channel 2:

```
fr_data(2,3)
```

You can also access the data in specific channels using channel names. To list multiple channel names, use a cell array. For example, to retrieve the power output, and the voltage and speed inputs, use the following syntax:

```
fr_data('power',{'voltage','speed'})
```

To retrieve only the responses corresponding to frequency values between 200 and 300, use the following command:

```
fr_data_sub = fselect(fr_data,[200:300])
```

You can also use logical expressions to subreference data. For example, to retrieve all frequency-response values between frequencies 1.27 and 9.3 in the idfrd object fr_data, use the following syntax:

```
fr_data_sub = fselect(fr_data,fr_data.f>1.27&fr_data.f<9.3)
```

Tip Use end to reference the last sample number in the data. For example, data(77:end).

Note You do not need to type the entire property name. In this example, f in fr_data.f uniquely identifies the Frequency property of the idfrd object.

Adding Input or Output Channels in idfrd Objects

About Concatenating idfrd Objects

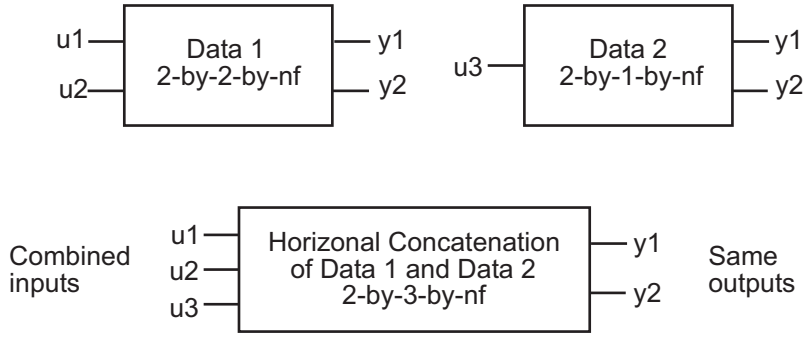
The horizontal and vertical concatenation of `idfrd` objects combine information in the `ResponseData` properties of these objects. `ResponseData` is an `ny`-by-`nu`-by-`nf` array that stores the response of the system, where `ny` is the number of output channels, `nu` is the number of input channels, and `nf` is a vector of frequency values (see "Properties").

Horizontal Concatenation of idfrd Objects

The following syntax creates a new `idfrd` object `data` that contains the horizontal concatenation of `data1,data2,...,dataN`:

```
data = [data1,data2,...,dataN]
```

`data` contains the frequency responses from all of the inputs in `data1,data2,...,dataN` to the same outputs. The following diagram is a graphical representation of horizontal concatenation of frequency-response data. The `(j,i,:)` vector of the resulting response data represents the frequency response from the `i`th input to the `j`th output at all frequencies.



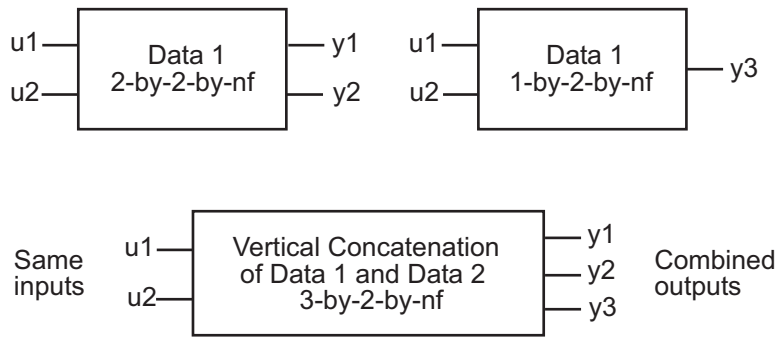
Note Horizontal concatenation of `idfrd` objects requires that they have the same outputs and frequency vectors. If the output channel names are different and their dimensions are the same, the concatenation operation resets the output names to their default values.

Vertical Concatenation of idfrd Objects

The following syntax creates a new `idfrd` object `data` that contains the vertical concatenation of `data1,data2,...,dataN`:

```
data = [data1;data2;... ;dataN]
```

The resulting `idfrd` object `data` contains the frequency responses from the same inputs in `data1,data2,...,dataN` to all the outputs. The following diagram is a graphical representation of vertical concatenation of frequency-response data. The `(j,i,:)` vector of the resulting response data represents the frequency response from the `i`th input to the `j`th output at all frequencies.



Note Vertical concatenation of `idfrd` objects requires that they have the same inputs and frequency vectors. If the input channel names are different and their dimensions are the same, the concatenation operation resets the input names to their default values.

Concatenating Noise Spectrum Data of idfrd Objects

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

In case of horizontal concatenation, there is no meaningful way to combine the `SpectrumData` of individual `idfrd` objects and the resulting `SpectrumData` property is empty. An empty property results because each `idfrd` object has its own set of noise channels, where the number of noise channels equals the

number of outputs. When the resulting `idfrd` object contains the same output channels as each of the individual `idfrd` objects, it cannot accommodate the noise data from all the `idfrd` objects.

In case of vertical concatenation, the toolbox concatenates individual noise models diagonally. The following shows that `data.SpectrumData` is a block diagonal matrix of the power spectra and cross spectra of the output noise in the system:

$$data.s = \begin{pmatrix} data1.s & & 0 \\ & \ddots & \\ 0 & & dataN.s \end{pmatrix}$$

`s` in `data.s` is the abbreviation for the `SpectrumData` property name.

Managing idfrd Objects

- "Subreferencing idfrd Objects" on page 2-83
- "Concatenating idfrd Objects" on page 2-83

Subreferencing idfrd Objects

See "Select I/O Channels and Data in idfrd Objects" on page 2-79.

Concatenating idfrd Objects

See "Adding Input or Output Channels in idfrd Objects" on page 2-80.

Operations That Create idfrd Objects

The following operations create `idfrd` objects:

- Constructing `idfrd` objects.
- Estimating nonparametric models using `etfe`, `spa`, and `spafdr`. For more information, see "Identifying Frequency-Response Models" on page 3-8.
- Converting the Control System Toolbox `frd` object. For more information, see "Using Identified Models for Control Design Applications" on page 10-2.

- Converting any linear dynamic system using the `idfrd` command.

  For example:

  ```
  sys_idpoly = idpoly([1 2 1],[0 2],'Ts',1);
  G = idfrd(sys_idpoly,linspace(0,pi,128))
  ```

Analyzing Data Quality

In this section...

Is Your Data Ready for Modeling?

Before you start estimating models from data, you should check your data for the presence of any undesirable characteristics. For example, you might plot the data to identify drifts and outliers. You plot analysis might lead you to preprocess your data before model estimation.

The following data plots are available in the toolbox:

- Time plot — Shows data values as a function of time.

  Tip You can infer time delays from time plots, which are required inputs to most parametric models. A *time delay* is the time interval between the change in input and the corresponding change in output.

- Spectral plot — Shows a *periodogram* that is computed by taking the absolute squares of the Fourier transforms of the data, dividing by the number of data points, and multiplying by the sampling interval.
- Frequency-response plot — For frequency-response data, shows the amplitude and phase of the frequency-response function on a Bode plot. For time- and frequency-domain data, shows the empirical transfer function estimate (see `etfe`) .

See Also

"How to Analyze Data Using the advice Command" on page 2-94

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

Plotting Data in the GUI Versus at the Command Line

The plots you create using the System Identification Tool GUI provide options that are specific to the System Identification Toolbox product, such as selecting specific channel pairs in a multivariate signals or converting frequency units between Hertz and radians per second. For more information, see "How to Plot Data in the GUI" on page 2-86.

The plots you create using the plot commands, such as `plot`, and `bode` are displayed in the standard MATLAB Figure window, which provides options for formatting, saving, printing, and exporting plots to a variety of file formats. To learn about plotting at the command line, see "How to Plot Data at the Command Line" on page 2-92. For more information about working with Figure window, see "Figure Windows".

How to Plot Data in the GUI

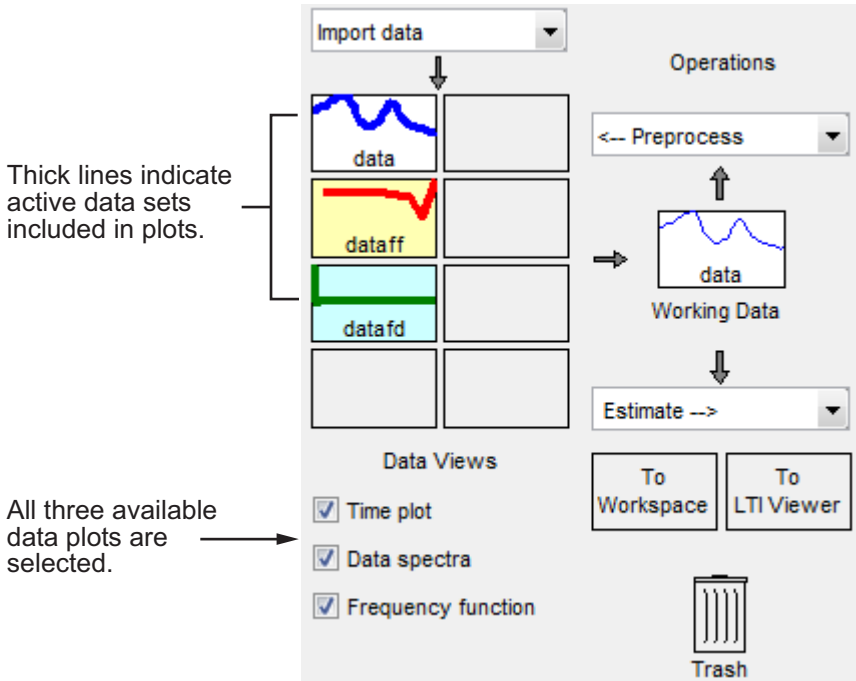
How to Plot Data in the GUI

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

To create one or more plots, select the corresponding check box in the **Data Views** area of the System Identification Tool GUI.

An *active* data icon has a thick line in the icon, while an *inactive* data set has a thin line. Only active data sets appear on the selected plots. To toggle including and excluding data on a plot, click the corresponding icon in the System Identification Tool GUI. Clicking the data icon updates any plots that are currently open.

When you have several data sets, you can view different input-output channel pair by selecting that pair from the **Channel** menu. For more information about selecting different input and output pairs, see "Selecting Measured and Noise Channels in Plots" on page 12-16.



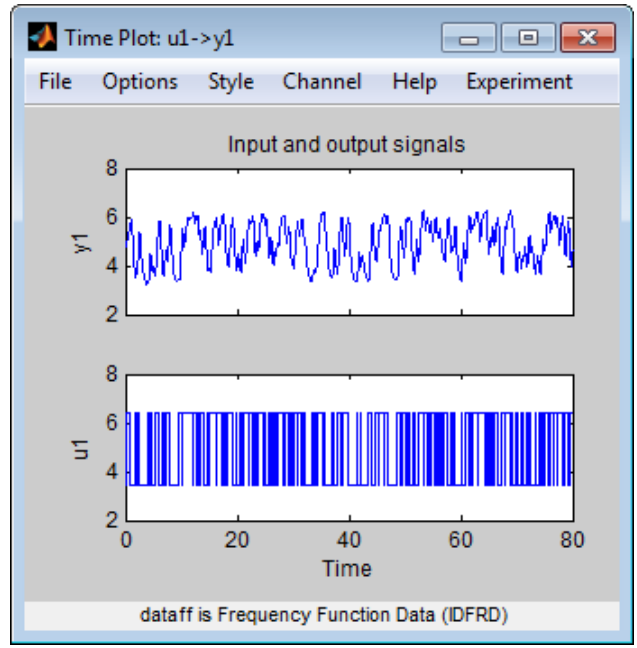
In this example, `data` and `dataff` are active and appear on the three selected plots.

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

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

Manipulating a Time Plot

The **Time plot** only shows time-domain data. In this example, `data1` is displayed on the time plot because, of the three data sets, it is the only one that contains time-domain input and output.



Time Plot of data1

The following table summarizes options that are specific to time plots, which you can select from the plot window menus. For general information about working with System Identification Toolbox plots, see “Working with Plots” on page 12-13.

Time Plot Options

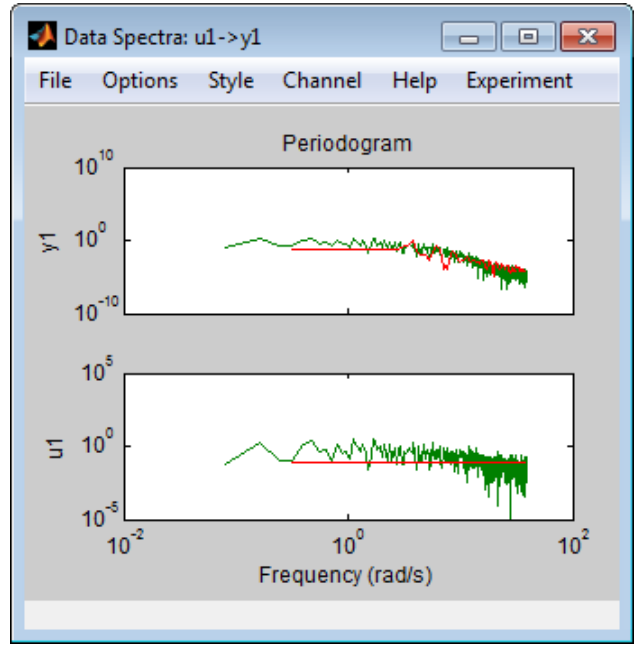
Action	Command
Toggle input display between piece-wise continuous (zero-order hold) and linear interpolation (first-order hold) between samples. **Note** This option only affects the display and not the intersample behavior specified when importing the data.	Select **Style > Staircase input** for zero-order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

Manipulating Data Spectra Plot

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

The periodogram is computed by taking the absolute squares of the Fourier transforms of the data, dividing by the number of data points, and multiplying by the sampling interval. The spectral estimate for time-domain data is a smoothed spectrum calculated using `spa`. For frequency-domain data, the **Data spectra** plot shows the square of the absolute value of the actual data, normalized by the sampling interval.

The top axes show the input and the bottom axes show the output. The vertical axis of each plot is labeled with the corresponding channel name.



Periodograms of data1 and data3fd

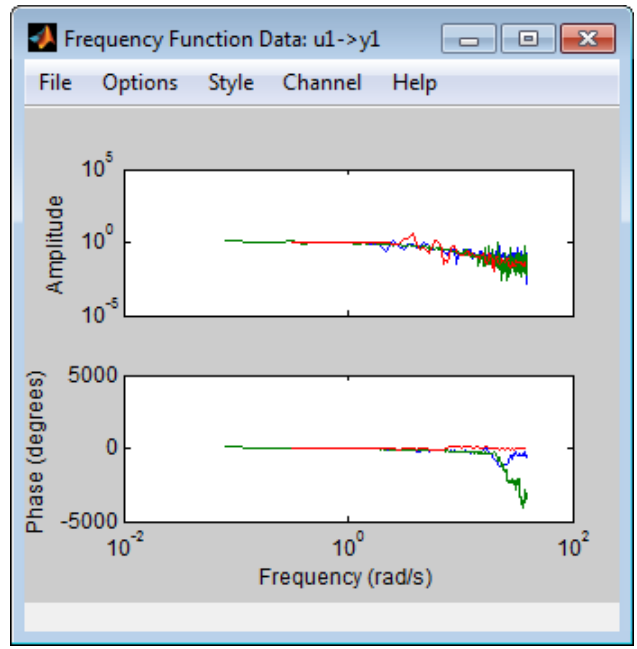
Data Spectra Plot Options

Action	Command
Toggle display between periodogram and spectral estimate.	Select **Options > Periodogram** or **Options > Spectral analysis**.
Change frequency units.	Select **Style > Frequency (rad/s)** or **Style > Frequency (Hz)**.
Toggle frequency scale between linear and logarithmic.	Select **Style > Linear frequency scale** or **Style > Log frequency scale**.
Toggle amplitude scale between linear and logarithmic.	Select **Style > Linear amplitude scale** or **Style > Log amplitude scale**.

Manipulating a Frequency Function Plot

For time-domain data, the **Frequency function** plot shows the empirical transfer function estimate (`etfe`). For frequency-domain data, the plot shows the ratio of output to input data.

The frequency-response plot shows the amplitude and phase plots of the corresponding frequency response. For more information about frequency-response data, see "Frequency-Response Data Representation" on page 2-13.



Frequency Functions of data1 and data3fd

Frequency Function Plot Options

Action	Command
Change frequency units.	Select **Style > Frequency (rad/s)** or **Style > Frequency (Hz)**.
Toggle frequency scale between linear and logarithmic.	Select **Style > Linear frequency scale** or **Style > Log frequency scale**.
Toggle amplitude scale between linear and logarithmic.	Select **Style > Linear amplitude scale** or **Style > Log amplitude scale**.

How to Plot Data at the Command Line

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

Commands for Plotting Data

Command	Description	Example

`bode`, `bodeplot`	For frequency-response data only. Shows the magnitude and phase of the frequency response on a logarithmic frequency scale of a Bode plot.	To plot `idfrd` data:
		`bode(idfrd_data)`
		or:
		`bodeplot(idfrd_data)`

`plot`	The type of plot corresponds to the type of data. For example, plotting time-domain data generates a time plot, and plotting frequency-response data generates a frequency-response plot. When plotting time- or frequency-domain inputs	To plot `iddata` or `idfrd` data: `plot(data)`

Commands for Plotting Data (Continued)

Command	Description	Example
	and outputs, the top axes show the output and the bottom axes show the input.	

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

To plot portions of the data, you can subreference specific samples (see "Select Data Channels, I/O Data and Experiments in iddata Objects" on page 2-63 and "Select I/O Channels and Data in idfrd Objects" on page 2-79. For example:

```
plot(data(1:300))
```

For time-domain data, to plot only the input data as a function of time, use the following syntax:

```
plot(data(:,[],:)
```

When `data.intersample = 'zoh'`, the input is piece-wise constant between sampling points on the plot. For more information about properties, see the `iddata` reference page.

You can generate plots of the input data in the time domain using:

```
plot(data.SamplingInstants,data.u)
```

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

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

When you specify to plot a multivariable `iddata` object, each input-output combination is displayed one at a time in the same MATLAB Figure window. You must press **Enter** to update the Figure window and view the next channel combination. To cancel the plotting operation, press **Ctrl+C**.

Tip To plot specific input and output channels, use `plot(data(:,ky,ku))`, where `ky` and `ku` are specific output and input channel indexes or names. For more information about subreferencing channels, see "Subreferencing Data Channels" on page 2-65.

To plot several `iddata` sets `d1,...,dN`, use `plot(d1,...,dN)`. Input-output channels with the same experiment name, input name, and output name are always plotted in the same plot.

How to Analyze Data Using the advice Command

You can use the `advice` command to analyze time- or frequency- domain data before estimating a model. The resulting report informs you about the possible need to preprocess the data and identifies potential restrictions on the model accuracy. You should use these recommendations in combination with plotting the data and validating the models estimated from this data.

Note `advice` does not support frequency-response data.

Before applying the `advice` command to your data, you must have represented your data as an `iddata` object. For more information, see "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-55.

If you are using the System Identification Tool GUI, you must export your data to the MATLAB workspace before you can use the `advice` command on this data. For more information about exporting data, see "Exporting Models from the GUI to the MATLAB Workspace" on page 12-12.

Use the following syntax to get advice about an `iddata` object `data`:

```
advice(data)
```

For more information about the `advice` syntax, see the `advice` reference page.

Advice provide guidance for these kinds of questions:

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

See Also

`advice`

`delayest`

`detrend`

`feedback`

`pexcit`

Selecting Subsets of Data

In this section...

Why Select Subsets of Data?

You can use data selection to create independent data sets for estimation and validation.

You can also use data selection as a way to clean the data and exclude parts with noisy or missing information. For example, when your data contains missing values, outliers, level changes, and disturbances, you can select one or more portions of the data that are suitable for identification and exclude the rest.

If you only have one data set and you want to estimate linear models, you should split the data into two portions to create two independent data sets for estimation and validation, respectively. Splitting the data is selecting parts of the data set and saving each part independently.

You can merge several data segments into a single multiexperiment data set and identify an average model. For more information, see "Importing Data into the GUI" on page 2-17 or "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-55.

Note Subsets of the data set must contain enough samples to adequately represent the system, and the inputs must provide suitable excitation to the system.

Selecting potions of frequency-domain data is equivalent to filtering the data. For more information about filtering, see "Filtering Data" on page 2-120.

Extract Subsets of Data Using the GUI

Ways to Select Data in the GUI

You can use System Identification Tool GUI to select ranges of data on a time-domain or frequency-domain plot. Selecting data in the frequency domain is equivalent to passband-filtering the data.

After you select portions of the data, you can specify to use one data segment for estimating models and use the other data segment for validating models. For more information, see "Specifying Estimation and Validation Data" on page 2-35.

Note Selecting **<--Preprocess > Quick start** performs the following actions simultaneously:

- Remove the mean value from each channel.
- Split the data into two parts.
- Specify the first part as estimation data (or **Working Data**).
- Specify the second part as **Validation Data**.

Selecting a Range for Time-Domain Data

You can select a range of data values on a time plot and save it as a new data set in the System Identification Tool GUI.

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

To extract a subset of time-domain data and save it as a new data set:

1. Import time-domain data into the System Identification Tool GUI, as described in "Importing Data into the GUI" on page 2-17.

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

3. If your data contains multiple I/O channels, in the **Channel** menu, select the channel pair you want to view. The upper plot corresponds to the input signal, and the lower plot corresponds to the output signal.

   Although you view only one I/O channel pair at a time, your data selection is applied to all channels in this data set.

4. Select the data of interest in either of the following ways:

   - Graphically — Draw a rectangle on either the input-signal or the output-signal plot with the mouse to select the desired time interval. Your selection appears on both plots regardless of the plot on which you draw the rectangle. The **Time span** and **Samples** fields are updated to match the selected region.
   - By specifying the **Time span** — Edit the beginning and the end times in seconds. The **Samples** field is updated to match the selected region. For example:

     ```
     28.5 56.8
     ```

   - By specifying the **Samples** range — Edit the beginning and the end indices of the sample range. The **Time span** field is updated to match the selected region. For example:

     ```
     342 654
     ```

   Note To clear your selection, click **Revert**.

5. In the **Data name** field, enter the name of the data set containing the selected data.

6. Click **Insert**. This action saves the selection as a new data set and adds it to the Data Board.

7 To select another range, repeat steps 4 to 6.

Selecting a Range of Frequency-Domain Data

Selecting a range of values in frequency domain is equivalent to filtering the data. For more information about data filtering, see "Filtering Frequency-Domain or Frequency-Response Data in the GUI" on page 2-123.

Extract Subsets of Data at the Command Line

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

For more information about subreferencing time-domain and frequency-domain data, see "Select Data Channels, I/O Data and Experiments in iddata Objects" on page 2-63.

For more information about subreferencing frequency-response data, see "Select I/O Channels and Data in idfrd Objects" on page 2-79.

Handling Missing Data and Outliers

In this section...

Handling Missing Data

Data acquisition failures sometimes result in missing measurements both in the input and the output signals. When you import data that contains missing values using the MATLAB Import Wizard, these values are automatically set to `NaN`. `NaN` serves as a flag for nonexistent or undefined data. When you plot data on a time-plot that contains missing values, gaps appear on the plot where missing data exists.

You can use `misdata` to estimate missing values. This command linearly interpolates missing values to estimate the first model. Then, it uses this model to estimate the missing data as parameters by minimizing the output prediction errors obtained from the reconstructed data. You can specify the model structure you want to use in the `misdata` argument or estimate a default-order model using the `n4sid` method. For more information, see the `misdata` reference page.

Note You can only use `misdata` on time-domain data stored in an `iddata` object. For more information about creating `iddata` objects, see "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-55.

For example, suppose `y` and `u` are output and input signals that contain `NaN`s. This data is sampled at `0.2` s. The following syntax creates a new `iddata` object with these input and output signals.

```
dat = iddata(y,u,0.2) % y and u contain NaNs
                      % representing missing data
```

Apply the `misdata` command to the new data object. For example:

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

Handling Outliers

Malfunctions can produce errors in measured values, called *outliers*. Such outliers might be caused by signal spikes or by measurement malfunctions. If you do not remove outliers from your data, this can adversely affect the estimated models.

To identify the presence of outliers, perform one of the following tasks:

- Before estimating a model, plot the data on a time plot and identify values that appear out of range.
- After estimating a model, plot the residuals and identify unusually large values. For more information about plotting residuals, see "Residual Analysis" on page 8-24. 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 "Extract and Model Specific Data Segments" on page 2-102). For more information about selecting and extracting data segments, see "Selecting Subsets of Data" on page 2-96.

Tip The inputs in each of the data segments must be consistently exciting the system. Splitting data into meaningful segments for steady-state data results in minimum information loss. Avoid making data segments too small.

- Manually replace outliers with `NaN`s and then use the `misdata` command to reconstruct flagged data. This approach treats outliers as missing data and is described in "Handling Missing Data" on page 2-100. Use this method when your data contains several inputs and outputs, and when you have difficulty finding reliable data segments in all variables.
- Remove outliers by prefiltering the data for high-frequency content because outliers often result from abrupt changes. For more information about filtering, see "Filtering Data" on page 2-120.

Note The estimation algorithm can handle outliers by assigning a smaller weight to outlier data. A robust error criterion applies an error penalty that is quadratic for small and moderate prediction errors, and is linear for large prediction errors. Because outliers produce large prediction errors, this approach gives a smaller weight to the corresponding data points during model estimation. Set the `ErrorThreshold` estimation option (see `Advanced.ErrorThreshold` in, for example, `polyestOptions`) to a nonzero value to activate the correction for outliers in the estimation algorithm.

Extract and Model Specific Data Segments

This example shows how to create a multi-experiment, time-domain data set by merging only the accurate data segments and ignoring the rest.

Assume that the data has poor or no measurements for some sample ranges (for example 341–499). You cannot simply concatenate the good data segments because the transients at the connection points compromise the model. Instead, you must create a multiexperiment `iddata` object, where each experiment corresponds to a good segment of data, as follows:

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

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

% Model the multiexperiment data set
% using "experiments" 1, 2, and 4
m = n4sid(getexp(datam,[1,2,4]))

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

See Also

To learn more about the theory of handling missing data and outliers, see the chapter on preprocessing data in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

Handling Offsets and Trends in Data

In this section...

When to Detrend Data

Detrending is removing means, offsets, or linear trends from regularly sampled time-domain input-output data signals. This data processing operation helps you estimate more accurate linear models because linear models cannot capture arbitrary differences between the input and output signal levels. The linear models you estimate from detrended data describe the relationship between the change in input signals and the change in output signals.

For steady-state data, you should remove mean values and linear trends from both input and output signals.

For transient data, you should remove physical-equilibrium offsets measured prior to the excitation input signal.

Remove one linear trend or several piecewise linear trends when the levels drift during the experiment. Signal drift is considered a low-frequency disturbance and can result in unstable models.

You should not detrend data before model estimation when you want:

- Linear models that capture offsets essential for describing important system dynamics. For example, when a model contains integration behavior, you could estimate a low-order transfer function (process model) from nondetrended data. For more information, see "Identifying Process Models" on page 3-26.

- Nonlinear black-box models, such as nonlinear ARX or Hammerstein-Wiener models. For more information, see "Nonlinear Model Identification".

Tip When signals vary around a large signal level, you can improve computational accuracy of nonlinear models by detrending the signal means.

- Nonlinear ODE parameters (nonlinear grey-box models). For more information, see "Estimating Nonlinear Grey-Box Models" on page 5-17.

To simulate or predict the linear model response at the system operating conditions, you can restore the removed trend to the simulated or predicted model output using the `retrend` command.

For more information about handling drifts in the data, see the chapter on preprocessing data in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

Examples

"How to Detrend Data Using the GUI" on page 2-108

"How to Detrend Data at the Command Line" on page 2-109

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

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

Both the GUI and the command line let you subtract the mean values and one linear trend from steady-state time-domain signals.

However, the `detrend` command provides the following additional functionality (not available in the GUI):

- Subtracting piecewise linear trends at specified breakpoints. A *breakpoint* is a time value that defines the discontinuities between successive linear trends.
- Subtracting arbitrary offsets and linear trends from transient data signals.
- Saving trend information to a variable so that you can apply it to multiple data sets.

To learn how to detrend data, see:

- "How to Detrend Data Using the GUI" on page 2-108
- "How to Detrend Data at the Command Line" on page 2-109

As an alternative to detrending data beforehand, you can specify the offsets levels as estimation options and use them directly with the estimation command.

For example, suppose your data has an input offset, *u0*, and an output offset, *y0*. There are two ways to perform a linear model estimation (say, a transfer function model estimation) using this data:

- Using detrend:

```
T=getTrend(data)
T.InputOffset = u0;
T.OutputOffset = y0;
datad = detrend(data, T);

model = tfest(datad, np);
```

- Specify offsets as estimation options:

```
opt = tfestOptions('InputOffset',u0, 'OutputOffset', y0);

model = tfest(data, np, opt)
```

  The advantage of this approach is that there is a record of offset levels in the model in `model.Report.OptionsUsed`. The limitation of this approach is that it cannot handle linear trends, which can only be removed from the data by using `detrend`.

Next Steps After Detrending

After detrending your data, you might do the following:

- Perform other data preprocessing operations. See “Ways to Prepare Data for System Identification” on page 2-6.
- Estimate a linear model. See “Linear Model Identification”.

How to Detrend Data Using the GUI

Before you can perform this task, you must have regularly-sampled, steady-state time-domain data imported into the System Identification Tool GUI. See “Importing Time-Domain Data into the GUI” on page 2-18). For transient data, see “How to Detrend Data at the Command Line” on page 2-109.

Tip You can use the shortcut **Preprocess > Quick start** to perform several operations: remove the mean value from each signal, split data into two halves, specify the first half as model estimation data (or **Working Data**), and specify the second half as model **Validation Data**.

---

1. In the System Identification Tool, drag the data set you want to detrend to the **Working Data** rectangle.
2. Detrend the data.
   - To remove linear trends, select **Preprocess > Remove trends**.
   - To remove mean values from each input and output data signal, select **Preprocess > Remove means**.

More About

“Handling Offsets and Trends in Data” on page 2-104

How to Detrend Data at the Command Line

In this section...

Detrending Steady-State Data

Before you can perform this task, you must have time-domain data as an `iddata` object. See "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-55.

Note If you plan to estimate models from this data, your data must be regularly sampled.

Use the `detrend` command to remove the signal means or linear trends:

```
[data_d,T]=detrend(data,Type)
```

where `data` is the data to be detrended. The second input argument `Type=0` removes signal means or `Type=1` removes linear trends. `data_d` is the detrended data. `T` is a `TrendInfo` object that stores the values of the subtracted offsets and slopes of the removed trends.

More About

"Handling Offsets and Trends in Data" on page 2-104

Detrending Transient Data

Before you can perform this task, you must have

- Time-domain data as an `iddata` object. See "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-55.

Note If you plan to estimate models from this data, your data must be regularly sampled.

- Values of the offsets you want to remove from the input and output data. If you do not know these values, visually inspect a time plot of your data. For more information, see "How to Plot Data at the Command Line" on page 2-92.

1. Create a default object for storing input-output offsets that you want to remove from the data.

   ```
   T = getTrend(data)
   ```

   where `T` is a `TrendInfo` object.

2. Assign offset values to `T`.

   ```
   T.InputOffset=I_value;
   T.OutputOffset=O_value;
   ```

   where `I_value` is the input offset value, and `O_value` is the input offset value.

3. Remove the specified offsets from `data`.

   ```
   data_d = detrend(data,T)
   ```

   where the second input argument `T` stores the offset values as its properties.

More About

"Handling Offsets and Trends in Data" on page 2-104

See Also

`detrend`

`TrendInfo`

Resampling Data

In this section...

What Is Resampling?

Resampling data signals in the System Identification Toolbox product applies an antialiasing (lowpass) FIR filter to the data and changes the sampling rate of the signal by decimation or interpolation.

If your data is sampled faster than needed during the experiment, you can decimate it without information loss. If your data is sampled more slowly than needed, there is a possibility that you miss important information about the dynamics at higher frequencies. Although you can resample the data at a higher rate, the resampled values occurring between measured samples do not represent new measured information about your system. Instead of resampling, repeat the experiment using a higher sampling rate.

Tip You should decimate your data when it contains high-frequency noise outside the frequency range of the system dynamics.

Resampling takes into account how the data behaves between samples, which you specify when you import the data into the System Identification Tool GUI (zero-order or first-order hold). For more information about the data properties you specify before importing the data, see "Importing Data into the GUI" on page 2-17.

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

Examples

"Resampling Data Using the GUI" on page 2-117

"Resampling Data at the Command Line" on page 2-118

Resampling Data Without Aliasing Effects

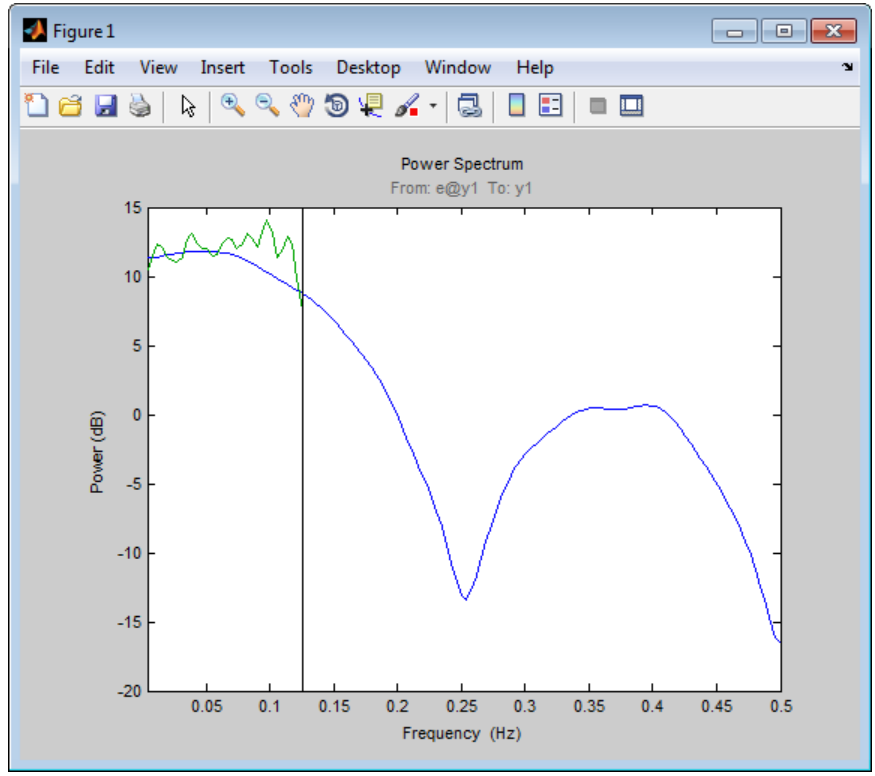
Typically, you decimate a signal to remove the high-frequency contributions that result from noise from the total energy. Ideally, you want to remove the energy contribution due to noise and preserve the energy density of the signal.

The command `resample` performs the decimation without aliasing effects. This command includes a factor of T to normalize the spectrum and preserve the energy density after decimation. For more information about spectrum normalization, see "Spectrum Normalization" on page 3-14.

If you use manual decimation instead of `resample`—by picking every fourth sample from the signal, for example—the energy contributions from higher frequencies are folded back into the lower frequencies("aliasing"). Because the total signal energy is preserved by this operation and this energy must now be squeezed into a smaller frequency range, the amplitude of the spectrum at each frequency increases. Thus, the energy density of the decimated signal is not constant.

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

```
% Construct fourth-order MA-process
m0 = idpoly(1,[ ],[1 1 1 1]); % a time series (no input) model

% Generate error signal
e = idinput(2000,'rgs');

% Simulate the output using error signal
sim_opt = simOptions('AddNoise',true,'NoiseData',e);
y = sim(m0,zeros(2000,0),sim_opt)
y = iddata(y,[],1)

% 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
h = spectrumplot(g1,g2,g1.Frequency);
opt = getoptions(h);
opt.FreqScale='linear';
opt.FreqUnits='Hz';
setoptions(h,opt)

% Estimate spectrum after prefiltering that does not
% introduce folding effects
g3 = spa(resample(y,1,4));
figure
spectrumplot(g1,g3,g1.Frequency,opt)
```

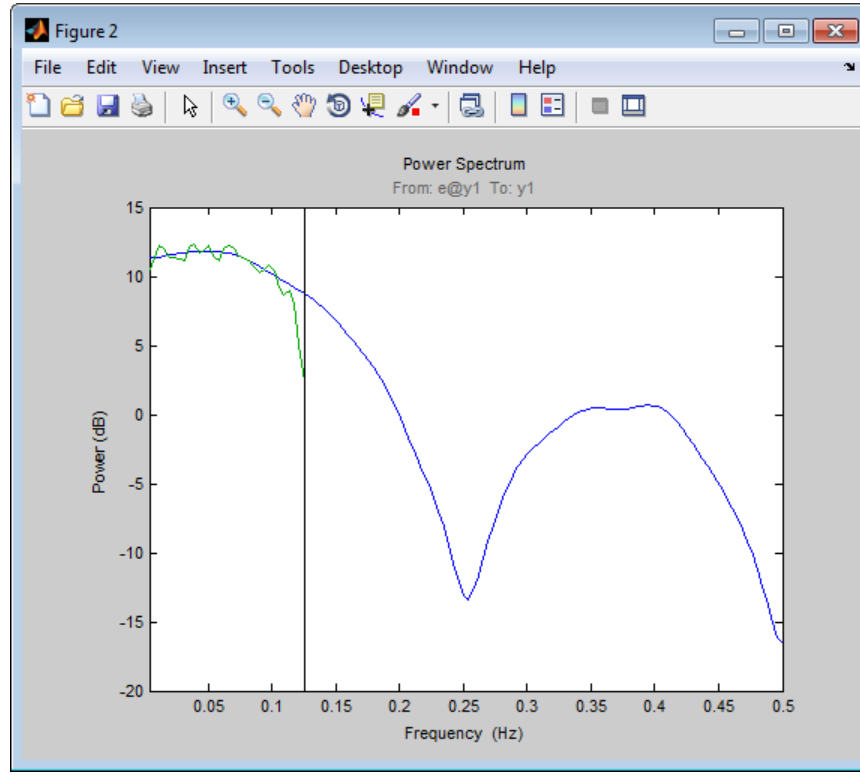


Folding Effects with Manual Decimation

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

```
g3 = spa(resample(y,1,4));
figure
spectrumplot(g1,g3,g1.Frequency,opt)
```

The following figure shows that the estimated spectrum of the resampled signal has the same amplitude as the original spectrum. Thus, there is no indication of folding effects when you use `resample` to eliminate aliasing.



No Folding Effects When Using resample

Examples

“Resampling Data Using the GUI” on page 2-117

"Resampling Data at the Command Line" on page 2-118

See Also

For a detailed discussion about handling disturbances, see the chapter on preprocessing data in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

Resampling Data Using the GUI

Use the System Identification Tool GUI to resample time-domain data. To specify additional options, such as the prefilter order, see “Resampling Data at the Command Line” on page 2-118.

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

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

1. Import time-domain data into the System Identification Tool GUI, as described in “Importing Data into the GUI” on page 2-17.
2. Drag the data set you want to resample to the **Working Data** area.
3. In the **Resampling factor** field, enter the factor by which to multiply the current sampling interval:
   - For decimation (fewer samples), enter a factor greater than `1` to increase the sampling interval by this factor.
   - For interpolation (more samples), enter a factor less than `1` to decrease the sampling interval by this factor.

   Default = `1`.
4. In the **Data name** field, type the name of the new data set. Choose a name that is unique in the Data Board.
5. Click **Insert** to add the new data set to the Data Board in the System Identification Toolbox window.
6. Click **Close** to close the Resample dialog box.

More About

“Resampling Data” on page 2-111

Resampling Data at the Command Line

Use `resample` to decimate and interpolate time-domain `iddata` objects. You can specify the order of the antialiasing filter as an argument.

Note `resample` uses the Signal Processing Toolbox™ command, when this toolbox is installed on your computer. If this toolbox is not installed, use `idresamp` instead. `idresamp` only lets you specify the filter order, whereas `resample` also lets you specify filter coefficients and the design parameters of the Kaiser window.

To create a new `iddata` object `datar` by resampling `data`, use the following syntax:

```
datar = resample(data,P,Q,filter_order)
```

In this case, `P` and `Q` are integers that specify the new sampling interval: the new sampling interval is `Q/P` times the original one. You can also specify the order of the resampling filter as a fourth argument `filter_order`, which is an integer (default is `10`). For detailed information about `resample`, see the corresponding reference page.

For example, `resample(data,1,Q)` results in decimation with the sampling interval modified by a factor `Q`.

The next example shows how you can increase the sampling rate by a factor of 1.5 and compare the signals:

```
plot(u)
ur = resample(u,3,2);
plot(u,ur)
```

When the Signal Processing Toolbox product is not installed, using `resample` calls `idresamp` instead.

`idresamp` uses the following syntax:

```
datar = idresamp(data,R,filter_order)
```

In this case, `R=Q/P`, which means that data is interpolated by a factor `P` and then decimated by a factor `Q`. To learn more about `idresamp`, type `help idresamp`.

The `data.InterSample` property of the `iddata` object is taken into account during resampling (for example, first-order hold or zero-order hold). For more information, see "iddata Properties" on page 2-58.

More About

"Resampling Data" on page 2-111

Filtering Data

In this section...

Supported Filters

You can filter the input and output signals through a linear filter before estimating a model in the System Identification Tool GUI or at the command line. How you want to handle the noise in the system determines whether it is appropriate to prefilter the data.

The filter available in the System Identification Tool GUI is a fifth-order (passband) Butterworth filter. If you need to specify a custom filter, use the `idfilt` command.

Examples

"How to Filter Data Using the GUI" on page 2-122

"How to Filter Data at the Command Line" on page 2-126

Choosing to Prefilter Your Data

Prefiltering data can help remove high-frequency noise or low-frequency disturbances (drift). The latter application is an alternative to subtracting linear trends from the data, as described in "Handling Offsets and Trends in Data" on page 2-104.

In addition to minimizing noise, prefiltering lets you focus your model on specific frequency bands. The frequency range of interest often corresponds to a passband over the breakpoints on a Bode plot. For example, if you are modeling a plant for control-design applications, you might prefilter the data to specifically enhance frequencies around the desired closed-loop bandwidth.

Prefiltering the input and output data through the same filter does not change the input-output relationship for a linear system. However, prefiltering does change the noise characteristics and affects the estimated model of the system.

To get a reliable noise model, avoid prefiltering the data. Instead, set the `Focus` property of the estimation algorithm to `Simulation`.

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

Examples

"How to Filter Data Using the GUI" on page 2-122

"How to Filter Data at the Command Line" on page 2-126

See Also

To learn how to filter data during linear model estimation instead, you can set the `Focus` property of the estimation algorithm to `Filter` and specify the filter characteristics.

For more information about prefiltering data, see the chapter on preprocessing data in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

For practical examples of prefiltering data, see the section on posttreatment of data in *Modeling of Dynamic Systems*, by Lennart Ljung and Torkel Glad, Prentice Hall PTR, 1994.

How to Filter Data Using the GUI

In this section...

Filtering Time-Domain Data in the GUI

The System Identification Tool GUI lets you filter time-domain data using a fifth-order Butterworth filter by enhancing or selecting specific passbands.

To create a filtered data set:

1. Import time-domain data into the System Identification Tool GUI, as described in "Importing Data into the GUI" on page 2-17.
2. Drag the data set you want to filter to the **Working Data** area.
3. Select **<--Preprocess > Filter**. By default, this selection shows a periodogram of the input and output spectra (see the `etfe` reference page).

   Note To display smoothed spectral estimates instead of the periodogram, select **Options > Spectral analysis**. This spectral estimate is computed using `spa` and your previous settings in the Spectral Model dialog box. To change these settings, select **<--Estimate > Spectral model** in the System Identification Tool GUI, and specify new model settings.

4. If your data contains multiple input/output channels, in the **Channel** menu, select the channel pair you want to view. Although you view only one channel pair at a time, the filter applies to all input/output channels in this data set.
5. Select the data of interest using one of the following ways:
   - Graphically — Draw a rectangle with the mouse on either the input-signal or the output-signal plot to select the desired frequency

interval. Your selection is displayed on both plots regardless of the plot on which you draw the rectangle. The **Range** field is updated to match the selected region. If you need to clear your selection, right-click the plot.

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

  For example:

  `8.5 20.0` (rad/s).

Tip To change the frequency units from `rad/s` to `Hz`, select **Style > Frequency (Hz)**. To change the frequency units from `Hz` to `rad/s`, select **Style > Frequency (rad/s)**.

---

6. In the **Range is** list, select one of the following:
   - `Pass band` — Allows data in the selected frequency range.
   - `Stop band` — Excludes data in the selected frequency range.
7. Click **Filter** to preview the filtered results. If you are satisfied, go to step 8. Otherwise, return to step 5.
8. In the **Data name** field, enter the name of the data set containing the selected data.
9. Click **Insert** to save the selection as a new data set and add it to the Data Board.
10. To select another range, repeat steps 5 to 9.

More About

"Filtering Data" on page 2-120

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

For frequency-domain and frequency-response data, *filtering* is equivalent to selecting specific data ranges.

To select a range of data in frequency-domain or frequency-response data:

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

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

3. Select **<--Preprocess > Select range**. This selection displays one of the following plots:

   - Frequency-domain data — Plot shows the absolute of the squares of the input and output spectra.
   - Frequency-response data — Top axes show the frequency response magnitude equivalent to the ratio of the output to the input, and the bottom axes show the ratio of the input signal to itself, which has the value of `1` at all frequencies.

4. If your data contains multiple input/output channels, in the **Channel** menu, select the channel pair you want to view. Although you view only one channel pair at a time, the filter applies to all input/output channels in this data set.

5. Select the data of interest using one of the following ways:

   - Graphically — Draw a rectangle with the mouse on either the input-signal or the output-signal plot to select the desired frequency interval. Your selection is displayed on both plots regardless of the plot on which you draw the rectangle. The **Range** field is updated to match the selected region.

     If you need to clear your selection, right-click the plot.

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

     For example:

     `8.5 20.0` (rad/s).

Tip If you need to change the frequency units from `rad/s` to `Hz`, select **Style > Frequency (Hz)**. To change the frequency units from `Hz` to `rad/s`, select **Style > Frequency (rad/s)**.

---

6 In the **Range is** list, select one of the following:

- `Pass band` — Allows data in the selected frequency range.
- `Stop band` — Excludes data in the selected frequency range.

7 In the **Data name** field, enter the name of the data set containing the selected data.

8 Click **Insert**. This action saves the selection as a new data set and adds it to the Data Board.

9 To select another range, repeat steps 5 to 8.

More About

"Filtering Data" on page 2-120

How to Filter Data at the Command Line

In this section...

Simple Passband Filter

Use `idfilt` to apply passband and other custom filters to a time-domain or a frequency-domain `iddata` object.

In general, you can specify any custom filter. Use this syntax to filter an `iddata` object `data` using the filter called `filter`:

```
fdata = idfilt(data,filter)
```

In the simplest case, you can specify a passband filter for time-domain data using the following syntax:

```
fdata = idfilt(data,[wl wh])
```

In this case, `w1` and `wh` represent the low and high frequencies of the passband, respectively.

You can specify several passbands, as follows:

```
filter=[[w1l,w1h];[ w2l,w2h]; ....;[wnl,wnh]]
```

The filter is an `n`-by-`2` matrix, where each row defines a passband in radians per second.

To define a stopband between `ws1` and `ws2`, use

```
filter = [0 ws1; ws2 Nyqf]
```

where, `Nyqf` is the Nyquist frequency.

For time-domain data, the passband filtering is cascaded Butterworth filters of specified order. The default filter order is 5. The Butterworth filter is the same as `butter` in the Signal Processing Toolbox product. For frequency-domain data, select the indicated portions of the data to perform passband filtering.

More About

"Filtering Data" on page 2-120

Defining a Custom Filter

Use `idfilt` to apply passband and other custom filters to a time-domain or a frequency-domain `iddata` object.

In general, you can specify any custom filter. Use this syntax to filter an `iddata` object `data` using the filter called `filter`:

```
fdata = idfilt(data,filter)
```

You can define a general single-input/single-output (SISO) system for filtering time-domain or frequency-domain data. For frequency-domain only, you can specify the (nonparametric) frequency response of the filter.

You use this syntax to filter an `iddata` object `data` using a custom filter specified by `filter`:

```
fdata = idfilt(data,filter)
```

`filter` can be also any of the following:

```
filter = idm
filter = {num,den}
filter = {A,B,C,D}
```

`idm` is a SISO identified linear model or LTI object. For more information about LTI objects, see the Control System Toolbox documentation.

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

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

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

```
filter = Wf
```

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

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

More About

"Filtering Data" on page 2-120

Causal and Noncausal Filters

For time-domain data, the filtering is causal by default. Causal filters typically introduce a phase shift in the results. To use a noncausal zero-phase filter (corresponding to `filtfilt` in the Signal Processing Toolbox product), specify a third argument in `idfilt`:

```
fdata = idfilt(data,filter,'noncausal')
```

For frequency-domain data, the signals are multiplied by the frequency response of the filter. With the filters defined as passband filters, this calculation gives ideal, zero-phase filtering ("brick wall filters"). Frequencies that have been assigned zero weight by the filter (outside the passband or via frequency response) are removed.

When you apply `idfilt` to an `idfrd` data object, the data is first converted to a frequency-domain `iddata` object (see "Transforming Between Frequency-Domain and Frequency-Response Data" on page 2-141). The result is an `iddata` object.

More About

"Filtering Data" on page 2-120

Generating Data Using Simulation

In this section...

Commands for Generating Data Using Simulation

You can generate input data and then use it with a model to create output data.

Simulating output data requires that you have a model with known coefficients. For more information about commands for constructing models, see "Commands for Constructing Model Structures" on page 1-25.

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

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

Commands for Generating Data

Command	Description	Example
`idinput`	Constructs a signal with the desired characteristics, such as a random Gaussian or binary signal or a	`u = iddata([],...` `    idinput(400,'rbs',[0 0.3]));`

Commands for Generating Data (Continued)

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

Create Periodic Input Data

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

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

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

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

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

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

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

4. (Optional) Simulate model output using the data.

   ```
   % Construct a polynomial model.
   m0 =idpoly([1 -1.5 0.7],[0 1 0.5]);
   ```

```
% Simulate model output with Gaussian noise.
sim_opt = simOptions('AddNoise',true);
sim(m0,u,sim_opt);
```

Generate Output Data Using Simulation

This example shows how to generate output data by simulating a model using an input signal created using `idinput`.

You use the generated data to estimate a model of the same order as the model used to generate the data. Then, you check how closely both models match to understand the effects of input data characteristics and noise on the estimation.

1. Create an ARMAX model with known coefficients.

   ```
   A = [1 -1.2 0.7];
   B = {[0 1 0.5 0.1],[0 1.5 -0.5],[0 -0.1 0.5 -0.1]};
   C = [1 0 0 0 0];
   Ts = 1;
   m = idpoly(A,B,C,'Ts',1);
   ```

   The leading zeros in the `B` matrix indicate the input delay (`nk`), which is `1` for each input channel.

2. Construct a pseudorandom binary input data.

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

3. Simulate model output with noise using the input data.

   ```
   sim(m0,u,simOptions('AddNoise',true))
   ```

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

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

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

   ```
   na = 2; nb = [3 2 3]; nc = 4; nk = [1 1 1];
   me = armax(iodata,[na,nb,nc,nk]);
   ```

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

Simulating Data Using Other MathWorks Products

You can also simulate data using the Simulink and Signal Processing Toolbox software. Data simulated outside the System Identification Toolbox product must be in the MATLAB workspace as double matrices. For more information about simulating models using the Simulink software, see "Simulating Identified Model Output in Simulink" on page 11-5.

Transforming Between Time- and Frequency-Domain Data

In this section...

Transforming Data Domain in the GUI

Transforming Time-Domain Data

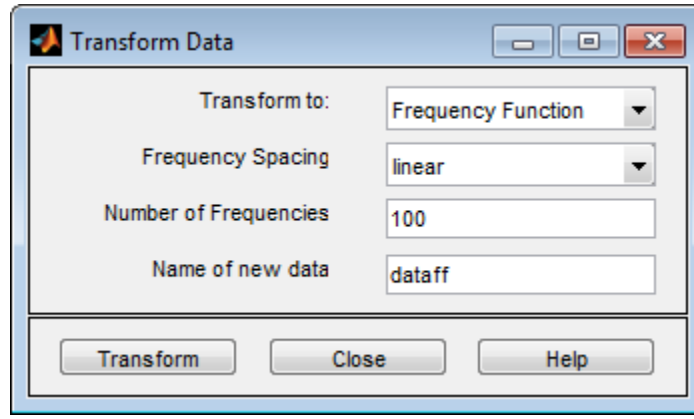
In the System Identification Tool GUI, time-domain data has an icon with a white background. You can transform time-domain data to frequency-domain or frequency-response data. The frequency values of the resulting frequency vector range from `0` to the Nyquist frequency $f_S = \pi/T_s$, where T_s is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

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

1. In the System Identification Tool GUI, drag the icon of the data you want to transform to the **Working Data** rectangle.
2. In the **Operations** area, select **<--Preprocess > Transform data** in the drop-down menu to open the Transform Data dialog box.

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

- `Frequency Function` — Create a new `idfrd` object using the `spafdr` method. Go to step 4.



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

4 In the **Frequency Spacing** list, select the spacing of the frequencies at which the frequency function is estimated:

- `linear` — Uniform spacing of frequency values between the endpoints.
- `logarithmic` — Base-10 logarithmic spacing of frequency values between the endpoints.

5 In the **Number of Frequencies** field, enter the number of frequency values.

6 In the **Name of new data** field, type the name of the new data set. This name must be unique in the Data Board.

7 Click **Transform** to add the new data set to the Data Board in the System Identification Tool GUI.

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

Transforming Frequency-Domain Data

In the System Identification Tool GUI, frequency-domain data has an icon with a green background. You can transform frequency-domain data to time-domain or frequency-response (frequency-function) data.

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

1. In the System Identification Tool GUI, drag the icon of the data you want to transform to the **Working Data** rectangle.
2. Select **<--Preprocess > Transform data**.
3. In the **Transform to** list, select one of the following:
   - `Frequency Function` — Create a new `idfrd` object using the `spafdr` method. Go to step 4.
   - `Time Domain Data` — Create a new `iddata` object using the `ifft` (inverse fast Fourier transform) method. Go to step 6.
4. In the **Frequency Spacing** list, select the spacing of the frequencies at which the frequency function is estimated:
   - `linear` — Uniform spacing of frequency values between the endpoints.
   - `logarithmic` — Base-10 logarithmic spacing of frequency values between the endpoints.
5. In the **Number of Frequencies** field, enter the number of frequency values.
6. In the **Name of new data** field, type the name of the new data set. This name must be unique in the Data Board.
7. Click **Transform** to add the new data set to the Data Board in the System Identification Tool GUI.
8. Click **Close** to close the Transform Data dialog box.

Transforming Frequency-Response Data

In the System Identification Tool GUI, frequency-response data has an icon with a yellow background. You can transform frequency-response data to frequency-domain data (`iddata` object) or to frequency-response data with a different frequency resolution.

When you select to transform single-input/single-output (SISO) frequency-response data to frequency-domain data, the toolbox creates outputs that equal the frequency responses, and inputs equal to 1. Therefore, the ratio between the Fourier transform of the output and the Fourier transform of the input is equal to the system frequency response.

For the multiple-input case, the toolbox transforms the frequency-response data to frequency-domain data as if each input contributes independently to the entire output of the system and then combines information. For example, if a system has three inputs, `u1`, `u2`, and `u3` and two frequency samples, the input matrix is set to:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

In general, for `nu` inputs and `ns` samples (the number of frequencies), the input matrix has `nu` columns and (`ns` · `nu`) rows.

Note To create a separate experiment for the response from each input, see "Transforming Between Frequency-Domain and Frequency-Response Data" on page 2-141.

When you transform frequency-response data by changing its frequency resolution, you can modify the number of frequency values by changing between linear or logarithmic spacing. You might specify variable frequency spacing to increase the number of data points near the system resonance

frequencies, and also make the frequency vector coarser in the region outside the system dynamics. Typically, high-frequency noise dominates away from frequencies where interesting system dynamics occur. The System Identification Tool GUI lets you specify logarithmic frequency spacing, which results in a variable frequency resolution.

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

---

1. In the System Identification Tool GUI, drag the icon of the data you want to transform to the **Working Data** rectangle.
2. Select **<--Preprocess > Transform data**.
3. In the **Transform to** list, select one of the following:
   - `Frequency Domain Data` — Create a new `iddata` object. Go to step 6.
   - `Frequency Function` — Create a new `idfrd` object with different resolution (number and spacing of frequencies) using the `spafdr` method. Go to step 4.
4. In the **Frequency Spacing** list, select the spacing of the frequencies at which the frequency function is estimated:
   - `linear` — Uniform spacing of frequency values between the endpoints.
   - `logarithmic` — Base-10 logarithmic spacing of frequency values between the endpoints.
5. In the **Number of Frequencies** field, enter the number of frequency values.
6. In the **Name of new data** field, type the name of the new data set. This name must be unique in the Data Board.
7. Click **Transform** to add the new data set to the Data Board in the System Identification Tool GUI.
8. Click **Close** to close the Transform Data dialog box.

See Also

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

To learn how to transform data at the command line instead of the GUI, see "Transforming Data Domain at the Command Line" on page 2-139.

Transforming Data Domain at the Command Line

Supported Data Transformations

The following table shows the different ways you can transform data from one data domain to another. If the transformation is supported for a given row and column combination in the table, the method used by the software is listed in the cell at their intersection.

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

Original Data Format	To Time Domain (iddata object)	To Frequency Domain (iddata object)	To Frequency Function (idfrd object)
		that has the same ratio between output and input as the original `idfrd` object's response data.	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

You can transform frequency-response data to frequency-domain data (`iddata` object). The `idfrd` object represents complex frequency-response of the system at different frequencies. For a description of this type of data, see "Frequency-Response Data Representation" on page 2-13.

When you select to transform single-input/single-output (SISO) frequency-response data to frequency-domain data, the toolbox creates outputs that equal the frequency responses, and inputs equal to 1. Therefore, the ratio between the Fourier transform of the output and the Fourier transform of the input is equal to the system frequency response.

For information about changing the frequency resolution of frequency-response data to a new constant or variable (frequency-dependent) resolution, see the `spafdr` reference page. You might use this feature to increase the number of data points near the system resonance frequencies and make the frequency vector coarser in the region outside the system dynamics. Typically, high-frequency noise dominates away from frequencies where interesting system dynamics occur.

Note You cannot transform an `idfrd` object to a time-domain `iddata` object.

To transform an `idfrd` object with the name `idfrdobj` to a frequency-domain `iddata` object, use the following syntax:

```
dataf = iddata(idfrdobj)
```

The resulting frequency-domain `iddata` object contains values at the same frequencies as the original `idfrd` object.

For the multiple-input case, the toolbox represents frequency-response data as if each input contributes independently to the entire output of the system and then combines information. For example, if a system has three inputs, `u1`, `u2`, and `u3` and two frequency samples, the input matrix is set to:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

In general, for `nu` inputs and `ns` samples, the input matrix has `nu` columns and (`ns` · `nu`) rows.

If you have `ny` outputs, the transformation operation produces an output matrix has `ny` columns and (`ns` · `nu`) rows using the values in the complex frequency response *G(iw)* matrix (`ny`-by-`nu`-by-`ns`). In this example, `y1` is determined by unfolding `G(1,1,:)`, `G(1,2,:)`, and `G(1,3,:)` into three column vectors and vertically concatenating these vectors into a single column. Similarly, `y2` is determined by unfolding `G(2,1,:)`, `G(2,2,:)`, and `G(2,3,:)` into three column vectors and vertically concatenating these vectors.

If you are working with multiple inputs, you also have the option of storing the contribution by each input as an independent experiment in a multiexperiment data set. To transform an `idfrd` object with the name `idfrdobj` to a multiexperiment data set `datf`, where each experiment corresponds to each of the inputs in `idfrdobj`

```
datf = iddata(idfrdobj,'me')
```

In this example, the additional argument `'me'` specifies that multiple experiments are created.

By default, transformation from frequency-response to frequency-domain data strips away frequencies where the response is `inf` or `NaN`. To preserve the entire frequency vector, use `datf = iddata(idfrdobj,'inf')`. For more information, type `help idfrd/iddata`.

See Also

Transforming from time-domain or frequency-domain data to frequency-response data is equivalent to creating a frequency-response model

from the data. For more information, see “Identifying Frequency-Response Models” on page 3-8.

Manipulating Complex-Valued Data

In this section...

Supported Operations for Complex Data

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

Model transformation routines, such as `freqresp` and `zpkdata`, work for complex-valued models. However, they do not provide pole-zero confidence regions. For complex models, the parameter variance-covariance information refers to the complex-valued parameters and the accuracy of the real and imaginary is not computed separately.

The display commands `compare` and `plot` also work with complex-valued data and models. To plot the real and imaginary parts of the data separately, use `plot(real(data))` and `plot(imag(data))`, respectively.

Processing Complex iddata Signals at the Command Line

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

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

Command	Description
`complex(data)`	For time-domain data, this command makes the `iddata` object complex—even when the imaginary parts are zero. For frequency-domain data that only stores the values for nonnegative frequencies, such that `realdata(data)=1`, it adds signal values for negative frequencies using complex conjugation.
`imag(data)`	Selects the imaginary parts of each signal in `iddata` object.
`isreal(data)`	`1` when `data` (time-domain or frequency-domain) contains only real input and output signals, and returns `0` when `data` (time-domain or frequency-domain) contains complex signals.
`real(data)`	Real part of complex signals in `iddata` object.
`realdata(data)`	Returns a value of `1` when `data` is a real-valued, time-domain signal, and returns `0` otherwise.

For example, suppose that you create a frequency-domain `iddata` object `Datf` by applying `fft` to a real-valued time-domain signal to take the Fourier transform of the signal. The following is true for `Datf`:

```
isreal(Datf) = 0
realdata(Datf) = 1
```

3

Linear Model Identification

Black-Box Modeling

In this section...

Selecting Black-Box Model Structure and Order

Black-box modeling is useful when your primary interest is in fitting the data regardless of a particular mathematical structure of the model. The toolbox provides several linear and nonlinear black-box model structures, which have traditionally been useful for representing dynamic systems. These model structures vary in complexity depending on the flexibility you need to account for the dynamics and noise in your system. You can choose one of these structures and compute its parameters to fit the measured response data.

Black-box modeling is usually a trial-and-error process, where you estimate the parameters of various structures and compare the results. Typically, you start with the simple linear model structure and progress to more complex structures. You might also choose a model structure because you are more familiar with this structure or because you have specific application needs.

The simplest linear black-box structures require the fewest options to configure:

- Transfer function, with a given number of poles and zeros.
- Linear ARX model, which is the simplest input-output polynomial model.
- State-space model, which you can estimate by specifying the number of model states

Estimation of some of these structures also uses noniterative estimation algorithms, which further reduces complexity.

You can configure a model structure using the *model order*. The definition of model order varies depending on the type of model you select. For example, if you choose a transfer function representation, the model order is related to the

number of poles and zeros. For state-space representation, the model order corresponds to the number of states. In some cases, such as for linear ARX and state-space model structures, you can estimate the model order from the data.

If the simple model structures do not produce good models, you can select more complex model structures by:

- Specifying a higher model order for the same linear model structure. Higher model order increases the model flexibility for capturing complex phenomena. However, unnecessarily high orders can make the model less reliable.
- Explicitly modeling the noise:

  $y(t)=Gu(t)+He(t)$

  where H models the additive disturbance by treating the disturbance as the output of a linear system driven by a white noise source $e(t)$.

  Using a model structure that explicitly models the additive disturbance can help to improve the accuracy of the measured component G. Furthermore, such a model structure is useful when your main interest is using the model for predicting future response values.
- Using a different linear model structure.

  See "Linear Model Structures" on page 1-21 in the User's Guide.
- Using a nonlinear model structure.

  Nonlinear models have more flexibility in capturing complex phenomena than linear models of similar orders. See "Available Nonlinear Models" on page 1-32 in User's Guide.

Ultimately, you choose the simplest model structure that provides the best fit to your measured data. For more information, see "Estimating Linear Models Using Quick Start".

Regardless of the structure you choose for estimation, you can simplify the model for your application needs. For example, you can separate out the measured dynamics (G) from the noise dynamics (H) to obtain a simpler model that represents just the relationship between y and u. You can also linearize a nonlinear model about an operating point.

When to Use Nonlinear Model Structures?

A linear model is often sufficient to accurately describe the system dynamics and, in most cases, you should first try to fit linear models. If the linear model output does not adequately reproduce the measured output, you might need to use a nonlinear model.

You can assess the need to use a nonlinear model structure by plotting the response of the system to an input. If you notice that the responses differ depending on the input level or input sign, try using a nonlinear model. For example, if the output response to an input step up is faster than the response to a step down, you might need a nonlinear model.

Before building a nonlinear model of a system that you know is nonlinear, try transforming the input and output variables such that the relationship between the transformed variables is linear. For example, consider a system that has current and voltage as inputs to an immersion heater, and the temperature of the heated liquid as an output. The output depends on the inputs via the power of the heater, which is equal to the product of current and voltage. Instead of building a nonlinear model for this two-input and one-output system, you can create a new input variable by taking the product of current and voltage and then build a linear model that describes the relationship between power and temperature.

If you cannot determine variable transformations that yield a linear relationship between input and output variables, you can use nonlinear structures such as Nonlinear ARX or Hammerstein-Wiener models. For a list of supported nonlinear model structures and when to use them, see “Available Nonlinear Models” on page 1-32 in User’s Guide.

Black-Box Estimation Example

You can use the GUI or commands to estimate linear and nonlinear models of various structures. In most cases, you choose a model structure and estimate the model parameters using a single command.

Consider the mass-spring-damper system, described in “About Dynamic Systems and Models”. If you do not know the equation of motion of this system, you can use a black-box modeling approach to build a model. For example, you can estimate transfer functions or state-space models by specifying the orders of these model structures.

A transfer function is a ratio of polynomials:

$$G(s) = \frac{\left(b_0 + b_1 s + b_2 s^2 + ...\right)}{\left(1 + f_1 s + f_2 s^2 + ...\right)}$$

For the mass-spring damper system, this transfer function is:

$$G(s) = \frac{1}{\left(ms^2 + cs + k\right)}$$

which is a system with no zeros and 2 poles.

In discrete-time, the transfer function of the mass-spring-damper system can be:

$$G\left(z^{-1}\right) = \frac{bz^{-1}}{\left(1 + f_1 z^{-1} + f_2 z^{-2}\right)}$$

where the model orders correspond to the number of coefficients of the numerator and the denominator (`nb` = 1 and `nf` = 2) and the input-output delay equals the lowest order exponent of z^{-1} in the numerator (`nk` = 1).

In continuous-time, you can build a linear transfer function model using the `tfest` command:

```
m = tfest(data, 2, 0)
```

where `data` is your measured input-output data, represented as an `iddata` object and the model order is the set of number of poles (2) and the number of zeros (0).

Similarly, you can build a discrete-time model Output Error structure using the following command:

```
m = oe(data, [1 2 1])
```

The model order is [`nb nf nk`] = [`1 2 1`]. Usually, you do not know the model orders in advance. You should try several model order values until you find the orders that produce an acceptable model.

Alternatively, you can choose a state-space structure to represent the mass-spring-damper system and estimate the model parameters using the `ssest` or the `n4sid` command:

```
m = ssest(data, 2)
```

where *`order`* = `2` represents the number of states in the model.

In black-box modeling, you do not need the system's equation of motion—only a guess of the model orders.

For more information about building models, see "Steps for Using the System Identification Tool GUI" on page 12-2 and "Model Estimation Commands" on page 1-40 in the User's Guide.

Identifying Frequency-Response Models

In this section...

What Is a Frequency-Response Model?

You can estimate *frequency-response models* and visualize the responses on a Bode plot, which shows the amplitude change and the phase shift as a function of the sinusoid frequency.

The frequency-response function describes the steady-state response of a system to sinusoidal inputs. For a linear system, a sinusoidal input of a specific frequency results in an output that is also a sinusoid with the same frequency, but with a different amplitude and phase. The frequency-response function describes the amplitude change and phase shift as a function of frequency.

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

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

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

The estimation result is an `idfrd` model, which stores the estimated frequency response and its covariance.

Data Supported by Frequency-Response Models

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

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

How to Estimate Frequency-Response Models in the GUI

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

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

1. In the System Identification Tool GUI, select **Estimate > Spectral models** to open the Spectral Model dialog box.
2. In the **Method** list, select the spectral analysis method you want to use. For information about each method, see "Selecting the Method for Computing Spectral Models" on page 3-11.
3. Specify the frequencies at which to compute the spectral model in *one* of the following ways:
   - In the **Frequencies** field, enter either a vector of values, a MATLAB expression that evaluates to a vector, or a variable name of a vector in the MATLAB workspace. For example, `logspace(-1,2,500)`.
   - Use the combination of **Frequency Spacing** and **Frequencies** to construct the frequency vector of values:

- In the **Frequency Spacing** list, select `Linear` or `Logarithmic` frequency spacing.

  Note For `etfe`, only the `Linear` option is available.

- In the **Frequencies** field, enter the number of frequency points.

  For time-domain data, the frequency ranges from 0 to the Nyquist frequency. For frequency-domain data, the frequency ranges from the smallest to the largest frequency in the data set.

4. In the **Frequency Resolution** field, enter the frequency resolution, as described in "Controlling Frequency Resolution of Spectral Models " on page 3-12. To use the default value, enter `default` or, equivalently, the empty matrix `[]`.

5. In the **Model Name** field, enter the name of the correlation analysis model. The model name should be unique in the Model Board.

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

7. In the Spectral Model dialog box, click **Close**.

8. To view the frequency-response plot, select the **Frequency resp** check box in the System Identification Tool GUI. For more information about working with this plot, see "Frequency Response Plots" on page 8-42.

9. To view the estimated disturbance spectrum, select the **Noise spectrum** check box in the System Identification Tool GUI. For more information about working with this plot, see "Noise Spectrum Plots" on page 8-51.

10. Validate the model after estimating it. For more information, see "Model Validation".

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

How to Estimate Frequency-Response Models at the Command Line

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

The resulting models are stored as `idfrd` model objects. For detailed information about the commands and their arguments, see the corresponding reference page.

Commands for Frequency Response

Command	Description	Usage
`etfe`	Estimates an empirical transfer function using Fourier analysis.	To estimate a model `m`, use the following syntax: `m=etfe(data)`
`spa`	Estimates a frequency response with a fixed frequency resolution using spectral analysis.	To estimate a model `m`, use the following syntax: `m=spa(data)`
`spafdr`	Estimates a frequency response with a variable frequency resolution using spectral analysis.	To estimate a model `m`, use the following syntax: `m=spafdr(data,R,w)` where `R` is the resolution vector and `w` is the frequency vector.

Validate the model after estimating it. For more information, see "Model Validation".

Selecting the Method for Computing Spectral Models

This section describes how to select the method for computing spectral models in the estimation procedures "How to Estimate Frequency-Response Models in the GUI" on page 3-9 and "How to Estimate Frequency-Response Models at the Command Line" on page 3-11.

You can choose from the following three spectral-analysis methods:

- `etfe` (**E**mpirical **T**ransfer **F**unction **E**stimate)

  For input-output data. This method computes the ratio of the Fourier transform of the output to the Fourier transform of the input.

  For time-series data. This method computes a periodogram as the normalized absolute squares of the Fourier transform of the time series.

  ETFE works well for highly resonant systems or narrowband systems. The drawback of this method is that it requires linearly spaced frequency values, does not estimate the disturbance spectrum, and does not provide confidence intervals. ETFE also works well for periodic inputs and computes exact estimates at multiples of the fundamental frequency of the input and their ratio.

- `spa` (**SP**ectral **A**nalysis)

  This method is the Blackman-Tukey spectral analysis method, where windowed versions of the covariance functions are Fourier transformed.

- `spafdr` (**SP**ectral **A**nalysis with **F**requency **D**ependent **R**esolution)

  This method is a variant of the Blackman-Tukey spectral analysis method with frequency-dependent resolution. First, the algorithm computes Fourier transforms of the inputs and outputs. Next, the products of the transformed inputs and outputs with the conjugate input transform are smoothed over local frequency regions. The widths of the local frequency regions can vary as a function of frequency. The ratio of these averages computes the frequency-response estimate.

Controlling Frequency Resolution of Spectral Models

This section supports the estimation procedures "How to Estimate Frequency-Response Models in the GUI" on page 3-9 and "How to Estimate Frequency-Response Models at the Command Line" on page 3-11.

What Is Frequency Resolution?

Frequency resolution is the size of the smallest frequency for which details in the frequency response and the spectrum can be resolved by the estimate. A resolution of 0.1 rad/s means that the frequency response variations at frequency intervals at or below 0.1 rad/s are not resolved.

Note Finer resolution results in greater uncertainty in the model estimate.

Specifying the frequency resolution for `etfe` and `spa` is different than for `spafdr`.

Frequency Resolution for etfe and spa

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

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

M is a scalar integer that sets the size of the lag window. The value of `M` controls the trade-off between bias and variance in the spectral estimate.

The default value of M for `spa` is good for systems without sharp resonances. For `etfe`, the default value of M gives the maximum resolution.

A large value of M gives good resolution, but results in more uncertain estimates. If a true frequency function has sharp peak, you should specify higher `M` values.

Frequency Resolution for spafdr

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

Such customizing of the frequency grid assists in the estimation process by achieving high fidelity in the frequency range of interest.

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

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

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

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

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

etfe Frequency Resolution for Periodic Input

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

For periodic data, the frequency resolution is ignored.

Spectrum Normalization

The *spectrum* of a signal is the square of the Fourier transform of the signal. The spectral estimate using the commands `spa`, `spafdr`, and `etfe` is normalized by the sampling interval T:

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

where $W_M(k)$ is the lag window, and M is the width of the lag window. The output covariance $R_y(kT)$ is given by the following discrete representation:

$$\hat{R}_y(kT) = \frac{1}{N}\sum_{l=1}^{N} y(lT - kT)y(lT)$$

Because there is no scaling in a discrete Fourier transform of a vector, the purpose of T is to relate the discrete transform of a vector to the physically meaningful transform of the measured signal. This normalization sets the

units of $\Phi_y(\omega)$ as power per radians per unit time, and makes the frequency units radians per unit time.

The scaling factor of T is necessary to preserve the energy density of the spectrum after interpolation or decimation.

By Parseval's theorem, the average energy of the signal must equal the average energy in the estimated spectrum, as follows:

$$Ey^2(t) = \frac{1}{2\pi}\int_{-\pi/T}^{\pi/T} \Phi_y(\omega)d\omega$$

$$S1 \equiv Ey^2(t)$$

$$S2 \equiv \frac{1}{2\pi}\int_{-\pi/T}^{\pi/T} \Phi_y(\omega)d\omega$$

To compare the left side of the equation (`S1`) to the right side (`S2`), enter the following commands in the MATLAB Command Window:

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

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

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

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

MATLAB computes the following values for `S1` and `S2`:

```
S1 =

   19.2076
S2 =

   19.4646
```

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

Identifying Impulse-Response Models

In this section...

What Is Time-Domain Correlation Analysis?

Time-domain correlation analysis refers to non-parametric estimation of the impulse response of dynamic systems as a finite impulse response (FIR) model from the data. Correlation analysis assumes a linear system and does not require a specific model structure.

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

$$u(t) = 0 \quad t > 0$$
$$u(t) = 1 \quad t = 0$$

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

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

Data Supported by Correlation Analysis

You can estimate impulse-response models from data with the following characteristics:

- Real or complex data.
- Single- or multiple-output data.
- Time- or frequency-domain data with nonzero sampling time.

  Time-domain data must be regularly sampled. You cannot use time-series data for correlation analysis.

How to Estimate Impulse-Response Models Using the GUI

Before you can perform this task, you must have:

- Imported data into the System Identification Tool GUI. See "Importing Time-Domain Data into the GUI" on page 2-18. For supported data formats, see "Data Supported by Correlation Analysis" on page 3-17.
- Performed any required data preprocessing operations. To improve the accuracy of your model, you should detrend your data. See "Ways to Prepare Data for System Identification" on page 2-6.

To estimate in the System Identification Tool GUI using time-domain correlation analysis:

1. In the System Identification Tool GUI, select **Estimate > Correlation models** to open the Correlation Model dialog box.
2. In the **Time span (s)** field, specify a scalar value as the time interval over which the impulse or step response is calculated. For a scalar time span *T*, the resulting response is plotted from *-T/4* to *T*.

   Tip You can also enter a 2-D vector in the format `[min_value max_value]`.

3. In the **Order of whitening filter** field, specify the filter order.

   The prewhitening filter is determined by modeling the input as an autoregressive process of order *N*. The algorithm applies a filter of the form *A(q)u(t)=u_F(t).* That is, the input *u(t)* is subjected to an FIR filter *A* to produce the filtered signal *u_F(t). Prewhitening* the input by applying

a whitening filter before estimation might improve the quality of the estimated impulse response *g*.

The order of the prewhitening filter, *N*, is the order of the *A* filter. *N* equals the number of lags. The default value of *N* is `10`, which you can also specify as `[]`.

4. In the **Model Name** field, enter the name of the correlation analysis model. The name of the model should be unique in the Model Board.

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

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

Next Steps

- Export the model to the MATLAB workspace for further analysis by dragging it to the **To Workspace** rectangle in the System Identification Tool GUI.
- View the transient response plot by selecting the **Transient resp** check box in the System Identification Tool GUI. For more information about working with this plot and selecting to view impulse- versus step-response, see "Impulse and Step Response Plots" on page 8-33.

How to Estimate Impulse-Response Models at the Command Line

Before you can perform this task, you must have:

- Input/output or frequency-response data. See "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-55. For supported data formats, see "Data Supported by Correlation Analysis" on page 3-17.
- Performed any required data preprocessing operations. If you use time-domain data, you can detrend it before estimation. See "Ways to Prepare Data for System Identification" on page 2-6.

Use `impulseest` to compute impulse response models. `impulseest` estimates a high-order, noncausal FIR model using correlation analysis. The resulting models are stored as `idtf` model objects and contain impulse-response coefficients in the model numerator.

To estimate the model `m` and plot the impulse or step response, use the following syntax:

```
m=impulseest(data,N);
impulse(m,Time);
step(m,Time);
```

where `data` is a single- or multiple-output `iddata` or `idfrd` object. `N` is a scalar value specifying the order of the FIR system corresponding to the time range 0:Ts:(N-1)*Ts, where *Ts* is the data sampling time.

You can also specify estimation options, such as regularizing kernel, pre-whitening filter order and data offsets, using `impulseestOptions` and pass them as an input to `impulseest`. For example:

```
opt = impulseestOptions('RegulKernel','TC'));
m = impulseest(data,N,opt);
```

To view the confidence region for the estimated response, use `impulseplot` and `stepplot` to create the plot. Then use `showConfidence`.

For example:

```
h = stepplot(m,Time);
showConfidence(h,3) % 3 std confidence region
```

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

Next Steps

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

How to Compute Response Values

You can use `impulse` and `step` commands with output arguments to get the numerical impulse- and step-response vectors as a function of time, respectively.

To get the numerical response values:

1. Compute the FIR model by using `impulseest`, as described in "How to Estimate Impulse-Response Models at the Command Line" on page 3-19.

2. Apply the following syntax on the resulting model:

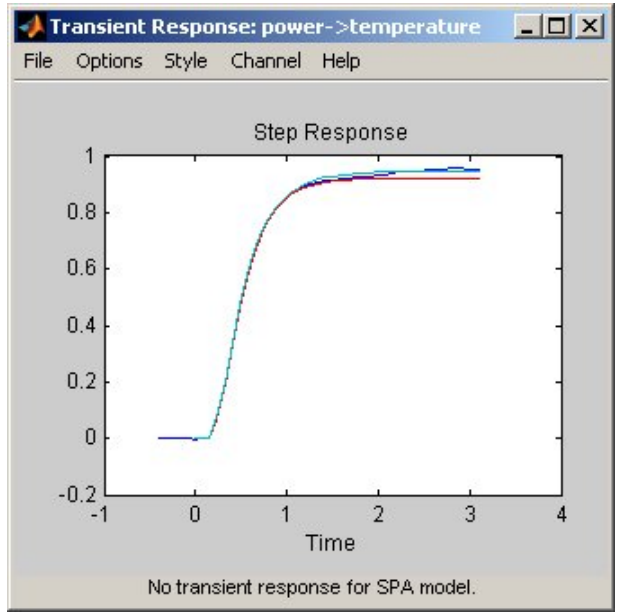
   ```
   % To compute impulse-response data
   [y,t,~,ysd] = impulse(model)
   % To compute step-response data
   [y,t,~,ysd] = step(model)
   ```

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

How to Identify Delay Using Transient-Response Plots

You can use transient-response plots to estimate the input delay, or *dead time*, of linear systems. Input delay represents the time it takes for the output to respond to the input.

In the System Identification Tool GUI. To view the transient response plot, select the **Transient resp** check box in the System Identification Tool GUI. For example, the following step response plot shows a time delay of about 0.25 s before the system responds to the input.



Step Response Plot

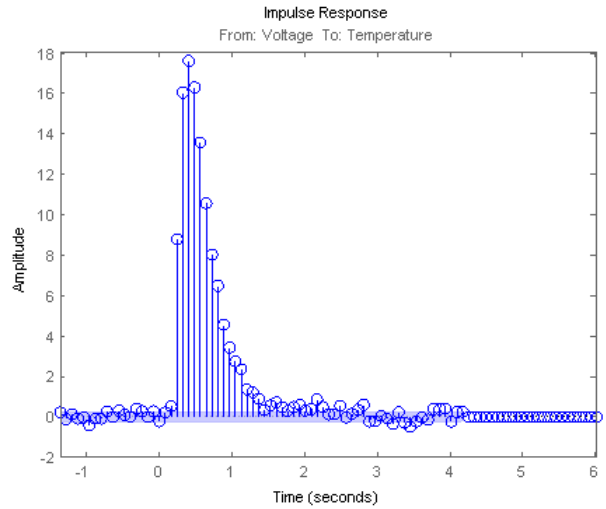
At the command line. You can use the `impulseplot` command to plot the impulse response. The time delay is equal to the first positive peak in the transient response magnitude that is greater than the confidence region for positive time values.

For example, the following commands create an impulse-response plot with a 1-standard-deviation confidence region:

```
% Load sample data
load dry2
ze = dry2(1:500);
opt = impulseestOptions('RegulKernel','TC');
sys = impulseest(ze,40,opt);
h = impulseplot(sys);
showConfidence(h,1);
```

Instead of using `showConfidence`, you can plot the confidence interval interactively, by right-clicking on the plot and selecting **Characteristics > Confidence Region**.

The resulting figure shows that the first positive peak of the response magnitude, which is greater than the confidence region for positive time values, occurs at 0.24 s.



Correlation Analysis Algorithm

Correlation analysis refers to methods that estimate the impulse response of a linear model, without specific assumptions about model orders.

The impulse response, g, is the system's output when the input is an impulse signal. The output response to a general input, $u(t)$, is obtained as the convolution with the impulse response. In continuous time:

$$y(t) = \int_{-\infty}^{t} g(\tau)u(t-\tau)d\tau$$

In discrete-time:

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

The values of $g(k)$ are the *discrete time impulse response coefficients*.

You can estimate the values from observed input-output data in several different ways. `impulseest` estimates the first n coefficients using the least-squares method to obtain a finite impulse response (FIR) model of order n.

Several important options are associated with the estimate:

- **Prewhitening** — The input can be pre-whitened by applying an input-whitening filter of order `PW` to the data. This minimizes the effect of the neglected tail (`k > n`) of the impulse response.

  1. A filter of order `PW` is applied such that it whitens the input signal `u`:

     `1/A = A(u)e`, where `A` is a polynomial and `e` is white noise.

  2. The inputs and outputs are filtered using the filter:

     `uf = Au`, `yf = Ay`

  3. The filtered signals `uf` and `yf` are used for estimation.

  You can specify prewhitening using the `PW` name-value pair argument of `impulseestOptions`.

- **Regularization** — The least-squares estimate can be regularized. This means that a prior estimate of the decay and mutual correlation among `g(k)` is formed and used to merge with the information about `g` from the observed data. This gives an estimate with less variance, at the price of some bias. You can choose one of the several kernels to encode the prior estimate.

  This option is essential because, often, the model order `n` can be quite large. In cases where there is no regularization, `n` can be automatically decreased to secure a reasonable variance.

  You can specify the regularizing kernel using the `RegulKernel` Name-Value pair argument of `impulseestOptions`.

- **Autoregressive Parameters** — The basic underlying FIR model can be complemented by `NA` autoregressive parameters, making it an ARX model.

$$y(t) = \sum_{k=1}^{n} g(k)u(t-k) - \sum_{k=1}^{NA} a_k y(t-k)$$

This gives both better results for small `n` and allows unbiased estimates when data are generated in closed loop. `impulseest` uses NA = 5 for t>0 and NA = 0 (no autoregressive component) for t<0.

- **Noncausal effects** — Response for negative lags. It may happen that the data has been generated partly by output feedback:

$$u(t) = \sum_{k=0}^{\infty} h(k)y(t-k) + r(t)$$

where $h(k)$ is the impulse response of the regulator and r is a setpoint or disturbance term. The existence and character of such feedback h can be estimated in the same way as g, simply by trading places between y and u in the estimation call. Using `impulseest` with an indication of negative delays, mi = impulseest($data, nk, nb$), $nk < 0$, returns a model `mi` with an impulse response

$$\left[h(-nk), h(-nk-1), \ldots, h(0), g(1), g(2), \ldots, g(nb+nk)\right]$$

aligned so that it corresponds to lags $\left[nk, nk+1, .., 0, 1, 2, \ldots, nb+nk\right]$. This is achieved because the input delay (`InputDelay`) of model `mi` is `nk`.

For a multi-input multi-output system, the impulse response $g(k)$ is an ny-by-nu matrix, where ny is the number of outputs and nu is the number of inputs. The i–j element of the matrix $g(k)$ describes the behavior of the ith output after an impulse in the jth input.

Identifying Process Models

In this section...

What Is a Process Model?

The structure of a *process model* is a simple continuous-time transfer function that describes linear system dynamics in terms of one or more of the following elements:

- Static gain K_p.
- One or more time constants T_{pk}. For complex poles, the time constant is called T_ω—equal to the inverse of the natural frequency—and the damping coefficient is ζ (`zeta`).
- Process zero T_z.
- Possible time delay T_d before the system output responds to the input (*dead time*).
- Possible enforced integration.

Process models are popular for describing system dynamics in many industries and apply to various production environments. The advantages of

these models are that they are simple, support transport delay estimation, and the model coefficients have an easy interpretation as poles and zeros.

You can create different model structures by varying the number of poles, adding an integrator, or adding or removing a time delay or a zero. You can specify a first-, second-, or third-order model, and the poles can be real or complex (underdamped modes).

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

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

Data Supported by Process Models

You can estimate low-order (up to third order), continuous-time transfer functions using regularly sampled time- or frequency-domain `iddata` or `idfrd` data objects. The frequency-domain data may have a zero sample time.

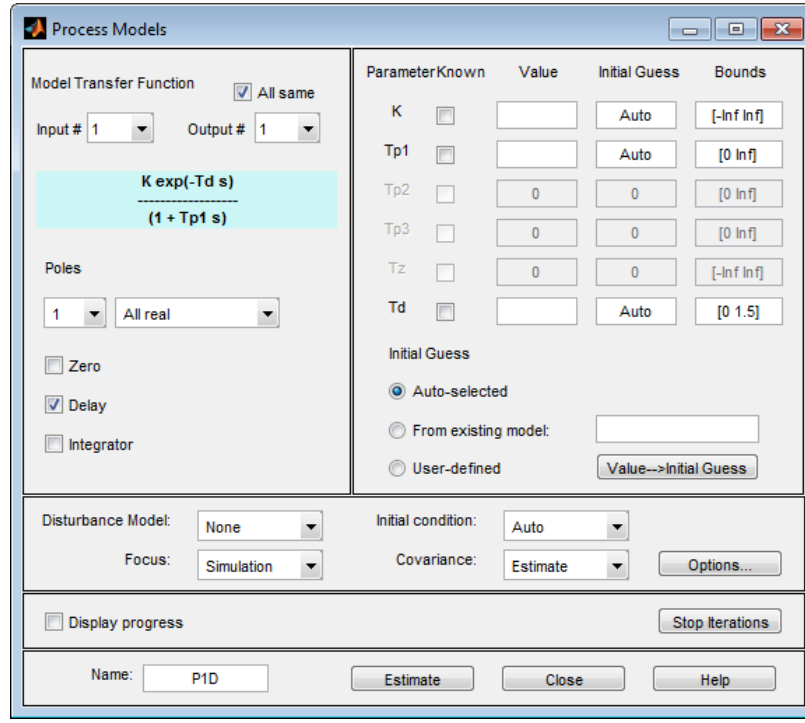
You must import your data into the MATLAB workspace, as described in "Data Preparation".

How to Estimate Process Models Using the GUI

Before you can perform this task, you must have

- Imported data into the System Identification Tool GUI. See "Importing Time-Domain Data into the GUI" on page 2-18. For supported data formats, see "Data Supported by Process Models" on page 3-27.
- Performed any required data preprocessing operations. If you need to model nonzero offsets, such as when model contains integration behavior, do not detrend your data. In other cases, to improve the accuracy of your model, you should detrend your data. See "Ways to Prepare Data for System Identification" on page 2-6.

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



2 If your model contains multiple inputs, select the input channel in the **Input** list. This list only appears when you have multiple inputs. For more information, see "Estimating Multiple-Input, Multi-Output Process Models" on page 3-41.

3 In the **Model Transfer Function** area, specify the model structure using the following options:

- Under **Poles**, select the number of poles, and then select `All real` or `Underdamped`.

  Note You need at least two poles to allow underdamped modes (complex-conjugate pair).

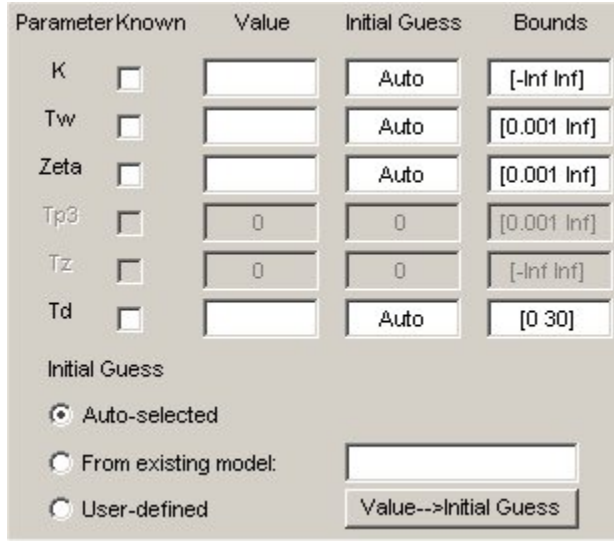
- Select the **Zero** check box to include a zero, which is a numerator term other than a constant, or clear the check box to exclude the zero.
- Select the **Delay** check box to include a delay, or clear the check box to exclude the delay.
- Select the **Integrator** check box to include an integrator (self-regulating process), or clear the check box to exclude the integrator.

The **Parameter** area shows as many active parameters as you included in the model structure.

Note By default, the model **Name** is set to the acronym that reflects the model structure, as described in "Process Model Structure Specification" on page 3-40.

4 In the **Initial Guess** area, select `Auto-selected` to calculate the initial parameter values for the estimation. The **Initial Guess** column in the

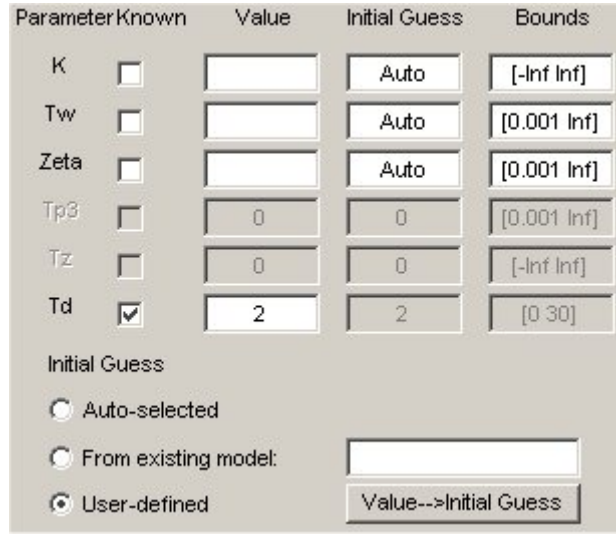
Parameter table displays `Auto`. If you do not have a good guess for the parameter values, `Auto` works better than entering an ad hoc value.



5 (Optional) If you approximately know a parameter value, enter this value in the **Initial Guess** column of the Parameter table. The estimation algorithm uses this value as a starting point. If you know a parameter value exactly, enter this value in the **Initial Guess** column, and also select the corresponding **Known** check box in the table to fix its value.

If you know the range of possible values for a parameter, enter these values into the corresponding **Bounds** field to help the estimation algorithm.

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



6 In the **Disturbance Model** list, select one of the available options. For more information about each option, see "Disturbance Model Structure for Process Models" on page 3-42.

7 In the **Focus** list, select how to weigh the relative importance of the fit at different frequencies. For more information about each option, see "Assigning Estimation Weightings" on page 3-43.

8 In the **Initial state** list, specify how you want the algorithm to treat initial states. For more information about the available options, see "Specifying Initial Conditions for Iterative Estimation Algorithms" on page 3-43.

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, select the **Display Progress** check box. This opens a progress viewer window in which the estimation progress is reported.

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

13 To stop the search and save the results after the current iteration has been completed, click **Stop Iterations**. To continue iterations from the current model, click the **Continue** button to assign current parameter values as initial guesses for the next search.

Next Steps

- Validate the model by selecting the appropriate check box in the **Model Views** area of the System Identification Tool GUI. For more information about validating models, see "Validating Models After Estimation" on page 8-3.
- Refine the model by clicking the **Value —> Initial Guess** button to assign current parameter values as initial guesses for the next search, edit the **Name** field, and click **Estimate**.
- Export the model to the MATLAB workspace for further analysis by dragging it to the **To Workspace** rectangle in the System Identification Tool GUI.

How to Estimate Process Models at the Command Line

- "Example – Estimating Process Models with Free Parameters at the Command Line" on page 3-34
- "Example – Estimating Process Models with Fixed Parameters at the Command Line" on page 3-36

Prerequisites

Before you can perform this task, you must have

- Input-output data as an `iddata` object or frequency response data as `frd` or `idfrd` objects. See "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-55. For supported data formats, see "Data Supported by Process Models" on page 3-27.
- Performed any required data preprocessing operations. When working with time domain data, if you need to model nonzero offsets, such as when model contains integration behavior, do not detrend your data. In other cases, to improve the accuracy of your model, you should detrend your data. See "Ways to Prepare Data for System Identification" on page 2-6.

Using procest to Estimate Process Models

You can estimate process models using the iterative estimation method `procest` that minimizes the prediction errors to obtain maximum likelihood estimates. The resulting models are stored as `idproc` model objects.

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

```
m = procest(data,mod_struc,opt)
```

`data` is the estimation data and `mod_struc` is one of the following:

- A string that represents the process model structure, as described in "Process Model Structure Specification" on page 3-40.
- A template `idproc` model. `opt` is an option set for configuring the estimation of the process model, such as handling of initial conditions, input offset and numerical search method.

Tip You do not need to construct the model object using `idproc` before estimation unless you want to specify initial parameter guesses, minimum/maximum bounds, or fixed parameter values, as described in "Example – Estimating Process Models with Fixed Parameters at the Command Line" on page 3-36.

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

You can use `procest` to refine parameter estimates of an existing process model, as described in "Refining Linear Parametric Models" on page 3-130.

For detailed information, see `procest` and `idproc`.

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

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

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

This process has two inputs and the response from each input is estimated by a first-order process model. All parameters are free to vary.

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, a delay, and a first order disturbance component for this model. The data contains known offsets. Specify them using `InputOffset` and `OutputOffset` options.

```
opt = procestOptions;
opt.InputOffset = [170;50];
opt.OutputOffset = -45;
opt.Display = 'on';
opt.DisturbanceModel = 'arma1';
m=procest(ze,'p1d',opt)
```

MATLAB computes the following output:

```
m =
Process model with 2 inputs: y = G11(s)u1 + G12(s)u2
From input "u1" to output "y1":
           Kp
G11(s) = ---------- * exp(-Td*s)
           1+Tp1*s

      Kp = 2.6542
     Tp1 = 0.15451
      Td = 2.3185

From input "u2" to output "y1":
           Kp
G12(s) = ---------- * exp(-Td*s)
           1+Tp1*s
```

```
       Kp = 9.9754
      Tp1 = 2.0653
       Td = 4.9195

An additive ARMA disturbance model exists for output "y1":
    y = G u + (C/D)e

    C(s) = s + 2.677
    D(s) = s + 0.6237

Parameterization:
    'P1D'    'P1D'

   Number of free coefficients: 8
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "ze".
Fit to estimation data: 91.07% (prediction focus)
FPE: 2.431, MSE: 2.413
```

Use dot notation to get the value of any model parameter. For example, to get the value of dc gain parameter *Kp*, type:

```
m.Kp
```

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

When you know the values of certain parameters in the model and want to estimate only the values you do not know, you must specify the fixed parameters after creating the `idproc` model object.

Use the following commands to prepare the data and construct a process model with one pole and a delay:

```
% Load sample data
load co2data
% Sampling interval is 0.5 min (known)
Ts = 0.5;
% Split data set into estimation data ze
% and validation data zv
ze = iddata(Output_exp1,Input_exp1,Ts,...
                        'TimeUnit','min');
zv = iddata(Output_exp2,Input_exp2,Ts,...
                        'TimeUnit','min');
mod = idproc({'p1d','p1d'},'TimeUnit','min')
```

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

Use the `Structure` model property to specify the initial guesses for unknown parameters, minimum/maximum parameter bounds and fix known parameters.

Set the value of `Kp` for the second transfer function to `10` and specify it as a fixed parameter. Initialize the delay values for the two transfer functions to 2 and 5 minutes, respectively. Specify them as free estimation parameters.

```
mod.Structure(2).Kp.Value = 10;
mod.Structure(2).Kp.Free = false;

mod.Structure(1).Td.Value = 2;
mod.Structure(2).Td.Value = 5;
```

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

```
mod_proc = procest(ze, mod, opt)
```

MATLAB computes the following result:

```
mod_proc =
Process model with 2 inputs: y = G11(s)u1 + G12(s)u2
From input "u1" to output "y1":
           Kp
G11(s) = ---------- * exp(-Td*s)
           1+Tp1*s

      Kp = 2.7448
     Tp1 = 0.40544
      Td = 1.9745

From input "u2" to output "y1":
           Kp
G12(s) = ---------- * exp(-Td*s)
           1+Tp1*s

      Kp = 10
     Tp1 = 2.0734
      Td = 4.92

An additive ARMA disturbance model exists for output "y1":
    y = G u + (C/D)e

    C(s) = s + 2.702
    D(s) = s + 0.6309

Parameterization:
    'P1D'    'P1D'

   Number of free coefficients: 7
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "ze".
Fit to estimation data: 91.06% (prediction focus)
FPE: 2.435, MSE: 2.419
```

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

Process Model Structure Specification

This topic describes how to specify the model structure in the estimation procedures "How to Estimate Process Models Using the GUI" on page 3-27 and "How to Estimate Process Models at the Command Line" on page 3-32.

In the System Identification Tool GUI. Specify the model structure by selecting the number of real or complex poles, and whether to include a zero, delay, and integrator. The resulting transfer function is displayed in the Process Models dialog box.

At the command line. Specify the model structure using an acronym that includes the following letters and numbers:

- (Required) `P` for a process model
- (Required) `0`, `1`, `2` or `3` for the number of poles
- (Optional) `D` to include a time-delay term e^{-sT_d}
- (Optional) `Z` to include a process zero (numerator term)
- (Optional) `U` to indicate possible complex-valued (underdamped) poles
- (Optional) `I` to indicate enforced integration

Typically, you specify the model-structure acronym as a string argument in the estimation command `procest`:

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

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

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

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

- `procest(data,'P0ID')` to estimate the following structure:

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

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

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

For more information about estimating models, see "How to Estimate Process Models at the Command Line" on page 3-32.

Estimating Multiple-Input, Multi-Output Process Models

If your model contains multiple inputs, multiple outputs, or both, you can specify whether to estimate the same transfer function for all input-output pairs, or a different transfer function for each. The information in this section supports the estimation procedures "How to Estimate Process Models Using the GUI" on page 3-27 and "How to Estimate Process Models at the Command Line" on page 3-32.

In the System Identification Tool GUI. To fit a data set with multiple inputs, or multiple outputs, or both, in the Process Models dialog box, configure the process model settings for one input-output pair at a time. Use the input and output selection lists to switch to a different input/output pair.

If you want the same transfer function to apply to all input/output pairs, select the **All same** check box. To apply a different structure to each channel, leave this check box clear, and create a different transfer function for each input.

At the command line. Specify the model structure as a cell array of acronym strings in the estimation command `procest`. For example, use this command to specify the first-order transfer function for the first input, and a second-order model with a zero and an integrator for the second input:

```
m = idproc({'P1','P2ZI'})
m = procest(data,m)
```

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

Disturbance Model Structure for Process Models

This section describes how to specify a noise model in the estimation procedures "How to Estimate Process Models Using the GUI" on page 3-27 and "How to Estimate Process Models at the Command Line" on page 3-32.

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

$$y = Gu + He$$

where e is white noise.

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

In the GUI. To specify whether to include or exclude a noise model in the Process Models dialog box, select one of the following options from the **Disturbance Model** list:

- `None` — The algorithm does not estimate a noise model (C=D=1). This option also sets **Focus** to `Simulation`.
- `Order 1` — Estimates a noise model as a continuous-time, first-order ARMA model.
- `Order 2` — Estimates a noise model as a continuous-time, second-order ARMA model.

At the command line. Specify the disturbance model using the `procestOptions` option set. For example, use this command to estimate a first-order transfer function and a first-order noise model:

```
opt = procestOptions;
opt.DisturbanceModel = 'arma1';
model = procest(data, 'P1D', opt);
```

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

Assigning Estimation Weightings

You can specify how the estimation algorithm weighs the fit at various frequencies. This information supports the estimation procedures "How to Estimate Process Models Using the GUI" on page 3-27 and "How to Estimate Process Models at the Command Line" on page 3-32.

In the System Identification Tool GUI. Set **Focus** to one of the following options:

- `Prediction` — Uses the inverse of the noise model *H* to weigh the relative importance of how closely to fit the data in various frequency ranges. Corresponds to minimizing one-step-ahead prediction, which typically favors the fit over a short time interval. Optimized for output prediction applications.
- `Simulation` — Uses the input spectrum to weigh the relative importance of the fit in a specific frequency range. Does not use the noise model to weigh the relative importance of how closely to fit the data in various frequency ranges. Optimized for output simulation applications.
- `Stability` — Behaves the same way as the `Prediction` option, but also forces the model to be stable. For more information about model stability, see "Unstable Models" on page 8-94.
- `Filter` — Specify a custom filter to open the Estimation Focus dialog box, where you can enter a filter, as described in "Simple Passband Filter" on page 2-126 or "Defining a Custom Filter" on page 2-127. This prefiltering applies only for estimating the dynamics from input to output. The disturbance model is determined from the estimation data.

At the command line. Specify the focus using the `procestOptions` option set. For example, use this command to optimize the fit for simulation and estimate a disturbance model:

```
opt = procestOptions('DisturbanceModel','arma2', 'Focus','simulation');
model = procest(data, 'P1D', opt);
```

Specifying Initial Conditions for Iterative Estimation Algorithms

You can optionally specify how the iterative algorithm treats initial conditions for estimation of model parameters. This information supports the estimation

procedures "How to Estimate Process Models Using the GUI" on page 3-27 and "How to Estimate Process Models at the Command Line" on page 3-32.

In the System Identification Tool GUI. Set **Initial condition** to one of the following options:

- `Zero` — Sets all initial states to zero.
- `Estimate` — Treats the initial states as an unknown vector of parameters and estimates these states from the data.
- `Backcast` — Estimates initial states using a backward filtering method (least-squares fit).
- `U-level est` — Estimates both the initial conditions and input offset levels. For multiple inputs, the input level for each input is estimated individually. Use if you included an integrator in the transfer function.
- `Auto` — Automatically chooses one of the preceding options based on the estimation data. If the initial conditions have negligible effect on the prediction errors, they are taken to be zero to optimize algorithm performance.

At the command line. Specify the initial conditions using the `InitialCondition` model estimation option, configured using the `procestOptions` command. For example, use this command to estimate a first-order transfer function and set the initial states to zero:

```
opt = procestOptions('InitialCondition','zero');
model = procest(data, 'P1D', opt)
```

Identifying Input-Output Polynomial Models

In this section...

What Are Polynomial Models?

Polynomial Model Structure

A polynomial model uses a generalized notion of transfer functions to express the relationship between the input, $u(t)$, the output $y(t)$, and the noise $e(t)$ using the equation:

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

The variables A, B, C, D, and F are polynomials expressed in the time-shift operator `q^-1`. u_i is the ith input, nu is the total number of inputs, and nk_i is the ith input delay that characterizes the transport delay. The variance of the white noise $e(t)$ is assumed to be λ. For more information about the time-shift operator, see "Understanding the Time-Shift Operator q" on page 3-47.

In practice, not all the polynomials are simultaneously active. Often, simpler forms, such as ARX, ARMAX, Output-Error, and Box-Jenkins are employed. You also have the option of introducing an integrator in the noise source so that the general model takes the form:

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

For more information, see "Different Configurations of Polynomial Models" on page 3-47.

You can estimate polynomial models using time or frequency domain data.

For estimation, you must specify the *model order* as a set of integers that represent the number of coefficients for each polynomial you include in your selected structure—na for A, nb for B, nc for C, nd for D, and nf for F. You must also specify the number of samples nk corresponding to the input delay—*dead time*—given by the number of samples before the output responds to the input.

The number of coefficients in denominator polynomials is equal to the number of poles, and the number of coefficients in the numerator polynomials is equal to the number of zeros plus 1. When the dynamics from $u(t)$ to $y(t)$ contain a delay of nk samples, then the first nk coefficients of B are zero.

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

Understanding the Time-Shift Operator q

The general polynomial equation is written in terms of the time-shift operator q^{-1}. To understand this time-shift operator, consider the following discrete-time difference equation:

$$y(t) + a_1 y(t-T) + a_2 y(t-2T) = b_1 u(t-T) + b_2 u(t-2T)$$

where *y(t)* is the output, *u(t)* is the input, and T is the sampling interval. q^{-1} is a time-shift operator that compactly represents such difference equations using $q^{-1}u(t) = u(t-T)$:

$$y(t) + a_1 q^{-1} y(t) + a_2 q^{-2} y(t) = b_1 q^{-1} u(t) + b_2 q^{-2} u(t)$$

or

$$A(q)y(t) = B(q)u(t)$$

In this case, $A(q) = 1 + a_1 q^{-1} + a_2 q^{-2}$ and $B(q) = b_1 q^{-1} + b_2 q^{-2}$.

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

Different Configurations of Polynomial Models

These model structures are subsets of the following general polynomial equation:

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

The model structures differ by how many of these polynomials are included in the structure. Thus, different model structures provide varying levels of flexibility for modeling the dynamics and noise characteristics.

The following table summarizes common linear polynomial model structures supported by the System Identification Toolbox product. If you have a specific structure in mind for your application, you can decide whether the dynamics and the noise have common or different poles. *A(q)* corresponds to poles that are common for the dynamic model and the noise model. Using common poles for dynamics and noise is useful when the disturbances enter the system at the input. F_i determines the poles unique to the system dynamics, and *D* determines the poles unique to the disturbances.

Model Structure	Equation	Description
ARX	$A(q)y(t) = \sum_{i=1}^{nu} B_i(q)u_i(t - nk_i) + e(t)$	The noise model is $\frac{1}{A}$ and the noise is coupled to the dynamics model. ARX does not let you model noise and dynamics independently. Estimate an ARX model to obtain a simple model at good signal-to-noise
ARIX	$Ay = Bu + \frac{1}{1 - q^{-1}}e$	Extends the ARX structure by including an integrator in the noise source, *e*(*t*). This is useful in cases where the disturbance is not stationary.

Model Structure	Equation	Description
ARMAX	$A(q)y(t) = \sum_{i=1}^{nu} B_i(q)u_i\left(t - nk_i\right) + C(q)e(t)$	Extends the ARX structure by providing more flexibility for modeling noise using the C parameters (a moving average of white noise). Use ARMAX when the dominating disturbances enter at the input. Such disturbances are called *load disturbances*.
ARIMAX	$Ay = Bu + C\frac{1}{1 - q^{-1}}e$	Extends theARMAX structure by including an integrator in the noise source, $e(t)$. This is useful in cases where the disturbance is not stationary.
Box-Jenkins (BJ)	$y(t) = \sum_{i=1}^{nu} \frac{B_i(q)}{F_i(q)}u_i\left(t - nk_i\right) + \frac{C(q)}{D(q)}e(t)$	Provides completely independent parameterization for the dynamics and the noise using rational polynomial functions. Use BJ models when the noise does not enter at the input, but is primary a measurement disturbance, This structure

Model Structure	Equation	Description
		provides additional flexibility for modeling noise.
Output-Error (OE)	$y(t) = \sum_{i=1}^{nu} \frac{B_i(q)}{F_i(q)} u_i(t - nk_i) + e(t)$	Use when you want to parameterize dynamics, but do not want to estimate a noise model. **Note** In this case, the noise models is $H = 1$ in the general equation and the white noise source *e(t)* affects only the output.

The polynomial models can contain one or more outputs and zero or more inputs.

The System Identification Tool GUI supports direct estimation of ARX, ARMAX, OE and BJ models. You can add a noise integrator to the ARX, ARMAX and BJ forms. However, you can use `polyest` to estimate all five polynomial or any subset of polynomials in the general equation. For more information about working with pem, see "Using polyest to Estimate Polynomial Models" on page 3-66.

Continuous-Time Representation of Polynomial Models

In continuous time, the general frequency-domain equation is written in terms of the Laplace transform variable s, which corresponds to a differentiation operation:

$$A(s)Y(s) = \frac{B(s)}{F(s)}U(s) + \frac{C(s)}{D(s)}E(s)$$

In the continuous-time case, the underlying time-domain model is a differential equation and the model order integers represent the number of estimated numerator and denominator coefficients. For example, n_a=3 and n_b=2 correspond to the following model:

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

You can only estimate continuous-time polynomial models directly using continuous-time frequency-domain data. In this case, you must set the `Ts` data property to 0 to indicate that you have continuous-time frequency-domain data, and use the `oe` command to estimate an Output-Error polynomial model. Continuous-time models of other structures such as ARMAX or BJ cannot be estimated. You can obtain those forms only by direct construction (using `idpoly`), conversion from other model types, or by converting a discrete-time model into continuous-time (`d2c`). Note that the OE form represents a transfer function expressed as a ratio of numerator (B) and denominator (F) polynomials. For such forms consider using the transfer function models, represented by `idtf` models. You can estimate transfer function models using both time and frequency domain data. In addition to the numerator and denominator polynomials, you can also estimate transport delays. See `idtf` and `tfest` for more information.

Multi-Output Polynomial Models

You can create multi-output polynomial models by using the `idpoly` command or estimate them using `ar`, `arx`, `bj`, `oe`, `armax`, and `polyest`. In the GUI, you can estimate such models by choosing a multi-output data set and setting the orders appropriately in the **Polynomial Models** dialog box. For more details on the orders of multi-output models, see "Polynomial Sizes and Orders of Multi-Output Polynomial Models" on page 3-68.

Data Supported by Polynomial Models

Types of Supported Data

You can estimate linear, black-box polynomial models from data with the following characteristics:

- Time- or frequency-domain data (`iddata` or `idfrd` data objects).

  Note For frequency-domain data, you can only estimate ARX and OE models.

  To estimate polynomial models for time-series data, see "Time-Series Model Identification".
- Real data or complex data in any domain.
- Single-output and multiple-output.

You must import your data into the MATLAB workspace, as described in "Data Preparation".

Designating Data for Estimating Continuous-Time Models

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

For continuous-time frequency-domain data, you can estimate directly only Output-Error (OE) continuous-time models. Other structures include noise models, which is not supported for frequency-domain data.

Tip To denote continuous-time frequency-domain data, set the data sampling interval to 0. You can set the sampling interval when you import data into the GUI or set the `Ts` property of the data object at the command line.

Designating Data for Estimating Discrete-Time Models

You can estimate arbitrary-order, linear state-space models for both time- or frequency-domain data.

Set the data property `Ts` to:

- `0`, for frequency response data that is measured directly from an experiment.
- Equal to the `Ts` of the original data, for frequency response data obtained by transforming time-domain `iddata` (using `spa` and `etfe`).

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

Preliminary Step – Estimating Model Orders and Input Delays

Why Estimate Model Orders and Delays?

To estimate polynomial models, you must provide input delays and model orders. If you already have insight into the physics of your system, you can specify the number of poles and zeros.

In most cases, you do not know the model orders in advance. To get initial model orders and delays for your system, you can estimate several ARX models with a range of orders and delays and compare the performance of these models. You choose the model orders that correspond to the best model performance and use these orders as an initial guess for further modeling.

Because this estimation procedure uses the ARX model structure, which includes the A and B polynomials, you only get estimates for the na, nb, and nk parameters. However, you can use these results as initial guesses for the corresponding polynomial orders and input delays in other model structures, such as ARMAX, OE, and BJ.

If the estimated nk is too small, the leading nb coefficients are much smaller than their standard deviations. Conversely, if the estimated nk is too large, there is a significant correlation between the residuals and the input for lags that correspond to the missing B terms. For information about residual analysis plots, see "Residual Analysis" on page 8-24.

Estimating Orders and Delays in the GUI

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

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

1. In the System Identification Tool GUI, select **Estimate > Polynomial and State Space Models** to open the Polynomials and State Space Models dialog box.

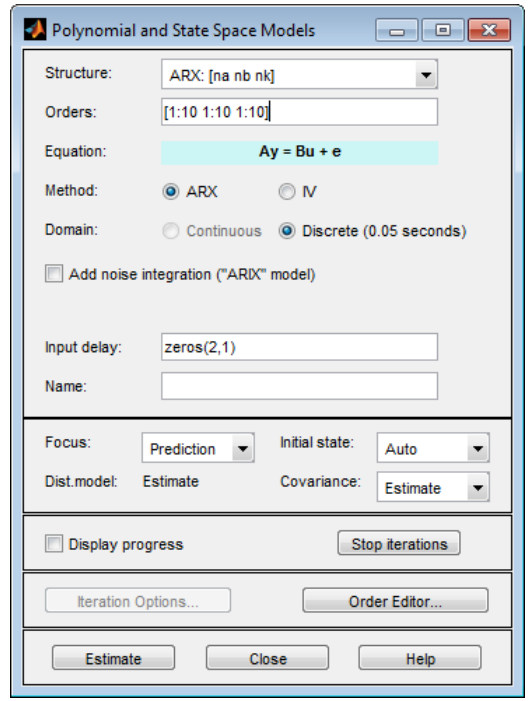
   The ARX model is already selected by default in the **Structure** list.

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

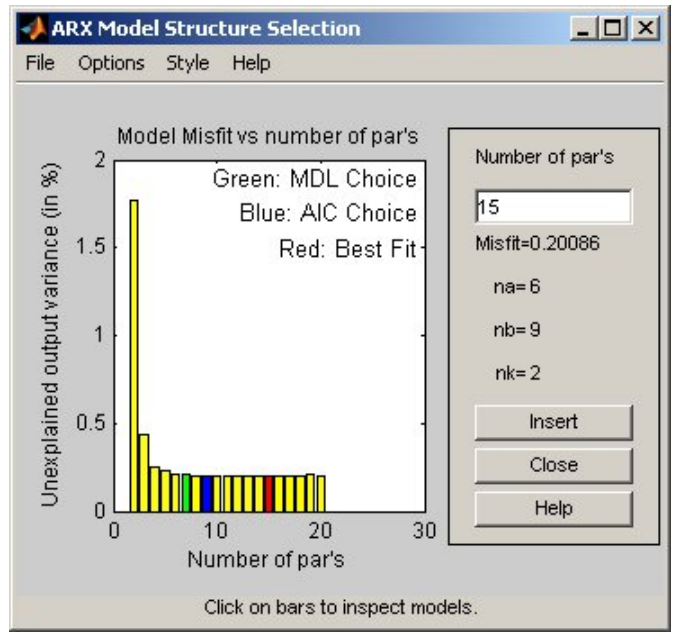
2 Edit the **Orders** field to specify a range of poles, zeros, and delays. For example, enter the following values for *na*, *nb*, and *nk*:

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

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



3 Click **Estimate** to open the ARX Model Structure Selection window, which displays the model performance for each combination of model parameters. The following figure shows an example plot.



4 Select a rectangle that represents the optimum parameter combination and click **Insert** to estimates a model with these parameters. For information about using this plot, see "Selecting Model Orders from the Best ARX Structure" on page 3-59.

This action adds a new model to the Model Board in the System Identification Tool GUI. The default name of the parametric model contains the model type and the number of poles, zeros, and delays. For example, `arx692` is an ARX model with n_a=6, n_b=9, and a delay of two samples.

5 Click **Close** to close the ARX Model Structure Selection window.

Note You cannot estimate model orders when using multi-output data.

After estimating model orders and delays, use these values as initial guesses for estimating other model structures, as described in "How to Estimate Polynomial Models in the GUI" on page 3-61.

Estimating Model Orders at the Command Line

You can estimate model orders using the `struc`, `arxstruc`, and `selstruc` commands in combination.

If you are working with a multiple-output system, you must use the `struc`, `arxstruc`, and `selstruc` commands one output at a time. You must subreference the correct output channel in your estimation and validation data sets.

For each estimation, you use two independent data sets—an estimation data set and a validation data set. These independent data set can be from different experiments, or data subsets from a single experiment. For more information about subreferencing data, see "Select Data Channels, I/O Data and Experiments in iddata Objects" on page 2-63 and "Select I/O Channels and Data in idfrd Objects" on page 2-79.

For an example of estimating model orders for a multiple-input system, see "Estimating Delays in the Multiple-Input System" in *System Identification Toolbox Getting Started Guide*.

struc. The `struc` command creates a matrix of possible model-order combinations for a specified range of n_a, n_b, and n_k values.

For example, the following command defines the range of model orders and delays `na=2:5`, `nb=1:5`, and `nk=1:5`:

```
NN = struc(2:5,1:5,1:5))
```

arxstruc. The `arxstruc` command takes the output from `struc`, estimates an ARX model for each model order, and compares the model output to the measured output. `arxstruc` returns the *loss* for each model, which is the normalized sum of squared prediction errors.

For example, the following command uses the range of specified orders `NN` to compute the loss function for single-input/single-output estimation data `data_e` and validation data `data_v`:

```
V = arxstruc(data_e,data_v,NN)
```

Each row in `NN` corresponds to one set of orders:

```
[na nb nk]
```

selstruc. The `selstruc` command takes the output from `arxstruc` and opens the ARX Model Structure Selection window to guide your choice of the model order with the best performance.

For example, to open the ARX Model Structure Selection window and interactively choose the optimum parameter combination, use the following command:

```
selstruc(V)
```

For more information about working with the ARX Model Structure Selection window, see “Selecting Model Orders from the Best ARX Structure” on page 3-59.

To find the structure that minimizes Akaike’s Information Criterion, use the following command:

```
nn = selstruc(V,'AIC')
```

where `nn` contains the corresponding `na`, `nb`, and `nk` orders.

Similarly, to find the structure that minimizes the Rissanen’s Minimum Description Length (MDL), use the following command:

```
nn = selstruc(V,'MDL')
```

To select the structure with the smallest loss function, use the following command:

```
nn = selstruc(V,0)
```

After estimating model orders and delays, use these values as initial guesses for estimating other model structures, as described in "Using polyest to Estimate Polynomial Models" on page 3-66.

Estimating Delays at the Command Line

The `delayest` command estimates the time delay in a dynamic system by estimating a low-order, discrete-time ARX model and treating the delay as an unknown parameter.

By default, `delayest` assumes that n_a=n_b=2 and that there is a good signal-to-noise ratio, and uses this information to estimate n_k.

To estimate the delay for a data set `data`, type the following at the prompt:

```
delayest(data)
```

If your data has a single input, MATLAB computes a scalar value for the input delay—equal to the number of data samples. If your data has multiple inputs, MATLAB returns a vector, where each value is the delay for the corresponding input signal.

To compute the actual delay time, you must multiply the input delay by the sampling interval of the data.

You can also use the ARX Model Structure Selection window to estimate input delays and model order together, as described in "Estimating Model Orders at the Command Line" on page 3-57.

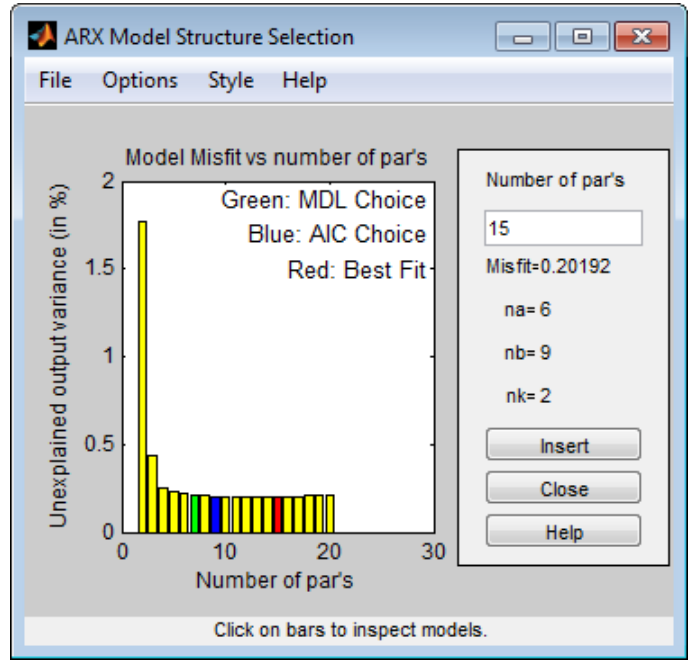
Selecting Model Orders from the Best ARX Structure

You generate the ARX Model Structure Selection window for your data to select the best-fit model.

For a procedure on generating this plot in the System Identification Tool GUI, see "Estimating Orders and Delays in the GUI" on page 3-54. To open this plot at the command line, see "Estimating Model Orders at the Command Line" on page 3-57.

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

You use this plot to select the best-fit model.



- The horizontal axis is the total number of parameters — $n_a + n_b$.
- The vertical axis, called **Unexplained output variance (in %)**, is the portion of the output not explained by the model—the ARX model prediction error for the number of parameters shown on the horizontal axis.

  The *prediction error* is the sum of the squares of the differences between the validation data output and the model one-step-ahead predicted output.
- n_k is the delay.

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

- Red — Best fit minimizes the sum of the squares of the difference between the validation data output and the model output. This rectangle indicates the overall best fit.

- Green — Best fit minimizes Rissanen MDL criterion.
- Blue — Best fit minimizes Akaike AIC criterion.

In the ARX Model Structure Selection window, click any bar to view the orders that give the best fit. The area on the right is dynamically updated to show the orders and delays that give the best fit.

For more information about the AIC criterion, see "Akaike's Criteria for Model Validation" on page 8-86.

How to Estimate Polynomial Models in the GUI

Prerequisites

Before you can perform this task, you must have:

- Imported data into the System Identification Tool GUI. See "Importing Time-Domain Data into the GUI" on page 2-18. For supported data formats, see "Data Supported by Polynomial Models" on page 3-52.
- Performed any required data preprocessing operations. To improve the accuracy of your model, you should detrend your data. Removing offsets and trends is especially important for Output-Error (OE) models and has less impact on the accuracy of models that include a flexible noise model structure, such as ARMAX and Box-Jenkins. See "Ways to Prepare Data for System Identification" on page 2-6.
- Select a model structure, model orders, and delays. For a list of available structures, see "What Are Polynomial Models?" on page 3-45 For more information about how to estimate model orders and delays, see "Estimating Orders and Delays in the GUI" on page 3-54. For multiple-output models, you must specify order matrices in the MATLAB workspace, as described in "Polynomial Sizes and Orders of Multi-Output Polynomial Models" on page 3-68.

1 In the System Identification Tool GUI, select **Estimate > Polynomial** to open the Polynomial and State Space 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 Polynomial and State Space Models dialog box to correspond with this model structure. For information about each model structure, see "What Are Polynomial Models?" on page 3-45.

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

---

3 In the **Orders** field, specify the model orders and delays, as follows:

- **For single-output polynomial models.** Enter the model orders and delays according to the sequence displayed in the **Structure** field. For multiple-input models, specify `nb` and `nk` as row vectors with as many elements as there are inputs. If you are estimating BJ and OE models, you must also specify `nf` as a vector.

  For example, for a three-input system, `nb` can be `[1 2 4]`, where each element corresponds to an input.

- **For multiple-output models.** Enter the model orders, as described in "Polynomial Sizes and Orders of Multi-Output Polynomial Models" on page 3-68.

Tip To enter model orders and delays using the Order Editor dialog box, click **Order Editor**.

---

4 (ARX models only) Select the estimation **Method** as **ARX** or **IV** (instrumental variable method). For information about the algorithms, see "Polynomial Model Estimation Algorithms" on page 3-73.

5. (ARX, ARMAX, and BJ models only) Check the **Add noise integration** check box to add an integrator to the noise source, *e*.

6. Specify the delay using the **Input delay** edit box. The value must be a vector of length equal to the number of input channels in the data. For discrete-time estimations (any estimation using data with nonzero sample-time), the delay must be expressed in the number of lags. These delays are separate from the "in-model" delays specified by the `nk` order in the **Orders** edit box.

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

8. In the **Focus** list, select how to weigh the relative importance of the fit at different frequencies. For more information about each option, see "Assigning Estimation Weightings" on page 3-72.

9. In the **Initial state** list, specify how you want the algorithm to treat initial conditions. For more information about the available options, see "Specifying Initial Conditions for Iterative Estimation Algorithms" on page 3-43.

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

---

10. In the **Covariance** list, select `Estimate` if you want the algorithm to compute parameter uncertainties. Effects of such uncertainties are displayed on plots as model confidence regions.

    To omit estimating uncertainty, select `None`. Skipping uncertainty computation for large, multiple-output models might reduce computation time.

11. (ARMAX, OE, and BJ models only) To view the estimation progress in the MATLAB Command Window, select the **Display progress** check box. This launches a progress viewer window in which estimation progress is reported.

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

13 (Prediction-error method only) To stop the search and save the results after the current iteration has been completed, click **Stop Iterations**. To continue iterations from the current model, click the **Continue iter** button to assign current parameter values as initial guesses for the next search.

Next Steps

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

Tip For ARX and OE models, you can use the exported model for initializing a nonlinear estimation at the command line. This initialization may improve the fit of the model. See "Using Linear Model for Nonlinear ARX Estimation" on page 4-28, and "Using Linear Model for Hammerstein-Wiener Estimation" on page 4-63.

How to Estimate Polynomial Models at the Command Line

Prerequisites

Before you can perform this task, you must have

- Input-output data as an `iddata` object or frequency response data as an `frd` or `idfrd` object. See "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-55. For supported data formats, see "Data Supported by Polynomial Models" on page 3-52.

- Performed any required data preprocessing operations. To improve the accuracy of results when using time domain data, you can detrend the data or specify the input/output offset levels as estimation options. Removing offsets and trends is especially important for Output-Error (OE) models and has less impact on the accuracy of models that include a flexible noise model structure, such as ARMAX and Box-Jenkins. See "Ways to Prepare Data for System Identification" on page 2-6.
- Select a model structure, model orders, and delays. For a list of available structures, see "What Are Polynomial Models?" on page 3-45 For more information about how to estimate model orders and delays, see "Estimating Model Orders at the Command Line" on page 3-57 and "Estimating Delays at the Command Line" on page 3-59. For multiple-output models, you must specify order matrices in the MATLAB workspace, as described in "Polynomial Sizes and Orders of Multi-Output Polynomial Models" on page 3-68.

Using arx and iv4 to Estimate ARX Models

You can estimate single-output and multiple-output ARX models using the `arx` and `iv4` commands. For information about the algorithms, see "Polynomial Model Estimation Algorithms" on page 3-73.

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

```
% Using ARX method
m = arx(data,[na nb nk],opt)
% Using IV method
m = iv4(data,[na nb nk],opt)
```

`data` is the estimation data and `[na nb nk]` specifies the model orders, as discussed in "What Are Polynomial Models?" on page 3-45.

The third input argument `opt` contains the options for configuring the estimation of the ARX model, such as handling of initial conditions and input offsets. You can create and configure the option set `opt` using the `arxOptions` and `iv4Options` commands. The three input arguments can also be followed by name and value pairs to specify optional model structure attributes such as `InputDelay`, `ioDelay`, and `IntegrateNoise`.

To get discrete-time models, use the time-domain data (`iddata` object).

Note Continuous-time polynomials of ARX structure are not supported.

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

You can use `pem` or `polyest` to refine parameter estimates of an existing polynomial model, as described in "Refining Linear Parametric Models" on page 3-130.

For detailed information about these commands, see the corresponding reference page.

Tip You can use the estimated ARX model for initializing a nonlinear estimation at the command line, which improves the fit of the model. See "Using Linear Model for Nonlinear ARX Estimation" on page 4-28.

Using polyest to Estimate Polynomial Models

You can estimate any polynomial model using the iterative prediction-error estimation method `polyest`. For Gaussian disturbances of unknown variance, this method gives the maximum likelihood estimate. The resulting models are stored as `idpoly` model objects.

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

```
m = polyest(data, [na nb nc nd nf nk], opt,Name,Value)
```

where `data` is the estimation data. `na`, `nb`, `nc`, `nd`, `nf` are integers that specify the model orders, and `nk` specifies the input delays for each input.For more information about model orders, see "What Are Polynomial Models?" on page 3-45.

Tip You do not need to construct the model object using `idpoly` before estimation.

If you want to estimate the coefficients of all five polynomials, *A*, *B*, *C*, *D*, and *F*, you must specify an integer order for each polynomial. However, if you want to specify an ARMAX model for example, which includes only the *A*, *B*, and *C* polynomials, you must set `nd` and `nf` to zero matrices of the appropriate size. For some simpler configurations, there are dedicated estimation commands such as `arx`, `armax`, `bj`, and `oe`, which deliver the required model by using just the required orders. For example, `oe(data, [nb nf nk],opt)` estimates an output-error structure polynomial model.

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

In addition to the polynomial models listed in "What Are Polynomial Models?" on page 3-45, you can use `polyest` to model the ARARX structure—called the *generalized least-squares model*—by setting `nc=nf=0`. You can also model the ARARMAX structure—called the *extended matrix model*—by setting `nf=0`.

The third input argument, `opt`, contains the options for configuring the estimation of the polynomial model, such as handling of initial conditions, input offsets and search algorithm. You can create and configure the option set `opt` using the `polyestOptions` command. The three input arguments can also be followed by name and value pairs to specify optional model structure attributes such as `InputDelay`, `ioDelay`, and `IntegrateNoise`.

For ARMAX, Box-Jenkins, and Output-Error models—which can only be estimated using the iterative prediction-error method—use the `armax`, `bj`, and `oe` estimation commands, respectively. These commands are versions of `polyest` with simplified syntax for these specific model structures, as follows:

```
m = armax(Data,[na nb nc nk])
m = oe(Data,[nb nf nk])
m = bj(Data,[nb nc nd nf nk])
```

Similar to `polyest`, you can specify as input arguments the option set configured using commands `armaxOptions`, `oeOptions`, and `bjOptions` for the estimators `armax`, `oe`, and `bj` respectively. You can also use name and value pairs to configure additional model structure attributes.

Tip If your data is sampled fast, it might help to apply a lowpass filter to the data before estimating the model, or specify a frequency range for the `Focus` property during estimation. For example, to model only data in the frequency range 0-10 rad/s, use the `Focus` property, as follows:

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

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

You can use `pem` or `polyest` to refine parameter estimates of an existing polynomial model (of any configuration), as described in "Refining Linear Parametric Models" on page 3-130.

For more information, see `polyest`, `pem` and `idpoly`.

Polynomial Sizes and Orders of Multi-Output Polynomial Models

For a model with *Ny* (`Ny > 1`) outputs and *Nu* inputs, the polynomials *A*, *B*, *C*, *D*, and *F* are specified as cell arrays of row vectors. Each entry in the cell array contains the coefficients of a particular polynomial that relates input, output, and noise values. *Orders* are matrices of integers used as input arguments to the estimation commands.

Polynomial	Dimension	Relation Described	Orders
`A`	N_y-by-N_y array of row vectors	`A{i,j}` contains coefficients of relation between output y_i and output y_j	`na`: N_y-by-N_y matrix such that each entry contains the degree of the corresponding A polynomial.
`B`	N_y-by-N_u array of row vectors	`B{i,j}` contain coefficients of relations between output y_i and input u_j	`nk`: N_y-by-N_u matrix such that each entry contains the number of leading fixed zeros of the corresponding B polynomial (input delay). `nb`: N_y-by-N_u matrix such `nb(i,j)`

Polynomial	Dimension	Relation Described	Orders
			= length(B{i,j})-nk(i,j).
C,D	N_y-by-1 array of row vectors	C{i} and D{i} contain coefficients of relations between output y_i and noise e_i	nc and nd are N_y-by-1 matrices such that each entry contains the degree of the corresponding C and D polynomial, respectively.
F	N_y-by-N_u array of row vectors	F{i,j} contains coefficients of relations between output y_i and input u_j	nf: N_y-by-N_u matrix such that each entry contains the degree of the corresponding F polynomial.

For more information, see `idpoly`.

For example, consider the ARMAX set of equations for a 2 output, 1 input model:

$$y_1(t) + 0.5\, y_1(t\text{-}1) + 0.9\, y_2(t\text{-}1) + 0.1\, y_2(t\text{-}2) = u(t) + 5\, u(t\text{-}1) + 2\, u(t\text{-}2) + e_1(t) + 0.01\, e_1$$
$$y_2(t) + 0.05\, y_2(t\text{-}1) + 0.3\, y_2(t\text{-}2) = 10\, u(t\text{-}2) + e_2(t) + 0.1\, e_2(t\text{-}1) + 0.02\, e_2(t\text{-}2)$$

y_1 and y_2 represent the two outputs and u represents the input variable. e_1 and e_2 represent the white noise disturbances on the outputs, y_1 and y_2, respectively. To represent these equations as an ARMAX form polynomial using `idpoly`, configure the A, B, and C polynomials as follows:

```
A = cell(2,2);
A{1,1} = [1 0.5]; A{1,2} = [0 0.9 0.1];
A{2,1} = [0];  A{2,2} = [1 0.05 0.3];

B = cell(2,1);
B{1,1} = [1 5 2]; B{2,1}  = [0 0 10];

C = cell(2,1);
C{1} = [1 0.01]; C{2} = [1 0.1 0.02];

model = idpoly(A,B,C)
```

`model` is a discrete-time ARMAX model with unspecified sample-time.

When estimating such models, you need to specify the orders of these polynomials as input arguments.

In the System Identification Tool GUI. You can enter the matrices directly in the **Orders** field.

At the command line. Define variables that store the model order matrices and specify these variables in the model-estimation command.

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

Assigning Estimation Weightings

You can specify how the estimation algorithm weighs the fit at various frequencies. This information supports the estimation procedures "How to Estimate Polynomial Models in the GUI" on page 3-61 and "Using polyest to Estimate Polynomial Models" on page 3-66.

In the System Identification Tool GUI. Set **Focus** to one of the following options:

- `Prediction` — Uses the inverse of the noise model H to weigh the relative importance of how closely to fit the data in various frequency ranges. Corresponds to minimizing one-step-ahead prediction, which typically favors the fit over a short time interval. Optimized for output prediction applications.
- `Simulation` — Uses the input spectrum to weigh the relative importance of the fit in a specific frequency range. Does not use the noise model to weigh the relative importance of how closely to fit the data in various frequency ranges. Optimized for output simulation applications.
- `Stability` — Estimates the best stable model. For more information about model stability, see "Unstable Models" on page 8-94.
- `Filter` — Specify a custom filter to open the Estimation Focus dialog box, where you can enter a filter, as described in "Simple Passband Filter" on page 2-126 or "Defining a Custom Filter" on page 2-127. This prefiltering applies only for estimating the dynamics from input to output. The disturbance model is determined from the unfiltered estimation data.

At the command line. Specify the focus as an estimation option (created using `polyestOptions`, `oeOptions` etc.) 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:

```
opt = arxOptions('Focus', 'simulation');
m = arx(data,[2 2 3],opt)
```

This `Focus` setting might produce more accurate simulation results, provided the orders picked are optimal for the given data..

Specifying Initial States for Iterative Estimation Algorithms

When you use the `pem` or `polyest` to estimate ARMAX, Box-Jenkins (BJ), Output-Error (OE), you must specify how the algorithm treats initial conditions.

This information supports the estimation procedures "How to Estimate Polynomial Models in the GUI" on page 3-61 and "Using polyest to Estimate Polynomial Models" on page 3-66.

In the System Identification Tool GUI. For ARMAX, OE, and BJ models, set **Initial state** to one of the following options:

- `Auto` — Automatically chooses `Zero`, `Estimate`, or `Backcast` based on the estimation data. If initial states have negligible effect on the prediction errors, the initial states are set to zero to optimize algorithm performance.
- `Zero` — Sets all initial states to zero.
- `Estimate` — Treats the initial states as an unknown vector of parameters and estimates these states from the data.
- `Backcast` — Estimates initial states using a smoothing filter.

At the command line. Specify the initial conditions as an estimation option. Use `polyestOptions` to configure options for the `polyest` command, `armaxOptions` for the `armax` command etc. Set the `InitialCondition` option to the desired value in the option set. For example, use this command to estimate an ARMAX model and set the initial states to zero:

```
opt = armaxOptions('InitialCondition','zero')
m = armax(data,[2 2 2 3],opt)
```

For a complete list of values for the `InitialCondition` estimation option, see the `armaxOptions` reference page.

Polynomial Model Estimation Algorithms

For linear ARX and AR models, you can choose between the ARX and IV algorithms. *ARX* implements the least-squares estimation method that uses QR-factorization for overdetermined linear equations. *IV* is the *instrument*

variable method. For more information about IV, see the section on variance-optimal instruments in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

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

Note AR models apply to time-series data, which has no input. For more information, see "Time-Series Model Identification". For more information about working with AR and ARX models, see "Identifying Input-Output Polynomial Models" on page 3-45.

Estimate Models Using armax

This example shows how to estimate a linear, polynomial model with an ARMAX structure for a three-input and single-output (MISO) system using the iterative estimation method `armax`. For a summary of all available estimation commands in the toolbox, see "Model Estimation Commands" on page 1-40.

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

   ```
   load iddata8
   ```

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

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

Typically you try different model orders and compare results, ultimately choosing the simplest model that best describes the system dynamics. The following command specifies the estimation data set, `z8`, and the orders of the *A*, *B*, and *C* polynomials as `na`, `nb`, and `nc`, respectively. `nk` of `[0 0 0]` specifies that there is no input delay for all three input channels.

```
opt = armaxOptions;
opt.Focus = 'simulation';
opt.SearchOption.MaxIter = 50;
opt.SearchOption.Tolerance = 1e-5;
na = 4;
nb = [3 2 3];
nc = 4;
nk = [0 0 0];
m_armax = armax(z8, [na nb nc nk], opt)
```

`Focus`, `Tolerance`, and `MaxIter` are estimation options that configure the estimation objective function and the attributes of the search algorithm. The `Focus` option specifies whether the model is optimized for simulation or prediction applications. The `Tolerance` and `MaxIter` search options specify when to stop estimation. For more information about these properties, see the `armaxOptions` reference page.

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

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

```
m_armax
```

MATLAB returns the following information about this model object:

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

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

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

  B3(z) = -0.06739 + 0.06828 z^-1 + 0.5509 z^-2

  C(z) = 1 - 0.06096 z^-1 - 0.1296 z^-2 + 0.02489 z^-3 - 0.04699 z^-4

Sample time: 1 seconds

Parameterization:
Polynomial orders:   na=4   nb=[3 2 3]   nc=4   nk=[0 0 0]
   Number of free coefficients: 16
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using ARMAX on time domain data "z8".
Fit to estimation data: 80.86% (simulation focus)
FPE: 1.056, MSE: 0.9888
```

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

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

---

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

```
get(m_armax)
```

MATLAB returns the following information:

```
                a: [1 -1.2836 0.3048 0.2648 -0.0571]
                b: {1x3 cell}
                c: [1 -0.0610 -0.1296 0.0249 -0.0470]
                d: 1
                f: {[1]  [1]  [1]}
         Variable: 'z^-1'
          ioDelay: [0 0 0]
    IntegrateNoise: 0
        Structure: [1x1 pmodel.polynomial]
     NoiseVariance: 0.9899
           Report: [1x1 idresults.polyest]
       InputDelay: [3x1 double]
      OutputDelay: 0
               Ts: 1
         TimeUnit: 'seconds'
        InputName: {3x1 cell}
        InputUnit: {3x1 cell}
       InputGroup: [1x1 struct]
       OutputName: {'y1'}
       OutputUnit: {''}
      OutputGroup: [1x1 struct]
             Name: ''
            Notes: {}
         UserData: []
```

5 The `Report` model property contains detailed information on the estimation results. To view the properties and values inside `Report`, use dot notation. For example:

```
m_armax.Report
```

This action displays the contents of estimation report such as model quality measures (`Fit`), search termination criterion (`Termination`), and a record of estimation data (`DataUsed`) and options (`OptionsUsed`).

```
             Status: 'Estimated using POLYEST with Focus = "simulation"'
             Method: 'ARMAX'
    InitialCondition: 'zero'
                Fit: [1x1 struct]
         Parameters: [1x1 struct]
        OptionsUsed: [1x1 idoptions.polyest]
           RandState: [1x1 struct]
           DataUsed: [1x1 struct]
        Termination: [1x1 struct]
```

Identifying State-Space Models

In this section...

What Are State-Space Models?

Definition of State-Space Models

State-space models are models that use state variables to describe a system by a set of first-order differential or difference equations, rather than by one or more *n*th-order differential or difference equations. State variables *x(t)* can be reconstructed from the measured input-output data, but are not themselves measured during an experiment.

The state-space model structure is a good choice for quick estimation because it requires only one user input, the *model order*, n.

The *model order* is an integer equal to the dimension of *x(t)* and relates to, but is not necessarily equal to, the number of delayed inputs and outputs used in the corresponding linear difference equation.

Continuous-Time Representation

In continuous-time, the state-space description has the following form:

$$\dot{x}(t) = Fx(t) + Gu(t) + \tilde{K}w(t)$$
$$y(t) = Hx(t) + Du(t) + w(t)$$
$$x(0) = x0$$

It is often easier to define a parameterized state-space model in continuous time because physical laws are most often described in terms of differential equations. In this case, the matrices F, G, H, and D contain elements with physical significance—for example, material constants. *x0* specifies the initial states. You can estimate continuous-time state-space model using both time- and frequency-domain data.

Note $\tilde{K} = 0$ gives the state-space representation of an Output-Error model. For more information about Output-Error models, see "What Are Polynomial Models?" on page 3-45.

Discrete-Time Representation

Discrete-time state-space models provide the same type of linear difference relationship between the inputs and the outputs as the linear ARX model, but are rearranged such that there is only one delay in the expressions. The discrete-time state-space model structure is often written in the *innovations form* that describes noise:

$$x(kT+T) = Ax(kT) + Bu(kT) + Ke(kT)$$
$$y(kT) = Cx(kT) + Du(kT) + e(kT)$$
$$x(0) = x0$$

where T is the sampling interval, $u(kT)$ is the input at time instant kT, and $y(kT)$ is the output at time instant kT.

Note K=0 gives the state-space representation of an Output-Error model. For more information about Output-Error models, see "What Are Polynomial Models?" on page 3-45.

Relationship Between Continuous-Time and Discrete-Time State Matrices

The relationships between the discrete state-space matrices A, B, C, D, and K and the continuous-time state-space matrices F, G, H, D, and $\tilde{K}$ are given for piece-wise-constant input, as follows:

$$A = e^{FT}$$
$$B = \int_0^T e^{F\tau} G d\tau$$
$$C = H$$

These relationships assume that the input is piece-wise-constant over time intervals $kT \le t < (k+1)T$.

The exact relationship between K and $\tilde{K}$ is complicated. However, for short sampling intervals T, the following approximation works well:

$$K = \int_0^T e^{F\tau}\tilde{K}d\tau$$

State-Space Representation of Transfer Functions

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

$$y = Gu + He$$

G is a transfer function that takes the input u to the output y. H is a transfer function that describes the properties of the additive output noise model.

The discrete-time state-space representation is given by the following equation:

$$x(kT + T) = Ax(kT) + Bu(kT) + Ke(kT)$$
$$y(kT) = Cx(kT) + Du(kT) + e(kT)$$
$$x(0) = x0$$

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.

The relationships between the transfer functions and the discrete-time state-space matrices are given by the following equations:

$$G(q) = C(qI_{nx} - A)^{-1}B + D$$
$$H(q) = C(qI_{nx} - A)^{-1}K + I_{ny}$$

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. The state-space representation in the continuous-time case is similar.

Data Supported by State-Space Models

You can estimate linear state-space models from data with the following characteristics:

- Real data or complex data in any domain
- Single-output and multiple-output
- Time- or frequency-domain data

  To estimate state-space models for time-series data, see "Time-Series Model Identification".

You must first import your data into the MATLAB workspace, as described in "Data Preparation".

Supported State-Space Parameterizations

The System Identification Toolbox product supports the following parameterizations that indicate which parameters are estimated and which remain fixed at specific values:

- **Free parameterization** results in the estimation of all system matrix elements A, B, C, D, and K.
- **Canonical forms** of A, B, C, D, and K matrices.

  Canonical parameterization represents a state-space system in a reduced-parameter form where many entries of the A, B and C matrices are fixed to zeros and ones. The free parameters appear in only a few of the rows and columns in the system matrices.
- **Structured parameterization** lets you specify the fixed values of specific parameters and exclude these parameters from estimation. You choose which entries of the system matrices to estimate and which to treat as fixed.
- **Completely arbitrary mapping** of user-chosen parameters to state-space matrices. For more information, see "Estimating Linear Grey-Box Models" on page 5-6.

Estimate State-Space Model With Order Selection

- "Estimate Model With Selected Order in the GUI" on page 3-84

To estimate a state-space model, you must provide a value of its order, which represents the number of states. When you do not know the order, you can search and select an order using the following procedures.

Estimate Model With Selected Order in the GUI

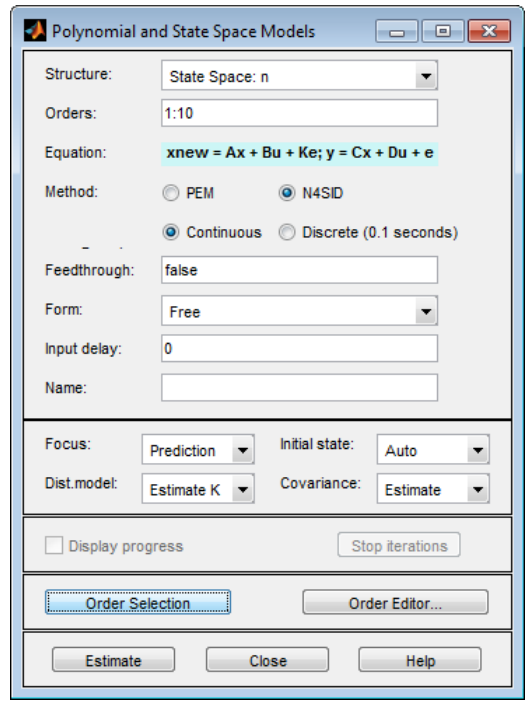
You must have already imported your data into the GUI, as described in "Importing Data into the GUI" on page 2-17.

To estimate model orders for a specific input delay:

1. In the System Identification Tool GUI, select **Estimate > State Space Models** to open the Polynomial and State Space Models dialog box.
2. In the **Structure** list, select `State Space:  n`.
3. Edit the **Orders** field to specify a range of orders for a specific delay. For example, enter the following values for n:

   ```
   1:10
   ```

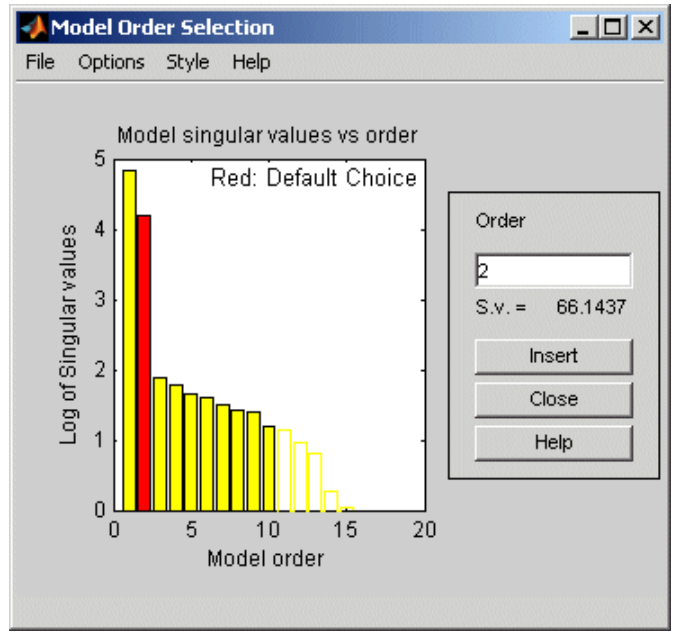
   Tip As a shortcut for entering `1:10`, click **Order Selection**.



4 Verify that the **Method** is set to **N4SID**.

5 Click **Estimate** to open the Model Order Selection window, which displays the relative measure of how much each state contributes to the

input-output behavior of the model (*log of singular values of the covariance matrix*). The following figure shows an example plot.



6 Select the rectangle that represents the cutoff for the states on the left that provide a significant contribution to the input-output behavior, and click **Insert** to estimate a model with this order. Red indicates the recommended choice. In the previous figure, states 1 and 2 provide the most significant contribution. The contributions to the right of state 2 drop significantly. For information about using the Model Order Selection window, see "Using the Model Order Selection Window" on page 3-88.

   This action adds a new model to the Model Board in the System Identification Tool GUI. The default name of the parametric model combines the string `n4s` and the selected model order.

7 Click **Close** to close the Model Order Selection window.

You can use this model as an initial guess for estimating other state-space models, as described in "How to Estimate State-Space Models in the GUI" on page 3-89.

Note You can specify additional attributes of the model structure when searching for best orders, such as input delays, the presence of feedthrough, the canonical form and the model sample time.

Estimate Model With Selected Order at the Command Line

You can estimate the state-space model with a selected order using the `n4sid` and `ssest` commands.

Use following syntax to specify the range of model orders to try for a specific input delay.

```
m = n4sid(data,n1:n2);
```

where `data` is the estimation data set, `n1` and `n2` specify the range of orders, and `nk` specifies the input delay.

This command opens the Model Order Selection window. For information about using this plot, see "Using the Model Order Selection Window" on page 3-88.

Alternatively, you can use the `ssest` command to open the Model Order Selection window, as follows:

```
m = ssest(data, nn)
```

where `nn = [n1,n2,...,nN]` specifies the vector or range of orders you want to try. `n4sid` estimates a model whose sample time matches that of `data`, hence a discrete-time model for time domain data. `ssest` estimates a continuous-time model by default. You can change the default setting by including the `Ts` name and value pair input arguments in the estimation command. For example, to estimate a discrete-time model of optimal order using `ssest`, assuming `Data.Ts>0`, type:

```
model = ssest(data,nn,'Ts',data.Ts);
```

To omit opening the Model Order Selection window and automatically select the best order, use the following syntax:

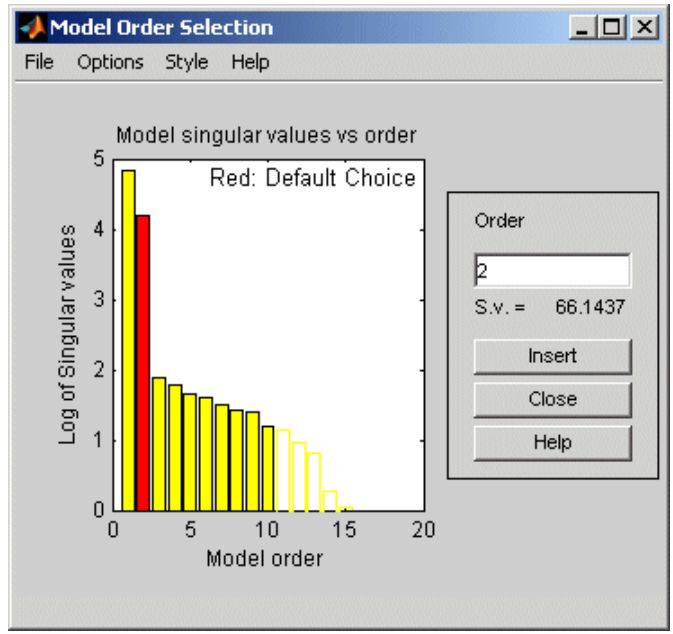
```
m = n4sid(data,'best')
```

Using the Model Order Selection Window

You can generate the Model Order Selection window for your data to select the number of states that provide the highest relative contribution to the input-output behavior of the model (*log of singular values of the covariance matrix*).

For a procedure on generating this plot in the System Identification Tool GUI, see "Estimate Model With Selected Order in the GUI" on page 3-84. To open this plot at the command line, see "Estimate Model With Selected Order at the Command Line" on page 3-87.

The following figure shows a sample Model Order Selection window.



The horizontal axis corresponds to the model order `n`. The vertical axis, called **Log of Singular values**, shows the singular values of a covariance matrix constructed from the observed data.

You use this plot to decide which states provide a significant relative contribution to the input-output behavior, and which states provide the smallest contribution. Based on this plot, select the rectangle that represents the cutoff for the states on the left that provide a significant contribution to the input-output behavior. The recommended choice is shown in red.

For example, in the previous figure, states 1 and 2 provide the most significant contribution. However, the contributions of the states to the right of state 2 drop significantly. This sharp decrease in the log of the singular values after n=2 indicates that using two states is sufficient to get an accurate model.

How to Estimate State-Space Models in the GUI

Prerequisites

Before you can perform this task, you must have

- Imported data into the System Identification Tool GUI. See "Importing Time-Domain Data into the GUI" on page 2-18. For supported data formats, see "Data Supported by State-Space Models" on page 3-83.
- Performed any required data preprocessing operations. To improve the accuracy of your model, you should detrend your data. See "Ways to Prepare Data for System Identification" on page 2-6.
- Select a model order. For more information about how to estimate models with selected orders, see "Estimate State-Space Model With Order Selection" on page 3-83.

Estimating State-Space Models in the GUI

To estimate a state-space model with free parameterization in the System Identification Tool GUI:

1. In the System Identification Tool GUI, select **Estimate > State Space Models** to open the Polynomial and State Space Models dialog box.

2. In the **Structure** list, select `State Space: n`.

   This action updates the options in the Polynomial and State Space Models dialog box to correspond with this model structure. For information about each model structure, see "What Are State-Space Models?" on page 3-79.

3. In the **Orders** field, specify the model order, for example, as `4`.

   Tip To enter model order using the Order Editor dialog box, click **Order Editor**. Using this editor, you can also configure options that influence the estimation such as `N4Weight` and `N4Horizon`.

4. Select the estimation **Method** as **N4SID** or **PEM**. For more information about these methods, "State-Space Model Estimation Algorithms" on page 3-112.

5. Choose to estimate either a continuous-time or discrete-time model. You cannot estimate a discrete-time model if the working data is continuous-time frequency domain data.

6. Specify whether the model has feedthrough or not. In the **Feedthrough** edit box, enter a logical vector of length equal to number of inputs. For example, for a model with 2 inputs, you may enter `[true false]` , which means that the first input has direct feedthrough to the model's outputs while the second input does not.

7. Select the type of state-space parameterization using the **Form** list.

8. Specify any input delays using the **Input delay** edit box. Specify the value as a vector of length equal to number of inputs. For continuous-time models, the delay must be specified in seconds. For discrete-time models, specify the value in terms on number of lags (multiples of sample data time).

9. 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.

10. In the **Focus** list, select how to weigh the relative importance of the fit at different frequencies. For more information about each option, see "Assigning Estimation Weightings" on page 3-110.

11 In the **Initial state** list, specify how you want the algorithm to treat initial states. For more information about the available options, see "Specifying Initial States for Iterative Estimation Algorithms" on page 3-111.

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

---

12 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.

13 (PEM only) To view the estimation progress, select the **Display progress** check box. This launches a progress viewer window in which estimation progress is reported.

14 Click **Estimate** to add this model to the System Identification Tool GUI.

15 (PEM only) To stop the search and save the results after the current iteration has been completed, click **Stop Iterations**. To continue iterations from the current model, click the **Continue iter** button to assign current parameter values as initial guesses for the next search.

Next Steps

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

How to Estimate State-Space Models at the Command Line

Prerequisites

Before you can perform this task, you must have

- Input-output data as an `iddata` object or frequency response data as an `frd` or `idfrd` object. See "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-55. For supported data formats, see "Data Supported by State-Space Models" on page 3-83.
- Performed any required data preprocessing operations. To improve the accuracy of results when using time domain data, you can detrend the data or specify the input/output offset levels as estimation options. See "Ways to Prepare Data for System Identification" on page 2-6.
- Select a model order. For more information about how to estimate model orders, see "Estimate State-Space Model With Order Selection" on page 3-83.

Black Box vs. Structured State-Space Model Estimation

You can estimate state-space models in two ways, depending upon your prior knowledge of the nature of the system and your requirements.

Black Box Estimation. You specify the model order, and, optionally, additional model structure attributes that configure the overall structure of the state-space matrices. In this way of estimating models, you call `ssest` or `n4sid` with data and model order as primary input arguments, and use name-value pairs to specify any additional attributes, such as model sample time, presence of feedthrough, absence of noise component, etc. You do not work directly with the coefficients of the *A*, *B*, *C*, *D*, *K*, and *X0* matrices.

Structured Estimation. You create and configure an `idss` model that contains the initial values for all the system matrices. You use the `Structure` property of the `idss` model to specify all the parameter constraints. For example, you can designate certain coefficients of system matrices as fixed and impose minimum/maximum bounds on the values of the others. For quick configuration of the parameterization and whether to estimate feedthrough and disturbance dynamics, use `ssform`.

After configuring the `idss` model with desired constraints, you specify this model as an input argument to the `ssest` command. You cannot use `n4sid` for structured estimation.

Note

- The structured estimation approach is also referred to as grey-box modeling. However, in this toolbox, the “grey box modeling” terminology is used only when referring to `idgrey` and `idnlgrey` models.
- Using the structured estimation approach, you cannot specify relationships among state-space coefficients. Each coefficient is essentially considered to be independent of others. For imposing dependencies, or to use more complex forms of parameterization, use the `idgrey` model and the associated `greyest` estimator.

Estimating State-Space Models Using ssest and n4sid

You can estimate continuous-time and discrete-time state-space model using the iterative estimation command `ssest` that minimizes the prediction errors to obtain maximum-likelihood values. You can also use the noniterative subspace estimator `n4sid`.

Use the following general syntax to both configure and estimate state-space models:

```
m = ssest(data,n,opt,Name,Value)
```

where `data` is the estimation data, `n` is the model order. `opt` contains the options for configuring the estimation of the state-space models. These options include the handling of the initial conditions, input offset, estimation focus and search algorithm options. opt can be followed by name and value pair input arguments that specify optional model structure attributes such as the presence of feedthrough, the canonical form of the model, and input delay. For more information, see `ssest`.

As an alternative to `ssest`, you can use `n4sid`:

```
m = n4sid(data,n,opt,Name,Value)
```

Unless the sample time is specified as a name and value pair input argument, `ssest` estimates a continuous-time model, while `n4sid` estimates a model whose sample time matches that of data.

Note `ssest` uses `n4sid` to initialize the state-space matrices, and takes longer than `n4sid` to estimate a model but typically provides better fit to data.

For more information about estimating model order, see “Estimate Model With Selected Order at the Command Line” on page 3-87.

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

Choosing the Structure of A, B, C Matrices

By default, all entries of the *A*, *B*, and *C* state-space matrices are treated as free parameters. Using the `Form` name and value pair input argument of `ssest` , you can choose various canonical forms that employ fewer parameters, such as the companion and modal forms.

For more information about estimating a specific state-space parameterization, see the following topics:

Choosing Between Continuous-Time and Discrete-Time Representations

For estimation of state-space models, you have the option of switching the model sample time between zero and that of the estimation data. You can do this using the `Ts` name and value pair input argument.

- By default, `ssest` estimates a continuous-time model. If you are using data set with nonzero sample time, `data`, which includes all time domain data, you can also estimate a discrete-time model by using:

  ```
  model = ssest(data,nx,'Ts',data.Ts);
  ```

  If you are using continuous-time frequency-domain data, you cannot estimate a discrete-time model.

- By default, `n4sid` estimates a model whose sample time matches that of the data. Thus, for time-domain data, `n4sid` delivers a discrete-time model. You can estimate a continuous-time model by using:

  ```
  model = n4sid(data,nx,'Ts',0);
  ```

Choosing to Estimate D, K, and X0 Matrices

For state-space models with any parameterization, you can specify whether to estimate the *D*, *K* and *X0* matrices, which represent the input-to-output feedthrough, noise model and the initial states, respectively.

For state-space models with structured parameterization, you can also specify to estimate the *D* matrix. However, for free and canonical forms, the structure of the *D* matrix is set based on your choice of `'Feedthrough'` name and value pair input argument.

D Matrix. By default, the D matrix is not estimated and its value is fixed to zero, except for static models.

- **Black box estimation:** Use the `Feedthrough` name and value pair input argument to denote the presence or absence of feedthrough from individual inputs. For example, in case of a two input model such that there is feedthrough from only the second output, use `model = n4sid(data,n,'Feedthrough',[false true]);`.
- **Structured estimation:** Configure the values of the `init_sys.Structure.d`, where `init_sys` is an `idss` model that represents the desired model structure. To force no feedthrough for the i-th input, set:

  ```
  init_sys.Structure.d.Value(:,i) = 0;
  init_sys.Structure.d.Free = true;
  init_sys.Structure.d.Free(:,i) = false;
  ```

  The first line specifies the value of the i-th column of D as zero. The next line specifies all the elements of D as free, estimable parameters. The last line specifies that the i-th column of the D matrix is fixed for estimation.

  Alternatively, use `ssform` with `'Feedthrough'` name-value pair..

K Matrix. K represents the noise matrix of the model, such that the noise component of the model is:.

$$\dot{x} = Ax + Ke$$
$$y_n = Cx + e$$

For frequency-domain data, no noise model is estimated and K is set to 0. For time-domain data, K is estimated by default in the black box estimation setup.

- **Black box estimation:** Use the `DisturbanceModel` name and value pair input argument to indicate if the disturbance component is fixed to zero (specify `Value = `none'`) or estimated as a free parameter (specify `Value = `estimate'`). For example, use `model = n4sid(data,n,'DisturbanceModel','none')`.
- **Structured estimation:** Configure the value of the `init_sys.Structure.k` parameter, where `init_sys` is an `idss`

model that represents the desired model structure. You can fix some K matrix coefficients to known values and prescribe minimum/maximum bounds for free coefficients. For example, to estimate only the first column of the *K* matrix for a two output model:

```
kpar = init_sys.Structure.k;
kpar.Free(:,1) = true;
kpar.Free(:,2) = false;
kpar.Value(:,2) = 0; % second column value is fixed to zero
init_sys.Structure.k = kpar;
```

Alternatively, use `ssform`.

When not sure how to easily fix or free all coefficients of *K*, initially you can omit estimating the noise parameters in *K* to focus on achieving a reasonable model for the system dynamics. After estimating the dynamic model, you can use `ssest` to refine the model while configuring the *K* parameters to be free. For example:

```
init_sys = ssest(data, n,'DisturbanceModel','none');
init_sys.Structure.k.Free = true;
sys = ssest(data, init_sys);
```

where `init_sys` is the dynamic model without noise.

To set *K* to zero in an existing model, you can set its `Value` to `0` and `Free` flag to `false`:

```
m.Structure.k.Value = 0;
m.Structure.k.Free = false;
```

X0 Matrices. The initial state vector *X0* is obtained as the by-product of model estimation. The `n4sid` and `ssest` commands return the value of *X0* as their second output arguments. You can choose how to handle initial conditions during model estimation by using the `InitialState` estimation option. Use `n4sidOptions` (for `n4sid`) and `ssestOptions` (for `ssest`) to create the estimation option set. For example, in order to hold the initial states to zero during estimation using `n4sid`:

```
opt = n4sidOptions;
opt.InitialState = 'zero';
```

```
[m,X0] = n4sid(data,n,opt);
```

The returned `X0` variable is a zero vector of length `n`.

When you estimate models using multiexperiment data, the `X0` matrix contains as many columns as data experiments.

For a complete list of values for the `InitialStates` option, see "Specifying Initial States for Iterative Estimation Algorithms" on page 3-111.

How to Estimate Free-Parameterization State-Space Models

The default parameterization of the state-space matrices A, B, C, D, and K is free; that is, any elements in the matrices are adjustable by the estimation routines. Because the parameterization of A, B, and C is free, a basis for the state-space realization is automatically selected to give well-conditioned calculations.

To estimate the disturbance model K, you must use time domain data.

Suppose that you have no knowledge about the internal structure of the discrete-time state-space model. To quickly get started, use the following syntax:

```
m = ssest(data)
```

where `data` is your estimation data. This command estimates a continuous-time state-space model for an automatically selected order between 1 and 10.

To find a model of a specific order `n`, use the following syntax:

```
m = ssest(data,n)
```

The iterative algorithm `ssest` is initialized by the subspace method `n4sid`. You can use `n4sid` directly, as an alternative to `ssest`:

```
m = n4sid(data,n)
```

How to Estimate State-Space Models with Canonical Parameterization

What Is Canonical Parameterization?

Canonical parameterization represents a state-space system in a reduced parameter form where the many entries of A, B and C matrices are fixed to zeros and ones. The free parameters appear in only a few of the rows and columns in system matrices A, B, C, and D, and the remaining matrix elements are fixed to zeros and ones.

The software supports the following canonical forms:

- Companion form: The characteristic polynomial appears in the rightmost column of the A matrix.
- Modal decomposition form: The state matrix A is block diagonal, with each block corresponding to a cluster of nearby modes.
- Observability canonical form: The free parameters appear only in select rows of the A matrix.

For more information about the distribution of free parameters in the observability canonical form, see the appendix on identifiability of black-box multivariable model structures in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999 (equation 4A.16).

Note The modal form has a certain symmetry in its block diagonal elements. If you update the parameters of a model of this form (as a structured estimation using ssest), the symmetry is not preserved, even though the updated model is still block-diagonal.

Estimating Canonical State-Space Models

You can estimate state-space models with chosen parameterization at the command line.

To specify a canonical form for `A`, `B`, `C`, and `D`, use the `'Form'` name and value pair input argument in the estimator syntax, as follows:

```
m = ssest(data, n, 'Form', 'canonical')
```

If you have time-domain data, the preceding command estimates a continuous-time model. If you want a discrete-time model, specify the data sample time using the `'Ts'` name and value pair input argument:

```
md = ssest(data, n, 'Form', 'canonical', 'Ts', data.Ts)
```

If you have continuous-time frequency-domain data, you can only estimate a continuous-time model.

How to Estimate State-Space Models with Structured Parameterization

What Is Structured Parameterization?

Structured parameterization lets you exclude specific parameters from estimation by setting these parameters to specific values. This approach is useful when you can derive state-space matrices from physical principles and provide initial parameter values based on physical insight. You can use this approach to discover what happens if you fix specific parameter values or if you free certain parameters.

In the case of structured parameterization, there are two stages to the estimation procedure:

1. Using the `idss` command to specify the structure of the state-space matrices and the initial values of the free parameters
2. Using the `ssest` estimation command to estimate the free model parameters

This approach differs from estimating models with free and canonical parameterizations, where it is not necessary to specify initial parameter values before the estimation. For free parameterization, there is no structure to specify because it is assumed to be unknown. For canonical parameterization, the structure is fixed to a specific form.

Note To estimate structured state-space models in the System Identification Tool GUI, define the corresponding model structures at the command line and import them into the System Identification Tool GUI.

Specifying the State-Space Structure

To specify the state-space model structure, first define the `A`, `B`, `C`, `D`, and `K` matrices in the MATLAB workspace.

To define a discrete-time state-space structure, use the following syntax:

```
m = idss(A,B,C,D,K,'Ts',T)
```

where `A`, `B`, `C`, `D`, and `K` specify the initial values for the free parameters. `T` is the sampling interval.

Similarly, to define a continuous-time state-space structure, use the following syntax:

```
m = idss(A,B,C,D,K,'Ts',0)
```

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 do this, use the `Structure` property of the model. `Structure` contains parameters for the five

state-space matrices, *A*, *B*, *C*, *D*, and *K*. For each parameter, you can set the following attributes:

- `Value` — Parameter values.
- `Minimum` — Minimum value that the parameter can assume during estimation.
- `Maximum` — Maximum value that the parameter can assume during estimation.
- `Free` — Boolean specifying whether the parameter is a free estimation variable. If you want to fix the value of a parameter during estimation, set the corresponding `Free = false`. For example, if *A* is a 3-by-3 matrix of the model `sys`, `sys.Structure.a.Free = eye(3)` fixes all of the off-diagonal entries in *A*, to the values specified in `sys.Structure.a.Value`. In this case, only the diagonal entries in *A* are estimable.
- `Scale` — Scale of the parameter's value. `Scale` is not used in estimation.
- `Info` — Structure array for storing parameter units and labels. The structure has `Label` and `Unit` fields.

  Use these fields for your convenience, to store strings that describe parameter units and labels.

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 do this, use:

```
m.Structure.a.Value(1,2) = 0;
m.Structure.a.Value(2,1) = 0;
m.Structure.a.Free(1,2) = false;
m.Structure.a.Free(2,1) = false;
```

Alternatively, to quickly configure the parameterization and whether to estimate feedthrough and disturbance dynamics, use `ssform`.

The estimation algorithm only estimates the parameters in `A` for which `m.Structure.a.Free` is `true`.

Finally, use `ssest` to estimate the model, as described in "How to Estimate State-Space Models at the Command Line" on page 3-92.

Use physical insight, whenever possible, to initialize the parameters for the iterative search algorithm. Because it is possible that the numerical minimization gets stuck in a local minimum, try several different initialization values for the parameters. For random initialization, use the `init` command. When the model structure contains parameters with different orders of magnitude, try to scale the variables so that the parameters are all roughly the same magnitude.

The iterative search computes gradients of the prediction errors with respect to the parameters using numerical differentiation. The step size is specified by the `nuderst` command. The default step size is equal to 10^{-4} times the absolute value of a parameter or equal to 10^{-7}, whichever is larger. To specify a different step size, edit the `nuderst` MATLAB file.

Are Grey-Box Models Similar to State-Space Models with Structured Parameterization?

Structured parameterization state-space models are similar to grey-box modeling. However, the state-space models are simpler to estimate than grey-box models. The structured estimation approach is also referred to as grey-box modeling. However, in this toolbox, the "grey box modeling" terminology is used only when referring to `idgrey` and `idnlgrey` models. Using the structured estimation approach, you cannot specify relationships among state-space coefficients. Each coefficient is essentially considered to be independent of others. For imposing dependencies, or to use more complex forms of parameterization, use the `idgrey` model and the associated `greyest` estimator.

To learn more about grey-box models, see "Grey-Box Model Estimation".

Example – Estimating Structured Discrete-Time State-Space Models

In this example, you estimate the unknown parameters $(\theta_1,\theta_2,\theta_3,\theta_4,\theta_5)$ in the following discrete-time model:

$$x(t+1) = \begin{bmatrix} 1 & \theta_1 \\ 0 & 1 \end{bmatrix} x(t) + \begin{bmatrix} \theta_2 \\ \theta_3 \end{bmatrix} u(t) + \begin{bmatrix} \theta_4 \\ \theta_5 \end{bmatrix} e(t)$$
$$y(t) = \begin{bmatrix} 1 & 0 \end{bmatrix} x(t) + e(t)$$
$$x(0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Suppose that the nominal values of the unknown parameters ($\theta_1, \theta_2, \theta_3, \theta_4, \theta_5$) are `-1`, `2`, `3`, `4`, and `5`, respectively.

The discrete-time state-space model structure is defined by the following equation:

$$x(kT+T) = Ax(kT) + Bu(kT) + Ke(kT)$$
$$y(kT) = Cx(kT) + Du(kT) + e(kT)$$
$$x(0) = x0$$

To construct and estimate the parameters of this discrete-time state-space model:

1. Construct the parameter matrices and initialize the parameter values using the nominal parameter values:

```
A = [1,-1;0,1];
B = [2;3];
C = [1,0];
D = 0;
K = [4;5];
```

2. Construct the state-space model object:

```
m = idss(A,B,C,D,K);
```

3. Specify the parameter values in the structure matrices that you do not want to estimate:

```
S = m.Structure;
S.a.Free(1,1) = false;
S.a.Free(2,:) = false;
```

```
S.c.Free = false;
m.Structure = S;
```

D is initialized, by default, as a fixed value and *K* and *B* are initialized as free values. Suppose you want to fix the initial states to known zero values. To enforce this, configure the `InitialState` estimation option:

```
opt = ssestOptions;
opt.InitialState = 'zero';
```

4 Estimate the model structure:

```
m = ssest(data, m, opt)
```

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`, and `K` matrices.

Example – Estimating Structured Continuous-Time State-Space Models

In this example, you estimate the unknown parameters $(\theta_1, \theta_2, \theta_3)$ in the following continuous-time model:

$$\dot{x}(t) = \begin{bmatrix} 0 & 1 \\ 0 & \theta_1 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ \theta_2 \end{bmatrix} u(t)$$

$$y(t) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x(t) + e(t)$$

$$x(0) = \begin{bmatrix} \theta_3 \\ 0 \end{bmatrix}$$

This equation represents an electrical motor, where $y_1(t) = x_1(t)$ is the angular position of the motor shaft, and $y_2(t) = x_2(t)$ is the angular velocity.

The parameter $-\theta_1$ is the inverse time constant of the motor, and $-\theta_2/\theta_1$ is the static gain from the input to the angular velocity.

The motor is at rest at t=0, but its angular position θ_3 is unknown. Suppose that the approximate nominal values of the unknown parameters are $\theta_1 = -1$ and $\theta_2 = 0.25$. The variance of the errors in the position measurement is `0.01`, and the variance in the angular velocity measurements is `0.1`. For more information about this example, see the section on state-space models in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

The continuous-time state-space model structure is defined by the following equation:

$$\dot{x}(t) = Fx(t) + Gu(t) + \tilde{K}w(t)$$
$$y(t) = Hx(t) + Du(t) + w(t)$$
$$x(0) = x0$$

To construct and estimate the parameters of this continuous-time state-space model:

1. Construct the parameter matrices and initialize the parameter values using the nominal parameter values:

   Note The following matrices correspond to continuous-time representation. However, to be consistent with the `idss` object property name, this example uses `A`, `B`, and `C` instead of `F`, `G`, and `H`.

   ```
   A = [0 1;0 -1];
   B = [0;0.25];
   C = eye(2);
   D = [0;0];
   K = zeros(2,2);
   x0 = [0;0];
   ```

2. Construct the continuous-time state-space model object:

   ```
   m = idss(A,B,C,D,K,'Ts',0);
   ```

3 Specify the parameter values in the structure matrices that you do not want to estimate:

```
S = m.Structure;
S.a.Free(1,:) = false;
S.a.Free(2,1) = false;
S.b.Free(1) = false;
S.c.Free = false;
S.d.Free = false;
S.k.Free = false;
m.Structure = S;

m.NoiseVariance = [0.01 0; 0 0.1];
```

The initial state is partially unknown. Use the `InitialState` option of the `ssestOptions` option set to configure the estimation behavior of *X0*:

```
opt = ssestOptions;
opt.InitialState = idpar(x0);
opt.InitialState.Free(2) = false;
```

4 Estimate the model structure:

```
m = ssest(data, m, opt)
```

where `data` is name of the `iddata` object containing time-domain or frequency-domain data. For this example, you may use the data set `dcmotordata` that is available in the folder *matlabroot*/toolbox/ident/iddemos/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 during estimation.

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,'Ts',0.1);
simopt = simOptions('AddNoise', true, 'NoiseData', e)
y = sim(m,simdat,simopt)
```

The continuous system is sampled using `Ts=0.1` for simulation purposes. The noise sequence is scaled according to the matrix `m.NoiseVariance`..

If you discover that the motor was not initially at rest, you can estimate $x_2(0)$ by setting the second element of the `InitialState` parameter to be free:

```
opt.InitialState.Free(2) = true;
m_new = ssest(data, m, opt)
```

How to Estimate the State-Space Equivalent of ARMAX and OE Models

This example shows how to estimate ARMAX and OE form models using the state-space estimation approach.

You can estimate the equivalent of ARMAX and output-error (OE) multiple-output models using state-space model structures. For the ARMAX case, specify to estimate the K matrix for the state-space model. For the OE case, set K = 0. Convert the resulting model into `idpoly` models to see them in the commonly defined ARMAX or Output-Error forms.

Load measured data.

```
load iddata1 z1
```

Estimate state-space models.

```
mss_noK = n4sid(z1, 2,'DisturbanceModel','none');
mss = n4sid(z1,2);
```

`mss_noK` is a second order state-space model with no disturbance model used during estimation. `mss` is also a second order state-space model, but with an estimated noise component. Both models use the measured data set `z1` for estimation.

Convert the state-space models to polynomial models.

```
mOE = idpoly(mss_noK);
mARMAX = idpoly(mss);
```

Converting to polynomial models results in the parameter covariance information for `mOE` and `mARMAX` to be lost.

You can use one of the following to recompute the covariance:

- Zero-iteration update using the same estimation data.
- `translatecov` as a Gauss approximation formula based translation of covariance of `mss_noK` and `mss` into covariance of `mOE` and `mARMAX`.

Reestimate `mOE` and `mARMAX` for the parameters of the polynomial model using a zero iteration update.

```
opt = polyestOptions;
opt.SearchOption.MaxIter = 0;

mOE = polyest(z1,mOE);
mARMAX = polyest(z1,mARMAX);
```

The options object, `opt`, specifies a zero iteration update for `mOE` and `mARMAX`. Consequently, the model parameters remain unchanged and only their covariance information is updated.

Alternatively, you can use `translatecov` to convert the estimated models into polynomial form:

```
fcn = @(x)idpoly(x);
mOE = translatecov(fcn, mss_noK)
mARMAX = translatecov(fcn, mss)
```

Because `polyest` and `translatecov` use different computation algorithms, the covariance data obtained by running a zero-iteration update may not match that obtained using `translatecov`.

View the uncertainties of the model parameters.

```
present(mOE)
present(mARMAX)
```

Tip You can use a state-space model with K = 0 (Output-Error (OE) form) for initializing a Hammerstein-Wiener estimation at the command line. This initialization may improve the fit of the model. See "Using Linear Model for Hammerstein-Wiener Estimation" on page 4-63.

For more information about ARMAX and OE models, see "Identifying Input-Output Polynomial Models" on page 3-45.

Assigning Estimation Weightings

You can specify how the estimation algorithm weighs the fit at various frequencies. This information supports the estimation procedures "How to Estimate State-Space Models in the GUI" on page 3-89 and "How to Estimate State-Space Models at the Command Line" on page 3-92.

In the System Identification Tool GUI. Set **Focus** to one of the following options:

- `Prediction` — Uses the inverse of the noise model *H* to weigh the relative importance of how closely to fit the data in various frequency ranges. Corresponds to minimizing one-step-ahead prediction, which typically favors the fit over a short time interval. Optimized for output prediction applications.
- `Simulation` — Uses the input spectrum to weigh the relative importance of the fit in a specific frequency range. Does not use the noise model to weigh the relative importance of how closely to fit the data in various frequency ranges. Optimized for output simulation applications.
- `Stability` — Estimates the best stable model. For more information about model stability, see "Unstable Models" on page 8-94.
- `Filter` — Specify a custom filter to open the Estimation Focus dialog box, where you can enter a filter, as described in "Simple Passband Filter" on page 2-126 or "Defining a Custom Filter" on page 2-127. This prefiltering applies only for estimating the dynamics from input to output. The disturbance model is determined from the estimation data.

At the command line. Specify the focus as an estimation option using the same options as in the GUI. For example, use this command to emphasize the fit between the 5 and 8 rad/s:

```
opt = ssestOptions;
opt.Focus = [5 8];
model = ssest(data,4,opt);
```

For more information on the `'Focus'` option, see the reference page for `ssestOptions`.

Specifying Initial States for Iterative Estimation Algorithms

If you estimate state-space models using the iterative estimation algorithm `ssest`, you can specify how the algorithm treats initial states. This information supports the estimation procedures "How to Estimate State-Space Models in the GUI" on page 3-89 and "How to Estimate State-Space Models at the Command Line" on page 3-92.

In the System Identification Tool GUI. Set **Initial state** to one of the following options:

- `Auto` — Automatically chooses `Zero`, `Estimate`, or `Backcast` based on the estimation data. If initial states have negligible effect on the prediction errors, the initial states are set to zero to optimize algorithm performance.
- `Zero` — Sets all initial states to zero.
- `Estimate` — Treats the initial states as an unknown vector of parameters and estimates these states from the data.
- `Backcast` — Estimates initial states using a backward filtering method (least-squares fit).

At the command line. Specify the method for handling initial states using the `InitialState` estimation option. For example, to estimate a fourth-order state-space model and set the initial states to be estimated from the data:

```
opt = ssestOptions('InitialState','estimate');
m = ssest(data, 4, opt)
```

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

State-Space Model Estimation Algorithms

For linear state-space models, you can use the subspace method, called *N4SID*. You can use the subspace method N4SID to get an initial model (see the `n4sid` reference page), and then try to refine the initial estimate using the iterative prediction-error method SSEST (see the `ssest` reference page).

N4SID is faster than SSEST, but is typically less accurate and robust, and requires additional arguments that might be difficult to specify.

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

Identifying Transfer Function Models

In this section...

What are Transfer Function Models?

Definition of Transfer Function Models

Transfer function models describe the relationship between the inputs and outputs of a system using a ratio of polynomials. The *model order* is equal to the order of the denominator polynomial. The roots of the denominator polynomial are referred to as the model *poles*. The roots of the numerator polynomial are referred to as the model *zeros*.

The parameters of a transfer function model are its poles, zeros and transport delays.

Continuous-Time Representation

In continuous-time, a transfer function model has the form:

$$Y(s) = \frac{num(s)}{den(s)}U(s) + E(s)$$

Where, $Y(s)$, $U(s)$ and $E(s)$ represent the Laplace transforms of the output, input and noise, respectively. $num(s)$ and $den(s)$ represent the numerator and denominator polynomials that define the relationship between the input and the output.

Discrete-Time Representation

In discrete-time, a transfer function model has the form:

$$y(t) = \frac{num(q^{-1})}{den(q^{-1})}u(t) + e(t)$$

$$num(q^{-1}) = b_0 + b_1 q^{-1} + b_2 q^{-2} + \ldots$$

$$den(q^{-1}) = 1 + a_1 q^{-1} + a_2 q^{-2} + \ldots$$

The roots of *num*(*q^-1*) and *den*(*q^-1*) are expressed in terms of the lag variable *q^-1*.

Delays

In continuous-time, input and transport delays are of the form:

$$Y(s) = \frac{num(s)}{den(s)}e^{-s\tau}U(s) + E(s)$$

Where τ represents the delay.

In discrete-time:

$$y(t) = \frac{num}{den}u(t - \tau) + e(t)$$

where *num* and *den* are polynomials in the lag operator `q^(-1)`.

Data Supported by Transfer Function Models

You can estimate transfer function models from data with the following characteristics:

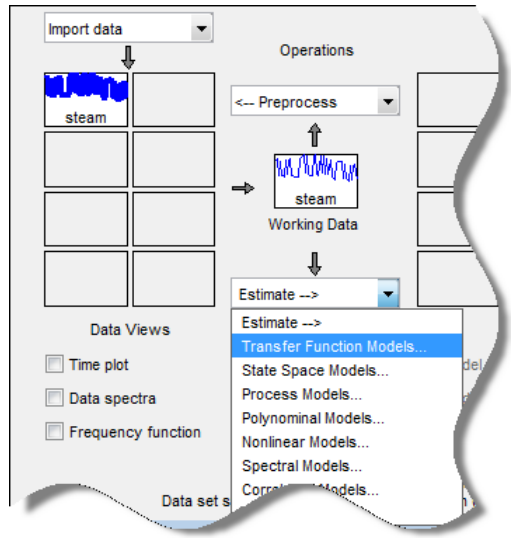
- Real data or complex data
- Single-output and multiple-output
- Time- or frequency-domain data

  Note that you cannot use time-series data for transfer function model identification.

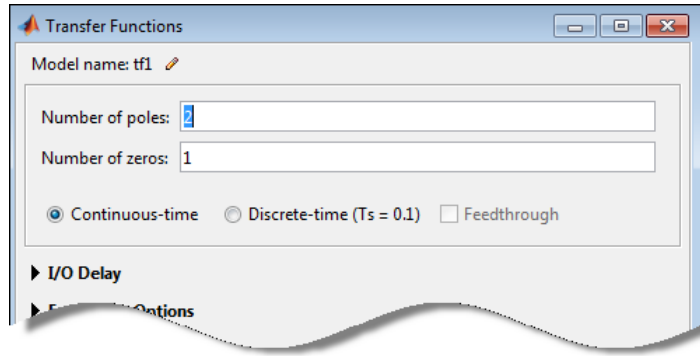
You must first import your data into the MATLAB workspace, as described in "Data Preparation".

How to Estimate Transfer Function Models in the System Identification Tool

1. Import data into the System Identification Tool. See "Importing Time-Domain Data into the GUI" on page 2-18. For supported data formats, see "Data Supported by Transfer Function Models" on page 3-115.
2. Perform any required data preprocessing operations. If input and/or output signals contain nonzero offsets, consider detrending your data. See "Ways to Prepare Data for System Identification" on page 2-6.
3. In the System Identification Tool, select **Estimate > Transfer Function Models**



The Transfer Functions dialog box opens.



4 Specify the number of poles and zeros of the transfer function as nonnegative integers.

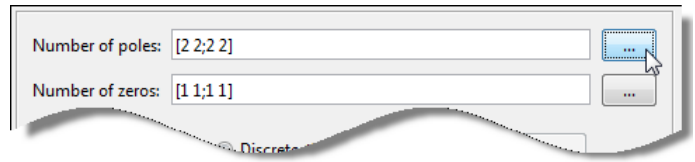
For systems that are multiple input, multiple output, or both:

- To use the same number of poles or zeros for all the input/output pairs, specify a scalar.

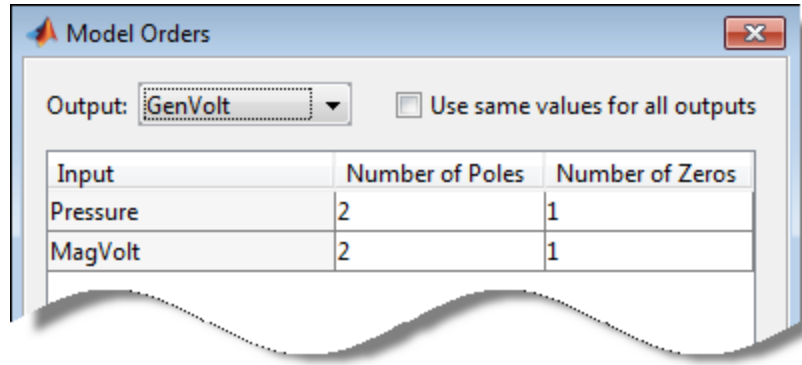
- To use a different number of poles and zeros for the input/output pairs, specify an *ny*-by-*nu* matrix. *ny* is the number of outputs and *nu* is the number of inputs.

  Alternatively, click 

.

  The Model Orders dialog box opens where you specify the number of poles and zeros for each input/output pair. Use the **Output** list to select an output.

5 Select **Continuous-time** or **Discrete-time** to specify whether the model is a continuous- or discrete-time transfer function.

For discrete-time models, the number of poles and zeros refers to the roots of the numerator and denominator polynomials expressed in terms of the lag variable `q^-1`.

6 (For discrete-time models only) Specify whether to estimate the model feedthrough.

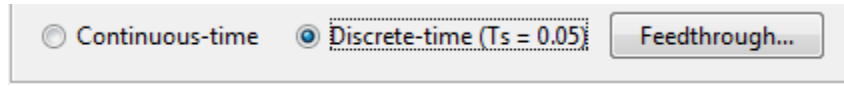
A discrete-time model with 2 poles and 3 zeros takes the following form:

$$Hz^{-1} = \frac{b0 + b1z^{-1} + b2z^{-2} + b3z^{-3}}{1 + a1z^{-1} + a2z^{-2}}$$

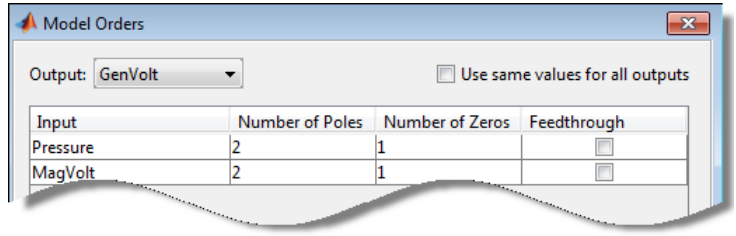
When the model has direct feedthrough, `b0` is a free parameter whose value is estimated along with the rest of the model parameters `b1`, `b2`, `b3`, `a1`, `a2`. When the model has no feedthrough, `b0` is fixed to zero.

- For SISO models, select the **Feedthrough** check box.

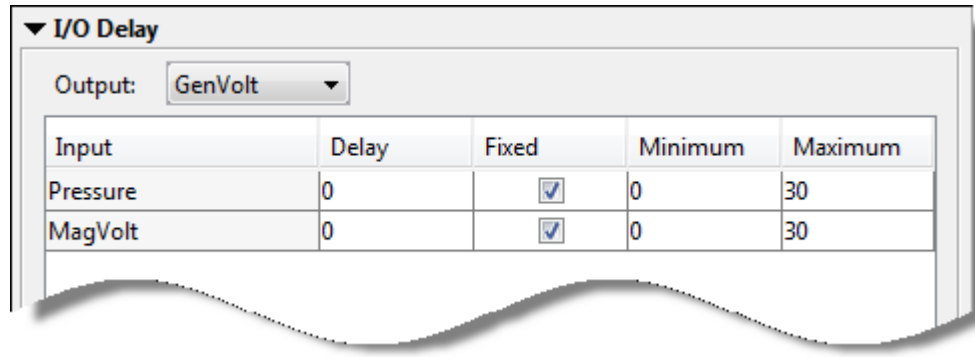
- For models that are multi input, multi output or both, click **Feedthrough**.



The Model Orders dialog box opens, where you specify to estimate the feedthrough for each input/output pair separately. Use the **Output** list to select an output.

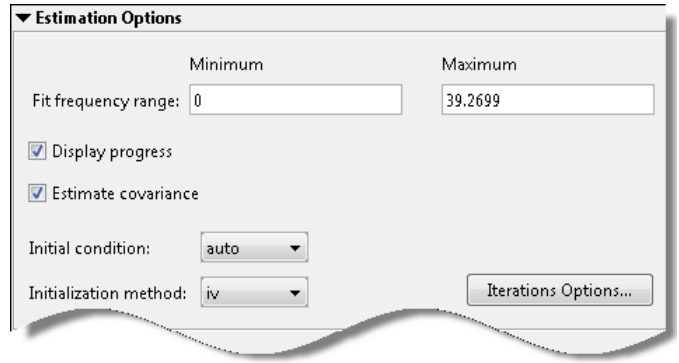


7 Expand the **I/O Delay** section to specify nominal values and constraints for transport delays for different input/output pairs.



Use the **Output** list to select an output. Select the **Fixed** check box to specify a transport delay as a fixed value. Specify its nominal value in the **Delay** field.

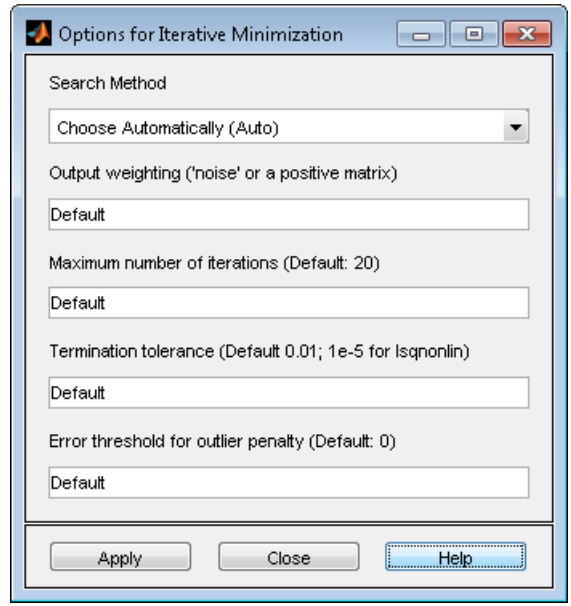
8 Expand the **Estimation Options** section to specify estimation options.



- In the **Minimum** and **Maximum** fields, specify the frequency range over which the transfer function model must fit the data.
- Select **Display progress** to view the progress of the optimization.
- Select **Estimate covariance** to estimate the covariance of the transfer function parameters.
- Specify how to treat the initial conditions in the **Initial condition** list. For more information, see “Specifying Initial Conditions for Iterative Estimation Algorithms” on page 3-127.

- Specify the algorithm used to initialize the values of the numerator and denominator coefficients in the **Initialization method** list.
  - `'iv'` — Instrument Variable approach.
  - `'svf'` — State Variable Filters approach.
  - `'gpmf'` — Generalized Poisson Moment Functions approach.
  - `'n4sid'` — Subspace state-space estimation approach.
  - `` `all' `` — Combination of all of the above approaches. The software tries all the above methods and selects the method that yields the smallest value of prediction error norm.

9 Click **Iterations Options** to specify the options for controlling the iterations. The Options for Iterative Minimization dialog box opens.



In this dialog box, you can specify the following iteration options:

- **Search Method** — Method used by the iterative search algorithm. Search method is `auto` by default. The descent direction is calculated using `gn` (Gauss-Newton), `gna` (Adaptive Gauss-Newton), `lm`

(Levenberg-Marquardt) and `grad` (Gradient Search) successively at each iteration until a sufficient reduction in error is achieved.

- **Output weighting** — Weighting applied to the loss function to be minimized. Use this option for multi-output estimations only. Specify as `'noise'` or a positive semidefinite matrix of size equal the number of outputs.
- **Maximum number of iterations** — Maximum number of iterations to use during search.
- **Termination tolerance** — Tolerance value when the iterations should terminate.
- **Error threshold for outlier penalty** — Robustification of the quadratic criterion of fit.

10 Click **Estimate** to estimate the model. A new model gets added to the System Identification Tool.

11 Validate the model by selecting the appropriate check box in the **Model Views** area of the System Identification Tool. For more information about validating models, see "Validating Models After Estimation" on page 8-3.

12 Export the model to the MATLAB workspace for further analysis. Drag the model to the **To Workspace** rectangle in the System Identification Tool.

How to Estimate Transfer Function Models at the Command Line

Before you estimate a transfer function model, you must have:

- Input/Output data. See "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-55. For supported data formats, see "Data Supported by Transfer Function Models" on page 3-115.
- Performed any required data preprocessing operations. You can detrend your data before estimation. For more information, see "Ways to Prepare Data for System Identification" on page 2-6.

  Alternatively, you can specify the input/output offset for the data using an estimation option set. Use `tfestOptions` to create the estimation option set. Use the `InputOffset` and `OutputOffset` name and value pairs to specify the input/output offset.

Estimate continuous-time and discrete-time transfer function models using `tfest`. The output of `tfest` is an `idtf` object, which represents the identified transfer function.

The general workflow in estimating a transfer function model is:

1. Create a data object (`iddata` or `idfrd`) that captures the experimental data.
2. (Optional) Specify estimation options using `tfestOptions`.
3. (Optional) Create a transfer function model that specifies the expected model structure and any constraints on the estimation parameters.
4. Use `tfest` to identify the transfer function model, based on the data.
5. Validate the model. See "Model Validation".

Transfer Function Structure Specification

You can use a priori knowledge of the expected transfer function model structure to initialize the estimation. The `Structure` property of an `idtf` model contains parameters that allow you to specify the values and constraints for the numerator, denominator and transport delays.

For example, specify a third-order transfer function model that contains an integrator and has a transport delay of at most 1.5 seconds:

```
init_sys = idtf([nan nan],[1 2 1 0]);
init_sys.Structure.ioDelay.Maximum = 1.5;
init_sys.Structure.den.Free(end) = false;
```

`int_sys` is an `idtf` model with three poles and one zero. The denominator coefficient for the `s^0` term is zero and implies that one of the poles is an integrator.

`init_sys.Structure.ioDelay.Maximum = 1.5` constrains the transport delay to a maximum of 1.5 seconds. The last element of the denominator coefficients (associated with the `s^0` term) is not a free estimation variable. This constraint forces one of the estimated poles to be at `s = 0`.

For more information regarding configuring the initial parameterization of an estimated transfer function, see `Structure` in `idtf`.

Estimate Transfer Function Models by Specifying Number of Poles

This example shows how to identify a transfer function containing a specified number of poles for given data.

Load time domain system response data and use it to estimate a transfer function for the system.

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

`z1` is an `iddata` object that contains time-domain, input-output data.

`np` specifies the number of poles in the estimated transfer function.

`sys` is an `idtf` model containing the estimated transfer function.

To see the numerator and denominator coefficients of the resulting estimated model `sys`, enter:

```
sys.num
sys.den
```

To view the uncertainty in the estimates of the numerator and denominator and other information, use `tfdata`.

How to Estimate Transfer Function Models with Transport Delay to Fit Given Frequency Response Data

This example shows how to identify a transfer function to fit a given frequency response data (FRD) containing additional phase roll off induced by input delay.

Obtain frequency response data.

For this example, use `bode` to obtain the magnitude and phase response data for the following system:

$$H(s) = e^{-.5s}\frac{s+0.2}{s^3+2s^2+s+1}$$

Use 100 frequency points, ranging from 0.1 rad/s to 10 rad/s, to obtain the frequency response data. Use `frd` to create a frequency response data object.

```
freq = logspace(-1,1,100);
[mag, phase] = bode(tf([1 .2],[1 2 1 1],'InputDelay',.5),freq);
data = frd(mag.*exp(1j*phase*pi/180),freq);
```

`data` is an `iddata` object that contains frequency response data for the described system.

Estimate a transfer function using `data`. Specify an unknown transport delay for the identified transfer function.

```
np = 3;
nz = 1;
iodelay = NaN;
sys = tfest(data,np,nz,iodelay)
```

`np` and `nz` specify the number of poles and zeros in the identified transfer function, respectively.

`iodelay` specifies an unknown transport delay for the identified transfer function.

`sys` is an `idtf` model containing the identified transfer function.

How to Estimate Transfer Function Models With Prior Knowledge of Model Structure and Constraints

This example shows how to estimate a transfer function for given data when the structure of the expected model is known. This example also shows how to apply constraints to the numerator and denominator coefficients.

Load time domain data.

```
load iddata1 z1;
z1.y = cumsum(z1.y);
```

`cumsum` integrates the output data of `z1`. The estimated transfer function should therefore contain an integrator.

Create a transfer function model with the expected structure.

```
init_sys = idtf([100 1500],[1 10 10 0]);
```

`int_sys` is an `idtf` model with three poles and one zero. The denominator coefficient for the `s^0` term is zero. Therefore, `int_sys` contains an integrator.

Specify constraints on the numerator and denominator coefficients of the transfer function model. To do so, configure fields in the `Structure` property:

```
init_sys.Structure.num.Minimum = eps;
init_sys.Structure.den.Minimum = eps;
init_sys.Structure.den.Free(end) = false;
```

The constraints specify that the numerator and denominator coefficients are nonnegative. Additionally, the last element of the denominator coefficients (associated with the `s^0` term) is not an estimable parameter. This constraint forces one of the estimated poles to be at `s = 0`.

Create an estimation option set that specifies using the Levenberg–Marquardt search method.

```
opt = tfestOptions('SearchMethod', 'lm');
```

Estimate a transfer function for `z1` using `init_sys` and the estimation option set.

```
sys = tfest(z1,init_sys,opt);
```

`tfest` uses the coefficients of `init_sys` to initialize the estimation of `sys`. Additionally, the estimation is constrained by the constraints you specify in the `Structure` property of `init_sys`. The resulting `idtf` model `sys` contains the parameter values that result from the estimation.

How to Estimate Transfer Function Models with Unknown Transport Delays

This example shows how to estimate a transfer function for given data with unknown transport delays. This example also shows how to apply an upper bound on the unknown transport delays.

Create a transfer function model with the expected numerator and denominator structure and delay constraints.

For this example, the experiment data consists of two inputs and one output. Both transport delays are unknown and have an identical upper bound. Additionally, the transfer functions from both inputs to the output are identical in structure.

```
init_sys = idtf(NaN(1,2),[1, NaN(1,3)],'ioDelay',NaN);
init_sys.Structure(1).ioDelay.Free = true;
init_sys.Structure(1).ioDelay.Maximum = 7;
```

`init_sys` is an `idtf` model describing the structure of the transfer function from one input to the output. The transfer function consists of one zero, three poles and a transport delay. The use of `NaN` indicates unknown coefficients.

`init_sys.Structure(1).ioDelay.Free = true` indicates that the transport delay is not fixed.

`init_sys.Structure(1).ioDelay.Maximum = 7` sets the upper bound for the transport delay to 7 seconds.

```
init_sys = [init_sys, init_sys];
```

`init_sys` now contains the transfer function from both inputs to the output.

Load time domain system response data and detrend .

```
load co2data;
Ts = 0.5;
data = iddata(Output_exp1,Input_exp1,Ts);
T = getTrend(data);
T.InputOffset = [170, 50];
T.OutputOffset = mean(data.y(1:75));
```

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

`data` is an `iddata` object and has a sample rate of 0.5 seconds.

Identify a transfer function for the measured data using the specified delay constraints.

```
sys = tfest(data,init_sys);
```

`sys` is an `idtf` model containing the identified transfer function.

Specifying Initial Conditions for Iterative Estimation Algorithms

If you estimate transfer function models using `tfest`, you can specify how the algorithm treats initial conditions.

In the System Identification Tool, set **Initial condition** to one of the following options:

- `auto` — Automatically chooses `Zero`, `Estimate`, or `Backcast` based on the estimation data. If initial conditions have negligible effect on the prediction errors, the initial conditions are set to zero to optimize algorithm performance.
- `Zero` — Sets all initial conditions to zero.
- `Estimate` — Treats the initial conditions as an estimation parameters.
- `Backcast` — Estimates initial conditions using a backward filtering method (least-squares fit).

At the command line. Specify the initial conditions by using an estimation option set. Use `tfestOptions` to create the estimation option set. For example, create an options set that sets the initial conditions to zero:

```
opt = tfestOptions('InitialCondition','zero);
```

For more information, see `tfestOptions`.

Estimating Transfer Functions with Delays

The `tfest` command supports estimation of IO delays. In the simplest case, if you specify `NaN` as the value for the `ioDelay` input argument, `tfest` estimates the corresponding delay value.

```
load iddata1 z1
sys = tfest(z1, 2, 2, NaN); % 2 poles, 2 zeros, unknown transport delay
```

If you want to assign an initial guess to the value of delay or prescribe bounds for its value, you must first create a template `idtf` model and configure `ioDelay` using the model's `Structure` property:

```
sys0 = idtf([nan nan nan],[1 nan nan]);
sys0.Structure.ioDelay.Value = 0.1; % initial guess
sys0.Structure.ioDelay.Maximum = 1; % maximum allowable value for delay
sys0.Structure.ioDelay.Free = true; % treat delay as estimatable quantity
sys = tfest(data, sys0)
```

If estimation data is in the time-domain, the delays are not estimated iteratively. If a finite initial value is specified, that value is retained as is with no iterative updates. The same is true of discrete-time frequency domain data. Thus in the example above, if `data` has a nonzero sample time, the estimated value of delay in the returned model `sys` is 0.1 (same as the initial guess specified for `sys0`). The delays are updated iteratively only for continuos-time frequency domain data. If, on the other hand, a finite initial value for delay is not specified (e.g., `sys0.Structure.ioDelay.Value = NaN`), then a value for delay is determined using the `delayest` function, regardless of the nature of the data.

Determination of delay as a quantity independent of the model's poles and zeros is a difficult task. Estimation of delays becomes especially difficult for multi-input or multi-output data. It is strongly recommended that you perform some investigation to determine delays before estimation. You can use functions such as `delayest`, `arxstruc`, `selstruc` and impulse response analysis to determine delays. Often, physical knowledge of the system or dedicated transient tests (how long does it take for a step change in input to show up in a measured output?) will reveal the value of transport delays. Use the results of such analysis to assign initial guesses as well as minimum and maximum bounds on the estimated values of delays.

For an example of specifying constraints on delays, see “How to Estimate Transfer Function Models with Unknown Transport Delays” on page 3-126

Refining Linear Parametric Models

In this section...

When to Refine Models

There are two situations where you can refine estimates of linear parametric models.

In the first situation, you have already estimated a parametric model and wish to update the values of its free parameters to improve the fit to the estimation data. This is useful if your previous estimation terminated because of search algorithm constraints such as maximum number of iterations or function evaluations allowed reached. However, if your model captures the essential dynamics, it is usually not necessary to continue improving the fit—especially when the improvement is a fraction of a percent.

In the second situation, you might have constructed a model using one of the model constructors described in "Commands for Constructing Model Structures" on page 1-25. In this case, you built initial parameter guesses into the model structure and wish to refine these parameter values.

What You Specify to Refine a Model

When you refine a model, you must provide two inputs:

- Parametric model
- Data — You can either use the same data set for refining the model as the one you originally used to estimate the model, or you can use a different data set.

How to Refine Linear Parametric Models in the GUI

The following procedure assumes that the model you want to refine is already in the System Identification Tool GUI. You might have estimated this model in the current session or imported the model from the MATLAB workspace. For information about importing models into the GUI, see "Importing Models into the GUI" on page 12-8.

To refine your model:

1. In the System Identification Tool GUI, verify that you have the correct data set in the **Working Data** area for refining your model.

   If you are using a different data set than the one you used to estimate the model, drag the correct data set into the **Working Data** area. For more information about specifying estimation data, see "Specifying Estimation and Validation Data" on page 2-35.

2. Select **Estimate > Polynomial Models** or **Estimate > State Space Models** to open the Polynomial and State Space Models dialog box, if this dialog box is not already open.

3. In the Linear Parametric Models dialog box, select `By Initial Model` from the **Structure** list.

4. Enter the model name into the **Initial model** field, and press **Enter**.

   The model name must be in the Model Board of the System Identification Tool GUI or a variable in the MATLAB workspace. This model need not be a state-space or polynomial model; it could also be a process model (`idproc`) or a transfer function model (`idtf`).

   Tip As a shortcut for specifying a model in the Model Board, you can drag the model icon from the System Identification Tool GUI into the **Initial model** field.

   When you enter the model name, algorithm settings in the Polynomial and State Space Models dialog box override the initial model settings.

5 Modify the iteration options, displayed in the Polynomial and State Space Models and Transfer Functions dialog box, if necessary.

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

7 Validate the new model.

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

How to Refine Linear Parametric Models at the Command Line

If you are working at the command line, you can use `pem` to refine parametric model estimates. You can also use the various model-structure specific estimators - `ssest` for `idss` models, `polyest` for `idpoly` models, `tfest` for `idtf` models, and `greyest` for `idgrey` models.

The general syntax for refining initial models is as follows:

```
m = pem(data,init_model)
```

`pem` uses the properties of the initial model.

You can also specify the estimation options configuring the objective function and search algorithm settings. For more information, see the reference page of the estimating function.

Refine ARMAX Model with Initial Parameter Guesses at Command Line

This example shows how to refine models for which you have initial parameter guesses. Estimate an ARMAX model for the data by initializing the *A*, *B*, and *C* polynomials.

In this case, you must first create a model object and set the initial parameter values in the model properties. Next, you provide this initial model as input to `armax`, `polyest`, or `pem`, which refine the initial parameter guesses using the data.

```
load iddata8
% Define model parameters.
% Leading zeros in B indicate input delay (nk),
% which is 1 for each input channel.
A = [1 -1.2 0.7];
B{1} = [0 1 0.5 0.1]; % first input
B{2} = [0 1.5 -0.5]; % second input
B{3} = [0 -0.1 0.5 -0.1]; % third input
C = [1 0 0 0 0];
Ts = 1;

% Create model object.
init_model = idpoly(A,B,C,'Ts',1);

% Use polyest to update the parameters of the initial model.
model = polyest(z8,init_model)

% Compare the two models.
compare(z8,init_model,model)
```

For more information about estimating polynomial models, see “Identifying Input-Output Polynomial Models” on page 3-45.

Refine Initial ARMAX Model at Command Line

This example shows to estimate an initial model and refine it using `pem`.

Load measured data.

```
load iddata8;
```

Split the data into an initial estimation data set and a refinement data set.

```
init_data = z8(1:100);
refine_data = z8(101:end);
```

`init_data` is an `iddata` object containing the first 100 samples from `z8` and `refine_data` is an `iddata` object representing the remaining data in `z8`.

Estimate an ARMAX model.

```
na=4;
nb=[3 2 3];
nc=2;
nk=[0 0 0];

sys = armax(init_data,[na nb nc nk]);
```

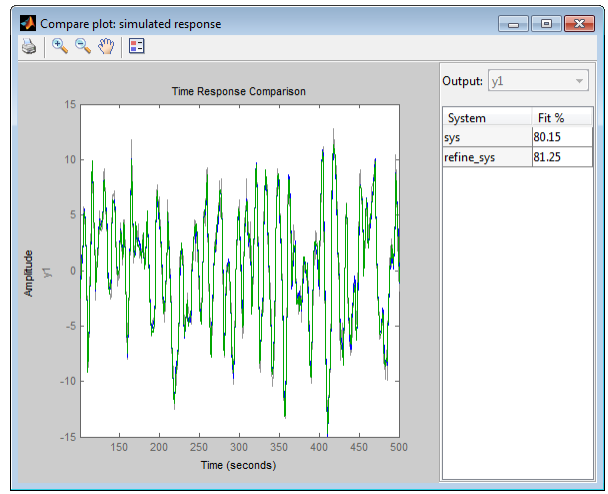
`armax` uses the default algorithm properties to estimate `sys`.

Refine the estimated model by specifying the estimation algorithm options. Specify stricter tolerance and increase the maximum iterations.

```
opt = armaxOptions;
opt.SearchOption.Tolerance = 1e-5;
opt.SearchOption.MaxIter = 50;

refine_sys = pem(refine_data,sys,opt)
```

Compare the fit of the initial and refined models.

```
compare(refine_data,sys,refine_sys)
```



`refine_sys` provides a closer fit to the data than `sys`.

You can similarly use `polyest` or `armax` to refine the estimated model.

See Also

`armaxpempolyest`

Extracting Numerical Model Data

You can extract the following numerical data from linear model objects:

- Coefficients and uncertainty

  For example, extract state-space matrices (`A`, `B`, `C`, `D` and `K`) for state-space models, and polynomials (`a`, `b`, `c`, `d` and `f`) for polynomial models.

  If you estimated model uncertainty data, this information is stored in the model in the form of the parameter covariance matrix. You can fetch the covariance matrix (in its raw or factored form) using the `getcov` command. The covariance matrix represents uncertainties in parameter estimates and is used to compute:

  - Confidence bounds on model output plots, Bode plots, residual plots, and pole-zero plots
  - Standard deviation in individual parameter values. For example, one standard deviation in the estimated value of the `A` polynomial in an ARX model, returned by the `polydata` command and displayed by the `present` command.

- Dynamic and noise models

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

  $$y = Gu + He$$

  G is an operator that takes the measured inputs u to the outputs and captures the system dynamics, also called the *measured model*. H is an operator that describes the properties of the additive output disturbance and takes the hypothetical (unmeasured) noise source inputs e to the outputs, also called the *noise model*. When you estimate a noise model, the toolbox includes one noise channel e for each output in your system.

The following table summarizes the commands for extracting model coefficients and uncertainty.

Commands for Extracting Model Coefficients and Uncertainty Data

Command	Description	Syntax
`freqresp`	Extracts frequency-response data (`H`) and corresponding covariance (`CovH`) from any linear identified model.	`[H,w,CovH] = freqresp(m)`
`polydata`	Extracts polynomials (such as `A`) from any linear identified model. The polynomial uncertainties (such as `dA`) are returned only for `idpoly` models.	`[A,B,C,D,F,dA,dB,dC,dD,dF] = ...` `polydata(m)`

`idssdata`	Extracts state-space matrices (such as `A`) from any linear identified model. The matrix uncertainties (such as `dA`) are returned only for `idss` models.	`[A,B,C,D,K,X0,...`
		`dA,dB,dC,dD,dK,dX0] = ...`
		`idssdata(Model)`

`tfdata`	Extracts numerator and denominator polynomials (`num`, `den`) and their uncertainties (`dnum`, `dden`) from any linear identified model.	`[Num,Den,Ts,dNum,dDen] = ...` `tfdata(Model)`
`zpkdata`	Extracts zeros, poles, and gains (`Z`, `P`, `K`) and their covariances (`covZ`, `covP`, `covK`) from any linear identified model.	`[Z,P,K,Ts,covZ,covP,covK] = ...` `zpkdata(m)`

You can also extract numerical model data by using dot notation to access model properties. For example, `m.A` displays the *A* polynomial coefficients from model `m`. Alternatively, you can use the `get` command, as follows: `get(m,'A')`.

Tip To view a list of model properties, type `get(model)`.

You can operate on extracted model data as you would on any other MATLAB vectors, matrices and cell arrays. You can also pass these numerical values to Control System Toolbox commands, for example, or Simulink blocks.

Transforming Between Discrete-Time and Continuous-Time Representations

In this section...

Why Transform Between Continuous and Discrete Time?

Transforming between continuous-time and discrete-time representations is useful, for example, if you have estimated a discrete-time linear model and require a continuous-time model instead for your application.

You can use `c2d` and `d2c` to transform any linear identified model between continuous-time and discrete-time representations. `d2d` is useful is you want to change the sampling interval of a discrete-time model. All of these operations change the sampling interval, which is called *resampling* the model.

These commands do not transform the estimated model uncertainty. If you want to translate the estimated parameter covariance during the conversion, use `translatecov`.

Note `c2d` and `d2d` correctly approximate the transformation of the noise model only when the sampling interval `T` is small compared to the bandwidth of the noise.

Using the c2d, d2c, and d2d Commands

The following table summarizes the commands for transforming between continuous-time and discrete-time model representations.

Command	Description	Usage Example
`c2d`	Converts continuous-time models to discrete-time models.You cannot use `c2d` for `idproc` models and for `idgrey` models whose `FcnType` is not `'cd'`. Convert these models into `idpoly`, `idtf`, or `idss` models before calling `c2d`.	To transform a continuous-time model `mod_c` to a discrete-time form, use the following command: `mod_d = c2d(mod_c,T)` where `T` is the sampling interval of the discrete-time model.
`d2c`	Converts parametric discrete-time models to continuous-time models.You cannot use `d2c` for `idgrey` models whose `FcnType` is not `'cd'`. Convert these models into `idpoly`, `idtf`, or `idss` models before calling `d2c`.	To transform a discrete-time model `mod_d` to a continuous-time form, use the following command: `mod_c = d2c(mod_d)`
`d2d`	Resample a linear discrete-time model and produce an equivalent discrete-time model with a new sampling interval. You can use the resampled model to simulate or predict output with a specified time interval.	To resample a discrete-time model `mod_d1` to a discrete-time form with a new sampling interval `Ts`, use the following command: `mod_d2 = d2d(mod_d1,Ts)`

The following commands compare estimated model `m` and its continuous-time counterpart `mc` on a Bode plot:

```
% Estimate discrete-time ARMAX model
% from the data
m = armax(data,[2 3 1 2]);
% Convert to continuous-time form
mc = d2c(m);
% Plot bode plot for both models
bode(m,mc)
```

Specifying Intersample Behavior

A sampled signal is characterized only by its values at the sampling instants. However, when you apply a continuous-time input to a continuous-time system, the output values at the sampling instants depend on the inputs at the sampling instants and on the inputs between these points. Thus, the `InterSample` data property describes how the algorithms should handle the input between samples. For example, you can specify the behavior between the samples to be piece-wise constant (zero-order hold, `zoh`) or linearly interpolated between the samples (first order hold, `foh`). The transformation formulas for `c2d` and `d2c` are affected by the intersample behavior of the input.

By default, `c2d` and `d2c` use the intersample behavior you assigned to the estimation data. To override this setting during transformation, add an extra argument in the syntax. For example:

```
% Set first-order hold intersample behavior
mod_d = c2d(mod_c,T,'foh')
```

Effects on the Noise Model

`c2d`, `d2c`, and `d2d` change the sampling interval of both the dynamic model and the noise model. Resampling a model affects the variance of its noise model.

A parametric noise model is a time-series model with the following mathematical description:

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

$$Ee^2 = \lambda$$

The noise spectrum is computed by the following discrete-time equation:

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

where λ is the variance of the white noise *e(t)*, and λT represents the spectral density of *e(t)*. Resampling the noise model preserves the spectral density λT. The spectral density λT is invariant up to the Nyquist frequency. For more information about spectrum normalization, see "Spectrum Normalization" on page 3-14.

`d2d` resampling of the noise model affects simulations with noise using `sim`. If you resample a model to a faster sampling rate, simulating this model results in higher noise level. This higher noise level results from the underlying continuous-time model being subject to continuous-time white noise disturbances, which have infinite, instantaneous variance. In this case, the *underlying continuous-time model* is the unique representation for discrete-time models. To maintain the same level of noise after interpolating the noise signal, scale the noise spectrum by $\sqrt{T_{New}/T_{Old}}$, where T_{new} is the new sampling interval and T_{old} is the original sampling interval. before applying `sim`.

Continuous-Discrete Conversion Methods

In this section...

Choosing a Conversion Method

The `c2d` command discretizes continuous-time models. Conversely, `d2c` converts discrete-time models to continuous time. Both commands support several discretization and interpolation methods, as shown in the following table.

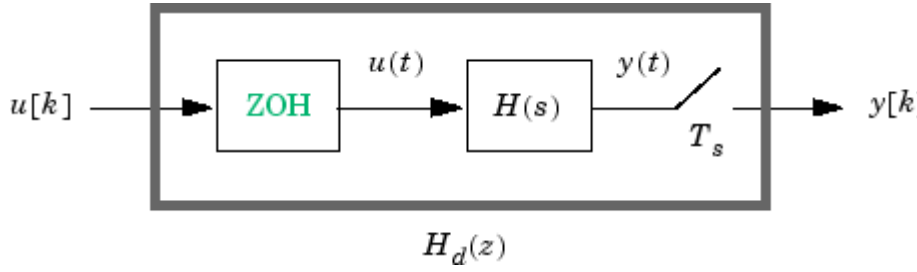
Discretization Method	Use when:
"Zero-Order Hold" on page 3-144	You want an exact discretization in the time domain for staircase inputs.
"First-Order Hold" on page 3-146	You want an exact discretization in the time domain for piecewise linear inputs.
"Impulse-Invariant Mapping" on page 3-147 (`c2d` only)	You want an exact discretization in the time domain for impulse train inputs.

Discretization Method	Use when:
"Tustin Approximation" on page 3-148	• You want good matching in the frequency domain between the continuous- and discrete-time models. • Your model has important dynamics at some particular frequency.
"Zero-Pole Matching Equivalents" on page 3-151	You have a SISO model, and you want good matching in the frequency domain between the continuous- and discrete-time models.

Zero-Order Hold

The Zero-Order Hold (ZOH) method provides an exact match between the continuous- and discrete-time systems in the time domain for staircase inputs.

The following block diagram illustrates the zero-order-hold discretization $H_d(z)$ of a continuous-time linear model $H(s)$



The ZOH block generates the continuous-time input signal $u(t)$ by holding each sample value $u(k)$ constant over one sample period:

$$u(t) = u[k], \qquad kT_s \le t \le (k+1)T_s$$

The signal $u(t)$ is the input to the continuous system $H(s)$. The output $y[k]$ results from sampling $y(t)$ every T_s seconds.

Conversely, given a discrete system $H_d(z)$, `d2c` produces a continuous system $H(s)$. The ZOH discretization of $H(s)$ coincides with $H_d(z)$.

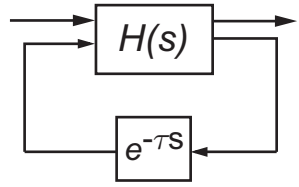
The ZOH discrete-to-continuous conversion has the following limitations:

- `d2c` cannot convert LTI models with poles at $z = 0$.
- For discrete-time LTI models having negative real poles, ZOH `d2c` conversion produces a continuous system with higher order. The model order increases because a negative real pole in the z domain maps to a pure imaginary value in the s domain. Such mapping results in a continuous-time model with complex data. To avoid this, the software instead introduces a conjugate pair of complex poles in the s domain.

ZOH Method for Systems with Time Delays

You can use the ZOH method to discretize SISO or MIMO continuous-time models with time delays. The ZOH method yields an exact discretization for systems with input delays, output delays, or transfer delays.

For systems with internal delays (delays in feedback loops), the ZOH method results in approximate discretizations. The following figure illustrates a system with an internal delay.



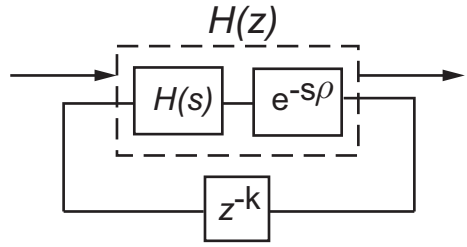
For such systems, `c2d` performs the following actions to compute an approximate ZOH discretization:

1 Decomposes the delay τ as $\tau = kT_s + \rho$ with $0 \le \rho < T_s$.

2 Absorbs the fractional delay ρ into $H(s)$.

3 Discretizes $H(s)$ to $H(z)$.

4 Represents the integer portion of the delay kT_s as an internal discrete-time delay z^{-k}. The final discretized model appears in the following figure:



First-Order Hold

The First-Order Hold (FOH) method provides an exact match between the continuous- and discrete-time systems in the time domain for piecewise linear inputs.

FOH differs from ZOH by the underlying hold mechanism. To turn the input samples $u[k]$ into a continuous input $u(t)$, FOH uses linear interpolation between samples:

$$u(t) = u[k] + \frac{t - kT_s}{T_s}\big(u[k+1] - u[k]\big), \quad kT_s \le t \le (k+1)T_s$$

This method is generally more accurate than ZOH for systems driven by smooth inputs.

This FOH method differs from standard causal FOH and is more appropriately called *triangle approximation* (see [2], p. 228). The method is also known as ramp-invariant approximation.

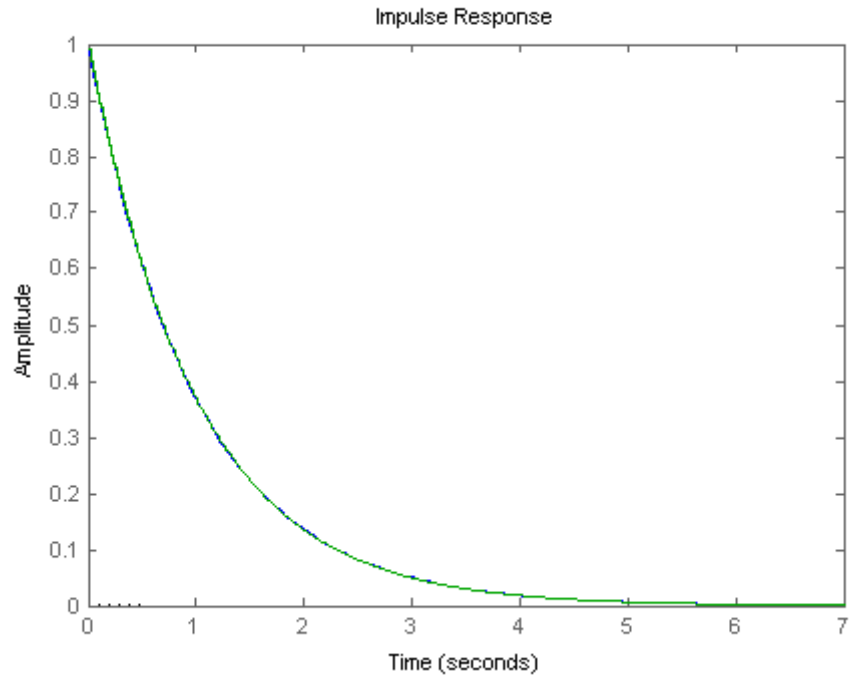
FOH Method for Systems with Time Delays

You can use the FOH method to discretize SISO or MIMO continuous-time models with time delays. The FOH method handles time delays in the same way as the ZOH method. See "ZOH Method for Systems with Time Delays" on page 3-145.

Impulse-Invariant Mapping

The impulse-invariant mapping produces a discrete-time model with the same impulse response as the continuous time system. For example, compare the impulse response of a first-order continuous system with the impulse-invariant discretization:

```
G = tf(1,[1,1]);
Gd1 = c2d(G,0.01,'impulse');
impulse(G,Gd1)
```



The impulse response plot shows that the impulse responses of the continuous and discretized systems match.

Impulse-Invariant Mapping for Systems with Time Delays

You can use impulse-invariant mapping to discretize SISO or MIMO continuous-time models with time delay, except that the method does not support `ss` models with internal delays. For supported models, impulse-invariant mapping yields an exact discretization of the time delay.

Tustin Approximation

The Tustin or bilinear approximation yields the best frequency-domain match between the continuous-time and discretized systems. This method relates the s-domain and z-domain transfer functions using the approximation:

$$z = e^{sT_s} \approx \frac{1 + sT_s/2}{1 - sT_s/2}.$$

In `c2d` conversions, the discretization $H_d(z)$ of a continuous transfer function $H(s)$ is:

$$H_d(z) = H(s'), \qquad s' = \frac{2}{T_s}\frac{z-1}{z+1}$$

Similarly, the `d2c` conversion relies on the inverse correspondence

$$H(s) = H_d(z'), \qquad z' = \frac{1 + sT_s/2}{1 - sT_s/2}$$

When you convert a state-space model using the Tustin method, the states are not preserved. The state transformation depends upon the state-space matrices and whether the system has time delays. For example, for an explicit ($E = I$) continuous-time model with no time delays, the state vector $w[k]$ of the discretized model is related to the continuous-time state vector $x(t)$ by:

$$w[kT_s] = \left(I - A\frac{T_s}{2}\right)x(kT_s) - \frac{T_s}{2}Bu(kT_s) = x(kT_s) - \frac{T_s}{2}\left(Ax(kT_s) + Bu(kT_s)\right).$$

T_s is the sampling time of the discrete-time model. A and B are state-space matrices of the continuous-time model.

Tustin Approximation with Frequency Prewarping

If your system has important dynamics at a particular frequency that you want the transformation to preserve, you can use the Tustin method with frequency prewarping. This method ensures a match between the continuous- and discrete-time responses at the prewarp frequency.

The Tustin approximation with frequency prewarping uses the following transformation of variables:

$$H_d(z) = H(s'), \qquad s' = \frac{\omega}{\tan(\omega T_s/2)}\frac{z-1}{z+1}$$

This change of variable ensures the matching of the continuous- and discrete-time frequency responses at the prewarp frequency ω, because of the following correspondence:

$$H(j\omega) = H_d\left(e^{j\omega T_s}\right)$$

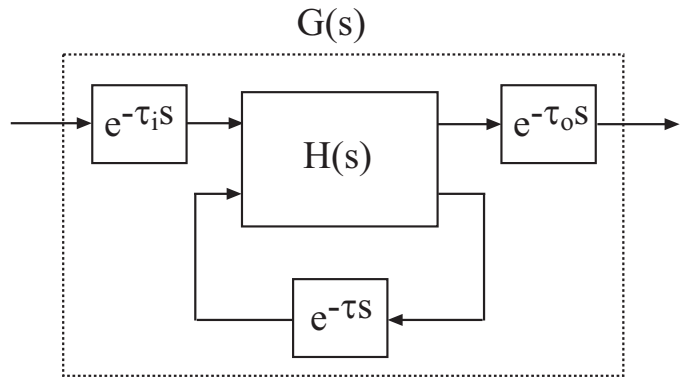
Tustin Approximation for Systems with Time Delays

You can use the Tustin approximation to discretize SISO or MIMO continuous-time models with time delays.

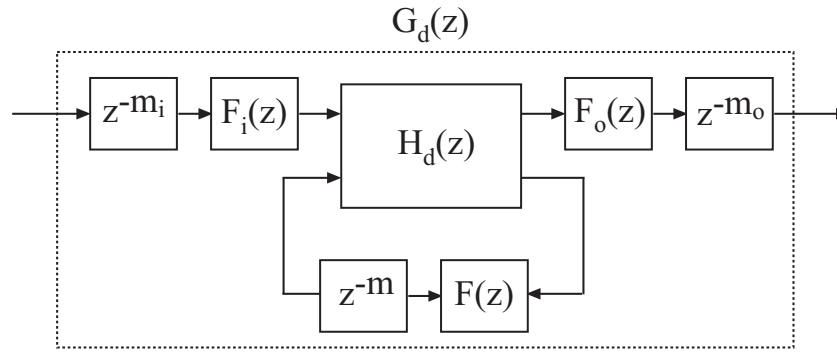
By default, the Tustin method rounds any time delay to the nearest multiple of the sampling time. Therefore, for any time delay `tau`, the integer portion of the delay, `k*Ts`, maps to a delay of `k` sampling periods in the discretized model. This approach ignores the residual fractional delay, `tau - k*Ts`.

You can to approximate the fractional portion of the delay by a discrete all-pass filter (Thiran filter) of specified order. To do so, use the `FractDelayApproxOrder` option of `c2dOptions`.

To understand how the Tustin method handles systems with time delays, consider the following SISO state-space model $G(s)$. The model has input delay τ_i, output delay τ_o, and internal delay τ.



The following figure shows the general result of discretizing $G(s)$ using the Tustin method.



By default, `c2d` converts the time delays to pure integer time delays. The `c2d` command computes the integer delays by rounding each time delay to the nearest multiple of the sample time T_s. Thus, in the default case, $m_i = \text{round}(\tau_i/T_s)$, $m_o = \text{round}(\tau_o/T_s)$, and $m = \text{round}(\tau/T_s)$.. Also in this case, $F_i(z) = F_o(z) = F(z) = 1$.

If you set `FractDelayApproxOrder` to a non-zero value, `c2d` approximates the fractional portion of the time delays by Thiran filters $F_i(z)$, $F_o(z)$, and $F(z)$.

The Thiran filters add additional states to the model. The maximum number of additional states for each delay is `FractDelayApproxOrder`.

For example, for the input delay τ_i, the order of the Thiran filter $F_i(z)$ is:

$$\text{order}(F_i(z)) = \max(\text{ceil}(\tau_i/T_s), \texttt{FractDelayApproxOrder}).$$

If $\text{ceil}(\tau_i/T_s) < \texttt{FractDelayApproxOrder}$, the Thiran filter $F_i(z)$ approximates the entire input delay τ_i. If $\text{ceil}(\tau_i/T_s) > \texttt{FractDelayApproxOrder}$, the Thiran filter only approximates a portion of the input delay. In that case, `c2d` represents the remainder of the input delay as a chain of unit delays z^{-m_i}, where

$$m_i = \text{ceil}(\tau_i/T_s) - \texttt{FractDelayApproxOrder}.$$

`c2d` uses Thiran filters and `FractDelayApproxOrder` in a similar way to approximate the output delay τ_o and the internal delay τ.

When you discretize`tf` and `zpk` models using the Tustin method, `c2d` first aggregates all input, output, and transfer delays into a single transfer delay τ_{TOT} for each channel. `c2d` then approximates τ_{TOT} as a Thiran filter and a chain of unit delays in the same way as described for each of the time delays in `ss` models.

For more information about Thiran filters, see the `thiran` reference page and [4].

Zero-Pole Matching Equivalents

The method of conversion by computing zero-pole matching equivalents applies only to SISO systems. The continuous and discretized systems have matching DC gains. Their poles and zeros are related by the transformation:

$$z_i = e^{s_i T_s}$$

where:

- z_i is the ith pole or zero of the discrete-time system.
- s_i is the ith pole or zero of the continuous-time system.
- T_s is the sampling time.

See [2] for more information.

Zero-Pole Matching for Systems with Time Delays

You can use zero-pole matching to discretize SISO continuous-time models with time delay, except that the method does not support `ss` models with internal delays. The zero-pole matching method handles time delays in the same way as the Tustin approximation. See "Tustin Approximation for Systems with Time Delays" on page 3-149.

References

[1] Åström, K.J. and B. Wittenmark, *Computer-Controlled Systems: Theory and Design*, Prentice-Hall, 1990, pp. 48-52.

[2] Franklin, G.F., Powell, D.J., and Workman, M.L., *Digital Control of Dynamic Systems* (3rd Edition), Prentice Hall, 1997.

[3] Smith, J.O. III, "Impulse Invariant Method", *Physical Audio Signal Processing*, August 2007. `http://www.dsprelated.com/dspbooks/pasp/Impulse_Invariant_Method.html`.

[4] T. Laakso, V. Valimaki, "Splitting the Unit Delay", *IEEE Signal Processing Magazine*, Vol. 13, No. 1, p.30-60, 1996.

See Also

`c2d` | `d2c` | `c2dOptions` | `d2cOptions` | `d2d` | `thiran`

Effect of Input Intersample Behavior on Continuous-Time Models

The intersample behavior of the input signals influences the estimation, simulation and prediction of continuous-time models. A sampled signal is characterized only by its values at the sampling instants. However, when you apply a continuous-time input to a continuous-time system, the output values at the sampling instants depend on the inputs at the sampling instants and on the inputs between these points.

The `iddata` and `idfrd` objects have an `InterSample` property which stores how the input behaves between the sampling instants. You can specify the behavior between the samples to be piecewise constant (zero-order hold), linearly interpolated between the samples (first-order hold) or band-limited. A band-limited intersample behavior of the input signal means:

- A filtered input signal (an input of finite bandwidth) was used to excite the system dynamics
- The input was measured using a sampling device (A/D converter with antialiasing) that reported it to be band-limited even though the true input entering the system was piecewise constant or linear. In this case, the sampling devices can be assumed to be a part of the system being modeled.

When input signal is band-limited, the estimation is performed as follows:

- Time-domain data is converted into frequency domain data using `fft` and the sample time of the data is set to zero.
- Discrete-time frequency domain data (`IDDATA` with domain = 'frequency' or `IDFRD` with sample time $Ts \neq 0$) is treated as continuous-time data by setting the sample time Ts to zero.

The resulting continuous-time frequency domain data is used for model estimation. For more information, see Pintelon, R. and J. Schoukens, *System Identification. A Frequency Domain Approach*, section 10.2, pp-352-356,Wiley-IEEE Press, New York, 2001.

Similarly, the intersample behavior of the input data affects the results of simulation and prediction of continuous-time models. `sim` and `predict`

commands use the `InterSample` property to choose the right algorithm for computing model response. Say more or give some pointers about which algorithms are used?

The following example simulates a system using first-order hold (`foh`) intersample behavior for input signal.

```
sys=idtf([-1 -2],[1 2 1 0.5]);
rng('default')
u = idinput([100 1 5],'sine',[],[],[5 10 1]);
Ts = 2;
y = lsim(sys, u, (0:Ts:999)', 'foh');
```

Create an `iddata` object for the simulated input-output data:

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

The default intersample behavior is zero-order hold (`zoh`):

```
data.InterSample
```

```
ans =

zoh
```

Estimate a transfer function using this data:

```
np = 3; % number of poles
nz = 1; % number of zeros
opt = tfestOptions('InitMethod','all','Display','on');
opt.SearchOption.MaxIter = 100;
modelZOH = tfest(data,np,nz,opt);
```

The model gives about 80% fit to data. The sample time of the data is large enough that intersample inaccuracy (using `zoh` rather than `foh`) leads to significant modeling errors.

Re-estimate the model using `foh` intersample behavior:

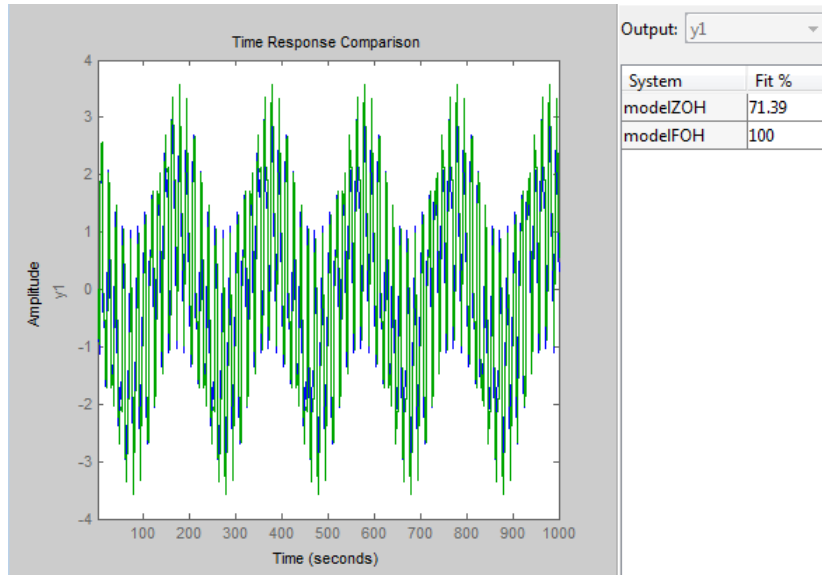
```
data.InterSample = 'foh';
modelFOH = tfest(data, np, nz,opt);
```

`modelFOH` is able to retrieve the original system correctly:

Compare the model outputs with data.

```
compare(data, modelZOH, modelFOH)
```

`modelZOH` is compared to data whose intersample behavior is `foh`. There, its fit decreases to around 70%.



Transforming Between Linear Model Representations

You can transform linear models between state-space and polynomial forms. You can also transform between frequency-response, state-space, and polynomial forms.

If you used the System Identification Tool GUI to estimate models, you must export the models to the MATLAB workspace before converting models.

For detailed information about each command in the following table, see the corresponding reference page.

Commands for Transforming Model Representations

Command	Model Type to Convert	Usage Example

`idfrd`	Converts any linear model to an `idfrd` model.
	If you have the Control System Toolbox product, this command converts any numeric LTI model too.
		To get frequency response of `m` at default frequencies, use the following command:
		
		`m_f = idfrd(m)`
		
		To get frequency response at specific frequencies, use the following command:
		
		`m_f = idfrd(m,f)`
		
		To get frequency response for a submodel from input `2` to output `3`, use the following command:
		
		`m_f = idfrd(m(2,3))`

`idpoly`	Converts any linear identified model, except `idfrd`, to ARMAX representation if the original model has a nontrivial noise component, or OE if the noise model is trivial (`H = 1`). If you have the Control System Toolbox product,	To get an ARMAX model from state-space model `m_ss`, use the following command: `m_p = idpoly(m_ss)`

Commands for Transforming Model Representations (Continued)

Command	Model Type to Convert	Usage Example
	this command converts any numeric LTI model, except `frd`.	
`idss`	Converts any linear identified model, except `idfrd`, to state-space representation. If you have the Control System Toolbox product, this command converts any numeric LTI model, except `frd`.	To get a state-space model from an ARX model `m_arx`, use the following command: `m_ss = idss(m_arx)`
`idtf`	Converts any linear identified model, except `idfrd`, to transfer function representation. The noise component of the original model is lost since an `idtf` object has no elements to model noise dynamics. If you have the Control System Toolbox product, this command converts any numeric LTI model, except `frd`.	To get a transfer function from a state-space model `m_ss`, use the following command: `m_tf = idtf(m_ss)`

Note Most transformations among identified models (among `idss`, `idtf`, `idpoly`) causes the parameter covariance information to be lost, with few exceptions:

- Conversion of an `idtf` model to an `idpoly` model.
- Conversion of an `idgrey` model to an `idss` model.

If you want to translate the estimated parameter covariance during conversion, use `translatecov`.

Subreferencing Models

In this section...

What Is Subreferencing?

You can use subreferencing to create models with subsets of inputs and outputs from existing multivariable models. Subreferencing is also useful when you want to generate model plots for only certain channels, such as when you are exploring multiple-output models for input channels that have minimal effect on the output.

The toolbox supports subreferencing operations for `idtf`, `idpoly`, `idproc`, `idss`, and `idfrd` model objects.

Subreferencing is not supported for `idgrey` models. If you want to analyze the sub-model, convert it into an `idss` model first, and then subreference the I/Os of the `idss` model. If you want a grey-box representation of a subset of I/Os, create a new `idgrey` model that uses an ODE function returning the desired I/O dynamics.

In addition to subreferencing the model for specific combinations of measured inputs and output, you can subreference dynamic and noise models individually.

Limitation on Supported Models

Subreferencing nonlinear models is not supported.

Subreferencing Specific Measured Channels

Use the following general syntax to subreference specific input and output channels in models:

```
model(outputs,inputs)
```

In this syntax, `outputs` and `inputs` specify channel indexes or channel names.

To select all output or all input channels, use a colon (`:`). To select no channels, specify an empty matrix (`[]`). If you need to reference several channel names, use a cell array of strings.

For example, to create a new model `m2` from `m` from inputs `1` (`'power'`) and `4` (`'speed'`) to output number `3` (`'position'`), use either of the following equivalent commands:

```
m2 = m('position',{'power','speed'})
```

or

```
m2 = m(3,[1 4])
```

For a single-output model, you can use the following syntax to subreference specific input channels without ambiguity:

```
m3 = m(inputs)
```

Similarly, for a single-input model, you can use the following syntax to subreference specific output channels:

```
m4 = m(outputs)
```

Separation of Measured and Noise Components of Models

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

$$y = Gu + He$$

G is an operator that takes the measured inputs u to the outputs and captures the system dynamics.

H is an operator that describes the properties of the additive output disturbance and takes the hypothetical (unmeasured) noise source inputs to the outputs. H represents the noise model. When you specify to estimate a noise model, the resulting model include one noise channel e at the input for each output in your system.

Thus, linear, parametric models represent input-output relationships for two kinds of input channels: measured inputs and (unmeasured) noise inputs. For example, consider the ARX model given by one of the following equations:

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

or

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

In this case, the dynamic model is the relationship between the measured input u and output y, $G = {B(q)}/{A(q)}$. The noise model is the contribution of the input noise e to the output y, given by $H = {1}/{A(q)}$.

Suppose that the model `m` contains both a dynamic model G and a noise model H. To create a new model that only has G and no noise contribution, simply set its `NoiseVariance` property value to zero value.

To create a new model by subreferencing H due to unmeasured inputs, use the following syntax:

```
m_H = m(:,[])
```

This operation creates a time-series model from `m` by ignoring the measured input.

The covariance matrix of e is given by the model property `NoiseVariance`, which is the matrix Λ:

$$\Lambda = LL^T$$

The covariance matrix of e is related to v, as follows:

$$e = Lv$$

where v is white noise with an identity covariance matrix representing independent noise sources with unit variances.

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'`.

If you want to create a model that has only the noise channels of an identified model as its measured inputs, use the `noise2meas` command. It results in a model with $y(t) = He$ or $y(t) = HLv$, where e or v is treated as a measured input.

Note When you plot models in the 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 12-16.

Concatenating Models

In this section...

About Concatenating Models

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

When you concatenate identified models, such as `idtf`, `idpoly`, `idproc`, and `idss` model objects, the resulting model combines the parameters of the individual models. However, the estimated parameter covariance is lost. If you want to translate the covariance information during concatenation, use `translatecov`.

Concatenation is not supported for `idgrey` models; convert them to `idss` models first if you want to perform concatenation.

You can also concatenate nonparametric models, which contain the estimated impulse-response (`idtf` object) and frequency-response (`idfrd` object) of a system.

In case of `idfrd` models, concatenation combines information in the `ResponseData` properties of the individual model objects. `ResponseData` is an `ny`-by-`nu`-by-`nf` array that stores the response of the system, where `ny` is the number of output channels, `nu` is the number of input channels, and `nf` is the number of frequency values. The `(j,i,:)` vector of the resulting response data represents the frequency response from the `i`th input to the `j`th output at all frequencies.

Limitation on Supported Models

Concatenation is supported for linear models only.

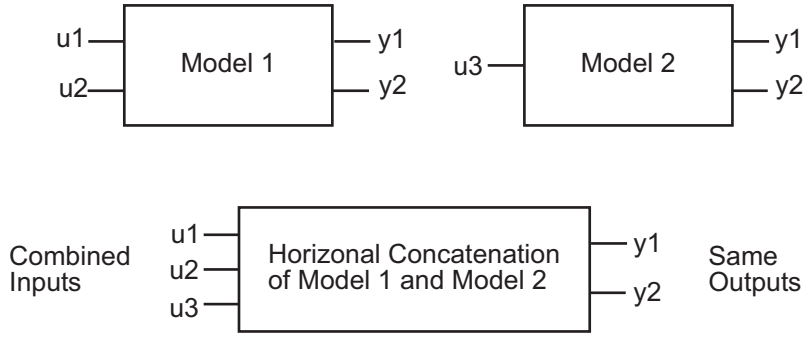
Horizontal Concatenation of Model Objects

Horizontal concatenation of model objects requires that they have the same outputs. If the output channel names are different and their dimensions are the same, the concatenation operation resets the output names to their default values.

The following syntax creates a new model object `m` that contains the horizontal concatenation of `m1,m2,...,mN`:

```
m = [m1,m2,...,mN]
```

`m` takes all of the inputs of `m1,m2,...,mN` to the same outputs as in the original models. The following diagram is a graphical representation of horizontal concatenation of the models.



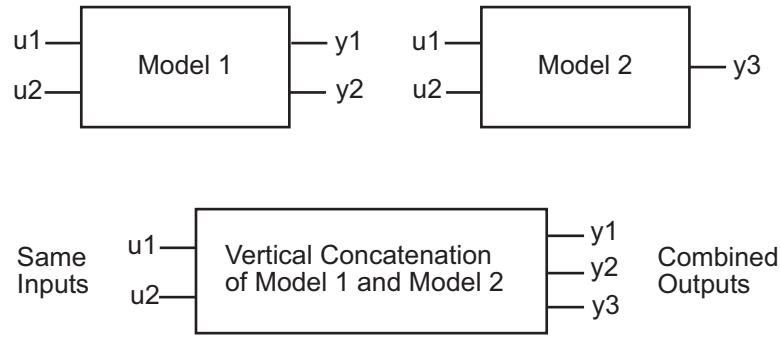
Vertical Concatenation of Model Objects

Vertical concatenation combines output channels of specified models. Vertical concatenation of model objects requires that they have the same inputs. If the input channel names are different and their dimensions are the same, the concatenation operation resets the input channel names to their default (`''`) values.

The following syntax creates a new model object `m` that contains the vertical concatenation of `m1,m2,...,mN`:

```
m = [m1;m2;... ;mN]
```

`m` takes the same inputs in the original models to all of the output of `m1,m2,...,mN`. The following diagram is a graphical representation of vertical concatenation of frequency-response data.



Concatenating Noise Spectrum Data of idfrd Objects

When `idfrd` models are obtained as a result of estimation (such as using `spa`), the `SpectrumData` property is not empty and contains the power spectra and cross spectra of the output noise in the system. For each output channel, this toolbox estimates one noise channel to explain the difference between the output of the model and the measured output.

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

In case of horizontal concatenation, there is no meaningful way to combine the `SpectrumData` of individual `idfrd` objects, and the resulting `SpectrumData` property is empty. An empty property results because each `idfrd` object has its own set of noise channels, where the number of noise channels equals the number of outputs. When the resulting `idfrd` object contains the same output channels as each of the individual `idfrd` objects, it cannot accommodate the noise data from all the `idfrd` objects.

In case of vertical concatenation, this toolbox concatenates individual noise models diagonally. The following shows that `m.SpectrumData` is a block diagonal matrix of the power spectra and cross spectra of the output noise in the system:

$$m.s = \begin{pmatrix} m1.s & & 0 \\ & \ddots & \\ 0 & & mN.s \end{pmatrix}$$

`s` in `m.s` is the abbreviation for the `SpectrumData` property name.

See Also

If you have the Control System Toolbox product, see "Combining Model Objects" on page 10-5 about additional functionality for combining models.

Merging Models

You can merge models of the same structure to obtain a single model with parameters that are statistically weighed means of the parameters of the individual models. When computing the merged model, the covariance matrices of the individual models determine the weights of the parameters.

You can perform the merge operation for the `idtf`, `idgrey`, `idpoly`, `idproc`, and `idss` model objects.

Note Each merge operation merges the same type of model object.

Merging models is an alternative to merging data sets into a single multiexperiment data set, and then estimating a model for the merged data. Whereas merging data sets assumes that the signal-to-noise ratios are about the same in the two experiments, merging models allows greater variations in model uncertainty, which might result from greater disturbances in an experiment.

When the experimental conditions are about the same, merge the data instead of models. This approach is more efficient and typically involves better-conditioned calculations. For more information about merging data sets into a multiexperiment data set, see “Creating Multiexperiment Data at the Command Line” on page 2-61.

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

Building and Estimating Process Models Using System Identification Toolbox™

This example shows how to build simple process models using System Identification Toolbox™. Techniques for creating these models and estimating their parameters using experimental data is described. This example requires Simulink.

Introduction

This example illustrates how to build simple process models often used in process industry. Simple, low-order continuous-time transfer functions are usually employed to describe process behavior. Such models are described by IDPROC objects which represent the transfer function in a pole-zero-gain form.

Process models are of the basic type 'Static Gain + Time Constant + Time Delay'. They may be represented as:

$$P(s) = K.e^{-T_d * s}.\frac{1 + T_z * s}{(1 + T_{p1} * s)(1 + T_{p2} * s)}$$

or as an integrating process:

$$P(s) = K.e^{-T_d * s}.\frac{1 + T_z * s}{s(1 + T_{p1} * s)(1 + T_{p2} * s)}$$

where the user can determine the number of real poles (0, 1, 2 or 3), as well as the presence of a zero in the numerator, the presence of an integrator term (`1/s`) and the presence of a time delay (`Td`). In addition, an underdamped (complex) pair of poles may replace the real poles.

Representation of Process Models using IDPROC Objects

IDPROC objects define process models by using the letters `P` (for process model), `D` (for time delay), `Z` (for a zero) and `I` (for integrator). An integer will denote the number of poles. The models are generated by calling `idproc` with a string identifier using these letters.

This should be clear from the following examples.

```
idproc('P1') % transfer function with only one pole (no zeros or delay)
idproc('P2DIZ') % model with 2 poles, delay integrator and delay
idproc('POID') % model with no poles, but an integrator and a delay
```

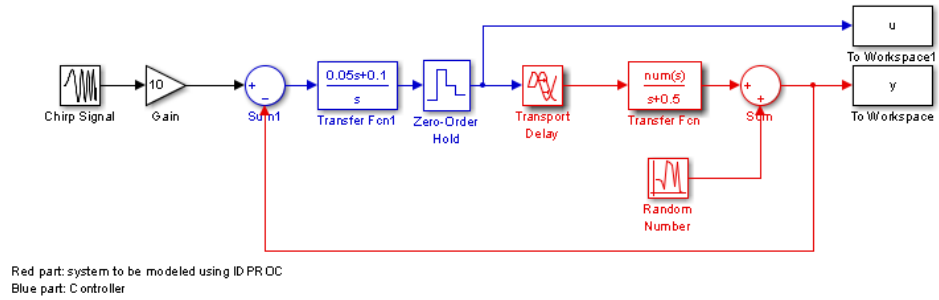
```
ans =
Process model with transfer function:
             Kp
  G(s) = ----------
          1+Tp1*s

        Kp = NaN
       Tp1 = NaN

Parameterization:
    'P1'
   Number of free coefficients: 2
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.

ans =
Process model with transfer function:
                   1+Tz*s
  G(s) = Kp * ------------------- * exp(-Td*s)
              s(1+Tp1*s)(1+Tp2*s)

         Kp = NaN
        Tp1 = NaN
        Tp2 = NaN
         Td = NaN
         Tz = NaN

Parameterization:
    'P2DIZ'
   Number of free coefficients: 5
   Use "getpvec", "getcov" for parameters and their uncertainties.
```

```
Status:
Created by direct construction or transformation. Not estimated.

ans =
Process model with transfer function:
          Kp
  G(s) = --- * exp(-Td*s)
         s

         Kp = NaN
         Td = NaN

Parameterization:
    'PODI'
   Number of free coefficients: 2
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

Creating an IDPROC Object (using a Simulink Model as Example)

Consider the system described by the following SIMULINK model:

```
open_system('iddempr1')
set_param('iddempr1/Random Number','seed','0')
```



The red part is the system, the blue part is the controller and the reference signal is a swept sinusoid (a chirp signal). The data sampling time is set to 0.5 seconds. As observed, the system is a continuous-time transfer function, and can hence be described using model objects in System Identification Toolbox, such as `idss`, `idpoly` or `idproc`.

Let us describe the system using `idpoly` and `idproc` objects. Using `idpoly` object, the system may be described as:

```
m0 = idpoly(1,0.1,1,1,[1 0.5],'Ts',0,'InputDelay',1.57,'NoiseVariance',0.01
```

The IDPOLY form used above is useful for describing transfer functions of arbitrary orders. Since the system we are considering here is quite simple (one pole and no zeros), and is continuous-time, we may use the simpler IDPROC object to capture its dynamics:

```
m0p = idproc('p1d','Kp',0.2,'Tp1',2,'Td',1.57) % one pole+delay, with initi
                                                % for gain, pole and delay s
```

```
m0p =
Process model with transfer function:
             Kp
  G(s) = ---------- * exp(-Td*s)
           1+Tp1*s

         Kp = 0.2
        Tp1 = 2
         Td = 1.57

Parameterization:
    'P1D'
   Number of free coefficients: 3
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

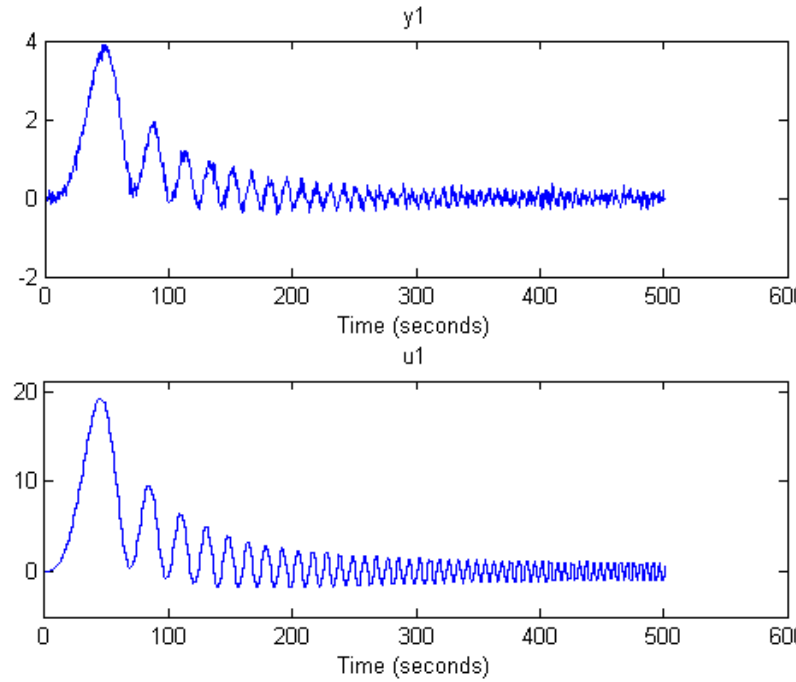
Estimating Parameters of IDPROC Models

Once a system is described by a model object, such as IDPROC, it may be used for estimation of its parameters using measurement data. As an example, we consider the problem of estimation of parameters of the Simulink model's system (red portion) using simulation data. We begin by acquiring data for estimation:

```
sim('iddempr1')
dat1e = iddata(y,u,0.5); % The IDDATA object for storing measurement data
```

Let us look at the data:

```
plot(dat1e)
```



We can identify a process model using `procest` command, by providing the same structure information specified to create IDPROC models. For example, the 1-pole+delay model may be estimated by calling `procest` as follows:

```
m1 = procest(dat1e,'p1d'); % estimation of idproc model using data 'dat1e'.

% Check the result of estimation:
m1


m1 =
Process model with transfer function:
             Kp
  G(s) = ---------- * exp(-Td*s)
          1+Tp1*s

        Kp = 0.20045
       Tp1 = 2.0431
        Td = 1.499

Parameterization:
    'P1D'
   Number of free coefficients: 3
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "dat1e".
Fit to estimation data: 87.34% (prediction focus)
FPE: 0.01068, MSE: 0.01063

To get information about uncertainties, use

present(m1)


m1 =
Process model with transfer function:
             Kp
  G(s) = ---------- * exp(-Td*s)
          1+Tp1*s

        Kp = 0.20045 +/- 0.00077275
       Tp1 = 2.0431 +/- 0.061216
        Td = 1.499 +/- 0.040854
```

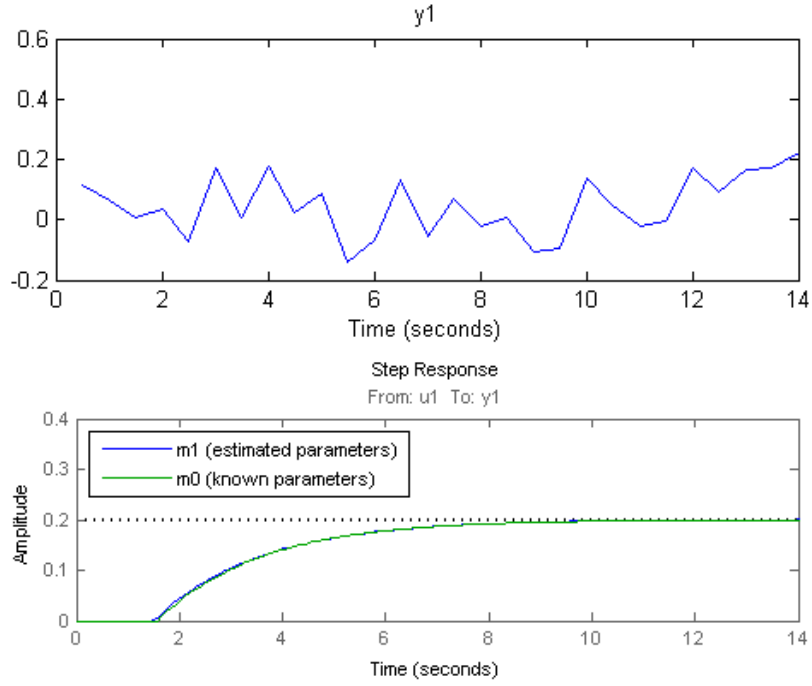
```
Parameterization:
    'P1D'
   Number of free coefficients: 3
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Termination condition: Near (local) minimum, (norm(g) < tol).
Number of iterations: 4, Number of function evaluations: 9

Estimated using PROCEST on time domain data "dat1e".
Fit to estimation data: 87.34% (prediction focus)
FPE: 0.01068, MSE: 0.01063
More information in model's "Report" property.
```

The model parameters, `K`, `Tp1` and `Td` are now shown with one standard deviation uncertainty range.

Computing Time and Frequency Response of IDPROC Models

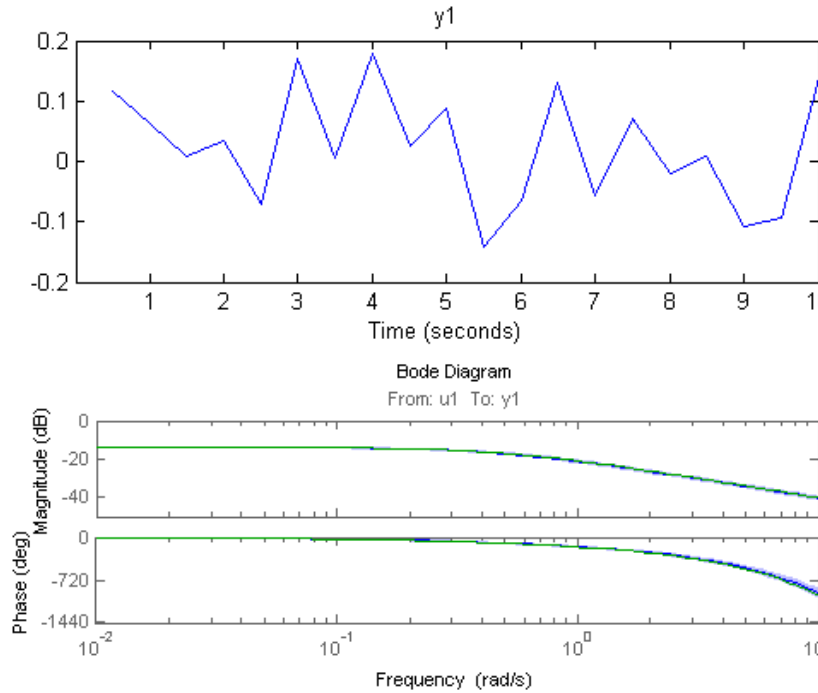
The model `m1` estimated above is an IDPROC model object to which all of the toolbox's model commands can be applied:

```
step(m1,m0) %step response of models m1 (estimated) and m0 (actual)
legend('m1 (estimated parameters)','m0 (known parameters)','location','nort
```



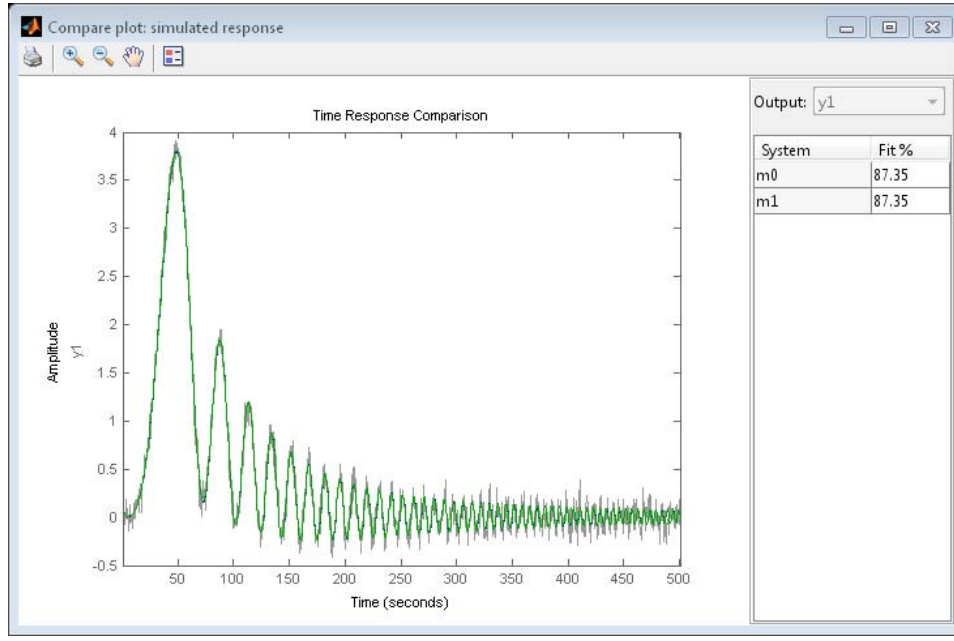
Bode response with confidence region corresponding to 5 s.d. may be computed by doing:

```
h = bodeplot(m1,m0);
showConfidence(h,3)
```



Similarly, the measurement data may be compared to the models outputs using `compare` as follows:

```
compare(dat1e,m0,m1)
```



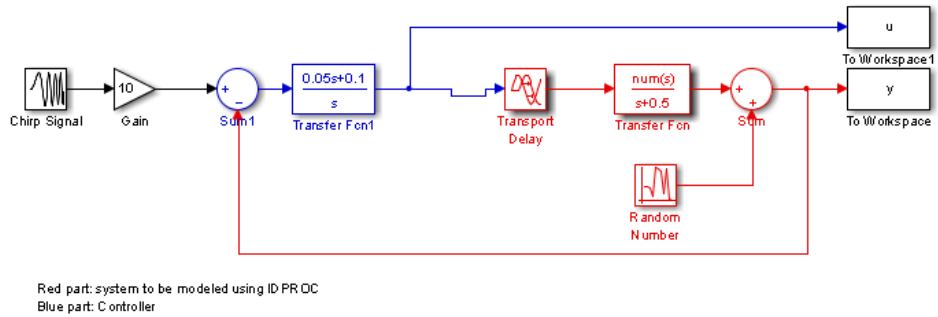
Other operations such as `sim`, `impulse`, `c2d` are also available, just as they are for other model objects.

```
bdclose('iddempr1')
```

Accommodating the Effect of Intersample Behavior in Estimation

It may be important (at least for slow sampling) to consider the intersample behavior of the input data. To illustrate this, let us study the same system as before, but without the sample-and-hold circuit:

```
open_system('iddempr5')
```



Simulate this system with the same sampling interval:

```
sim('iddempr5')
dat1f = iddata(y,u,0.5); % The IDDATA object for the simulated data
```

We estimate an IDPROC model using `data1f` while also imposing an upper bound on the allowable value delay. We will use 'lm' as search method and also choose to view the estimation progress.

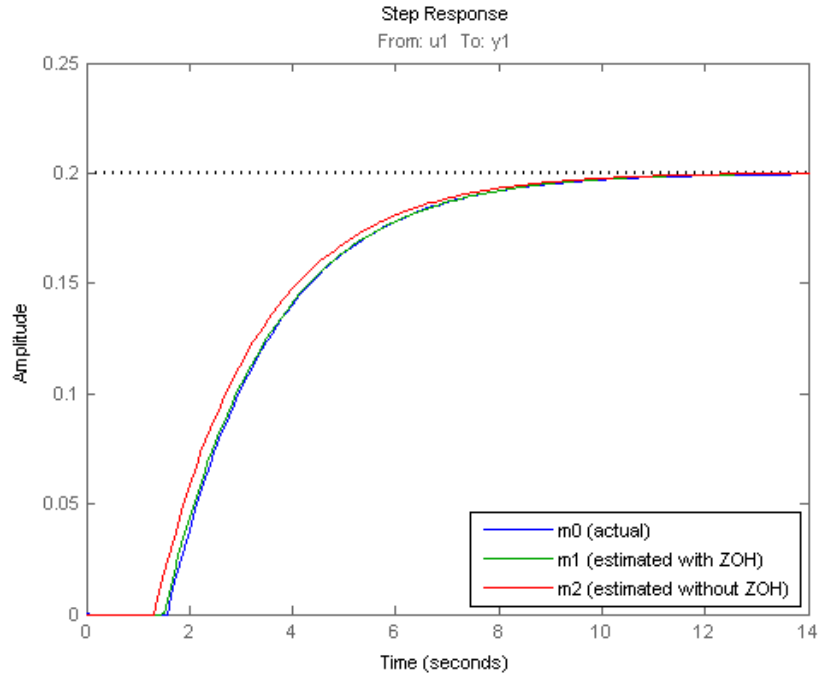
```
m2_init = idproc('P1D');
m2_init.Structure.Td.Maximum = 2;
opt = procestOptions('SearchMethod','lm','Display','on');
m2 = procest(dat1f,m2_init,opt);
m2
```

```
m2 =
Process model with transfer function:
             Kp
  G(s) = ---------- * exp(-Td*s)
           1+Tp1*s

         Kp = 0.20038
        Tp1 = 2.01
         Td = 1.31

Parameterization:
    'P1D'
```

```
   Number of free coefficients: 3
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "dat1f".
Fit to estimation data: 87.26% (prediction focus)
FPE: 0.01067, MSE: 0.01062
```

This model has a slightly less precise estimate of the delay than the previous one, m1:

```
[m0p.Td, m1.Td, m2.Td]
step(m0,m1,m2)
legend('m0 (actual)','m1 (estimated with ZOH)','m2 (estimated without ZOH)'


ans =

    1.5700    1.4990    1.3100
```



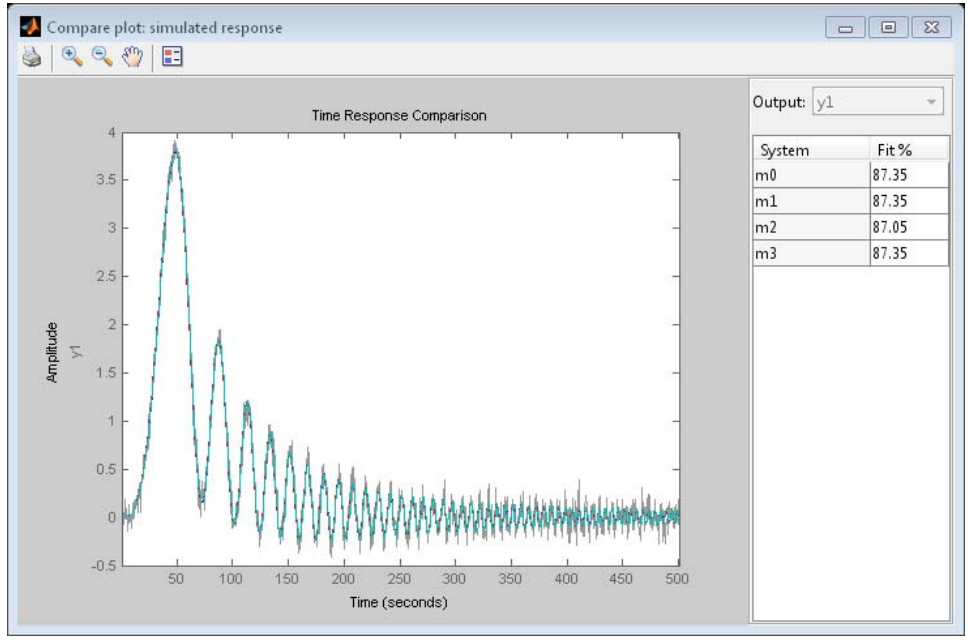
However, by telling the estimation process that the intersample behavior is first-order-hold (an approximation to the true continuous) input, we do better:

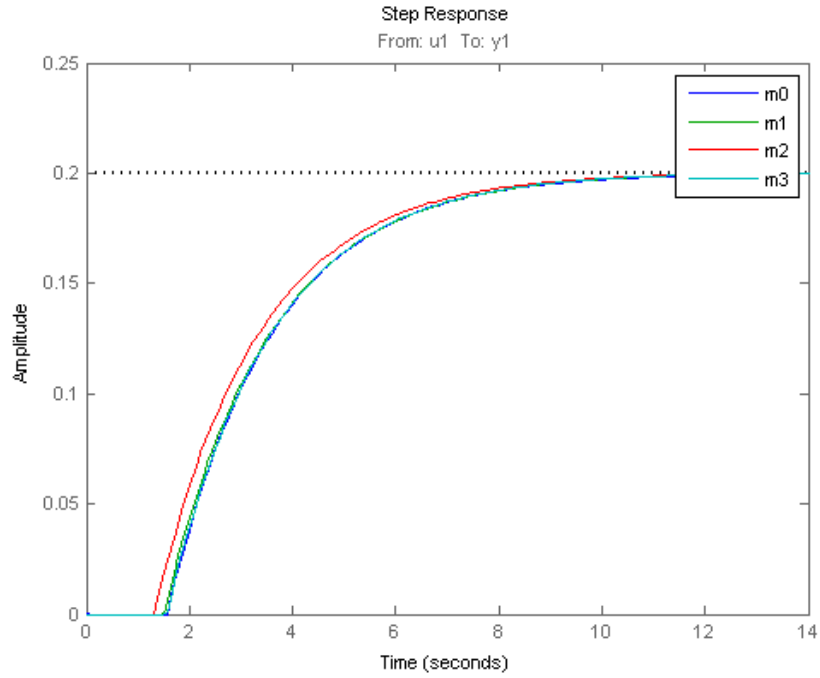
```
dat1f.InterSample = 'foh';
m3 = procest(dat1f,m2_init,opt);
```

Compare the four models m0 (true) m1 (obtained from zoh input) m2 (obtained for continuous input, with zoh assumption) and m3 (obtained for the same input, but with foh assumption)

```
[m0p.Td, m1.Td, m2.Td, m3.Td]
compare(dat1e,m0,m1,m2,m3)
step(m0,m1,m2,m3)
legend('m0','m1','m2','m3')
bdclose('iddempr5')
```

```
ans =

    1.5700    1.4990    1.3100    1.5570
```

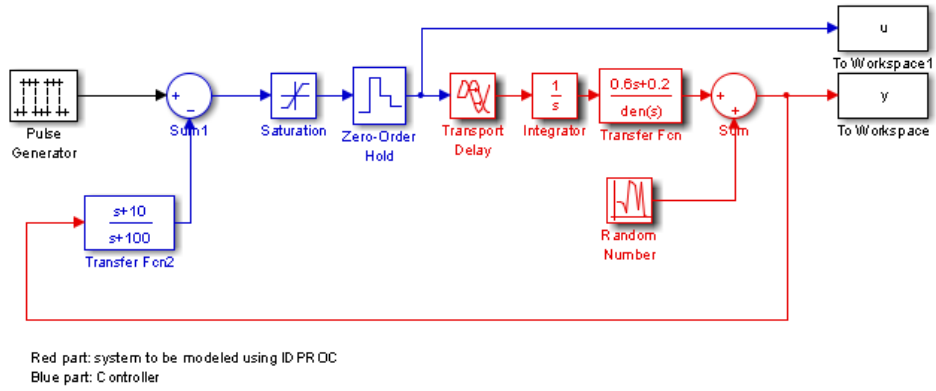




Modeling a System Operating in Closed Loop

Let us now consider a more complex process, with integration, that is operated in closed loop:

```
open_system('iddempr2')
```



The true system can be represented by:

```
m0 = idproc('P2ZDI','Kp',1,'Tp1',1,'Tp2',5,'Tz',3,'Td',2.2);
```

The process is controlled by a PD regulator with limited input amplitude and a zero order hold device. The sampling interval is 1 second.

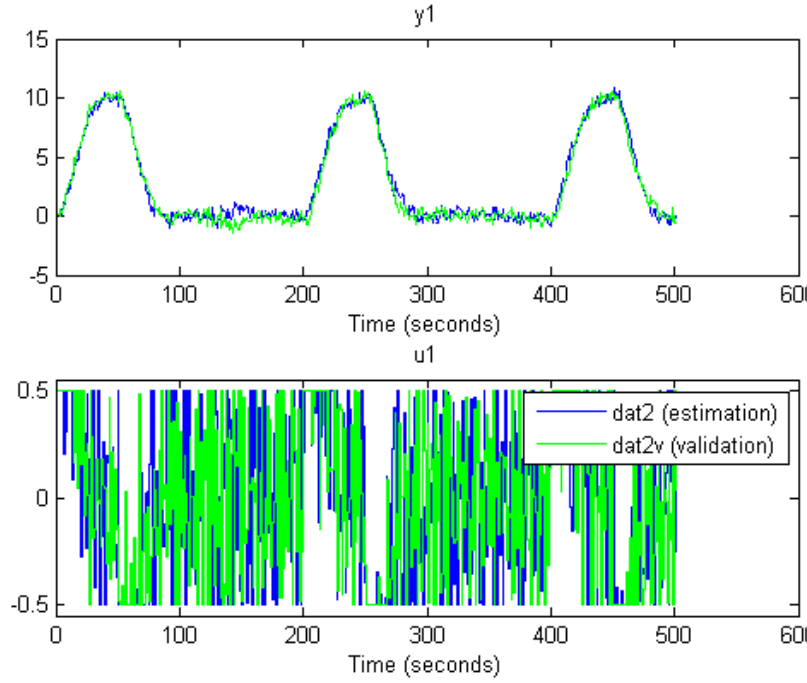
```
set_param('iddempr2/Random Number','seed','0')
sim('iddempr2')
dat2 = iddata(y,u,1); % IDDATA object for estimation
```

Two different simulations are made, the first for estimation and the second one for validation purposes.

```
set_param('iddempr2/Random Number','seed','13')
sim('iddempr2')
dat2v = iddata(y,u,1); % IDDATA object for validation purpose
```

Let us look at the data (estimation and validation).

```
plot(dat2,dat2v)
legend('dat2 (estimation)','dat2v (validation)')
```



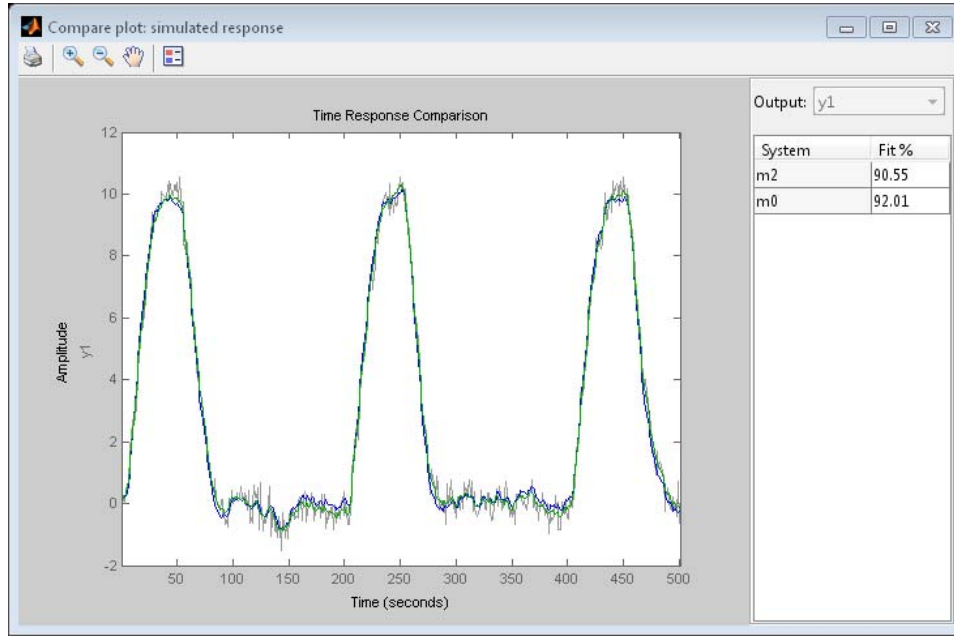
Let us now perform estimation using `dat2`.

```
m2_init = idproc('P2ZDI');
m2_init.Structure.Td.Maximum = 5;
m2_init.Structure.Tp1.Maximum = 2;
opt = procestOptions('SearchMethod','lm','Display','full');
opt.SearchOption.MaxIter  = 50;
m2 = procest(dat2, m2_init, opt)
```

```
m2 =
Process model with transfer function:
                  1+Tz*s
  G(s) = Kp * ------------------- * exp(-Td*s)
              s(1+Tp1*s)(1+Tp2*s)
```

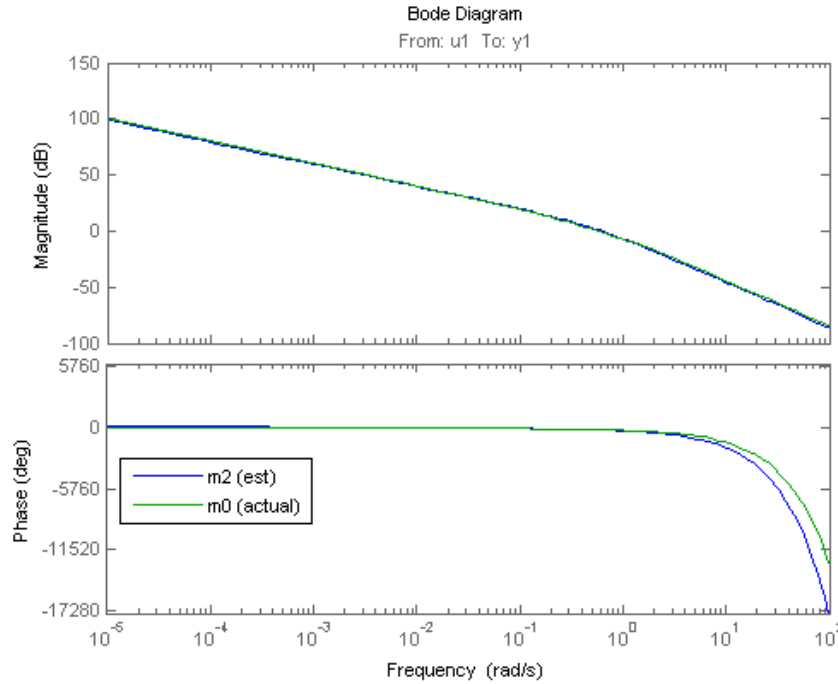
```
          Kp = -0.89555
         Tp1 = 2
         Tp2 = 1388.4
          Td = 3.08
          Tz = -1519.1

Parameterization:
    'P2DIZ'
   Number of free coefficients: 5
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "dat2".
Fit to estimation data: 90.21% (prediction focus)
FPE: 0.1651, MSE: 0.1457
```

```
compare(dat2v,m2,m0) % Gives very good agreement with data
```

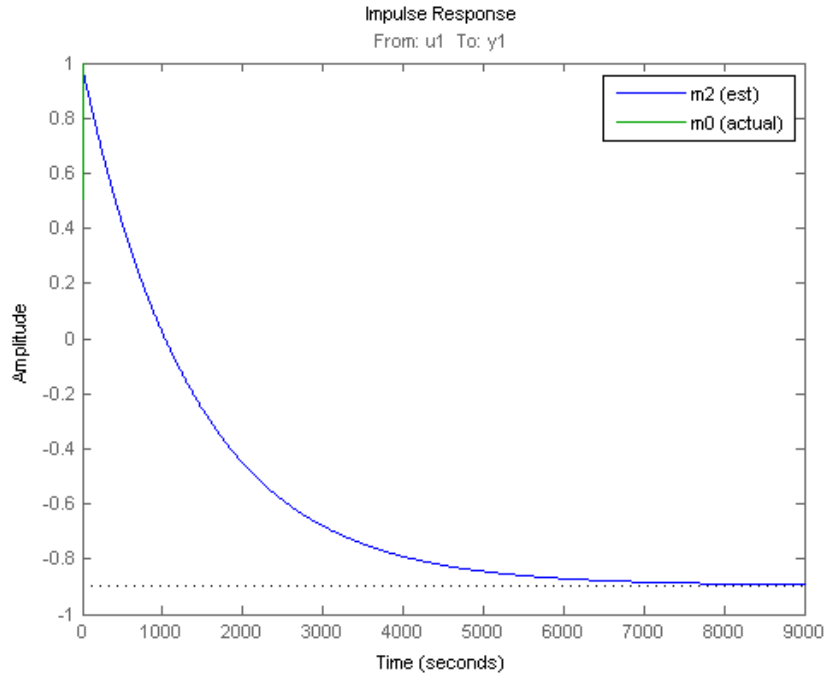



```
bode(m2,m0)
```

```
legend({'m2 (est)','m0 (actual)'},'location','west')
```




```
impulse(m2,m0)
legend({'m2 (est)','m0 (actual)'})
```



Compare also with the parameters of the true system:

```
present(m2)
[getpvec(m0), getpvec(m2)]
```

```
m2 =
Process model with transfer function:
                    1+Tz*s
  G(s) = Kp * ------------------- * exp(-Td*s)
              s(1+Tp1*s)(1+Tp2*s)

         Kp = -0.89555 +/- 22.268
        Tp1 = 2 +/- 0.25413
        Tp2 = 1388.4 +/- 16546
         Td = 3.08 +/- 0.20786
```

```
         Tz = -1519.1 +/- 19687

Parameterization:
    'P2DIZ'
   Number of free coefficients: 5
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Termination condition: Maximum number of iterations reached.
Number of iterations: 50, Number of function evaluations: 344

Estimated using PROCEST on time domain data "dat2".
Fit to estimation data: 90.21% (prediction focus)
FPE: 0.1651, MSE: 0.1457
More information in model's "Report" property.

ans =

   1.0e+03 *

    0.0010   -0.0009
    0.0010    0.0020
    0.0050    1.3884
    0.0022    0.0031
    0.0030   -1.5191
```

A word of caution. Identification of several real time constants may sometimes be an ill-conditioned problem, especially if the data are collected in closed loop.

To illustrate this, let us estimate a model based on the validation data:

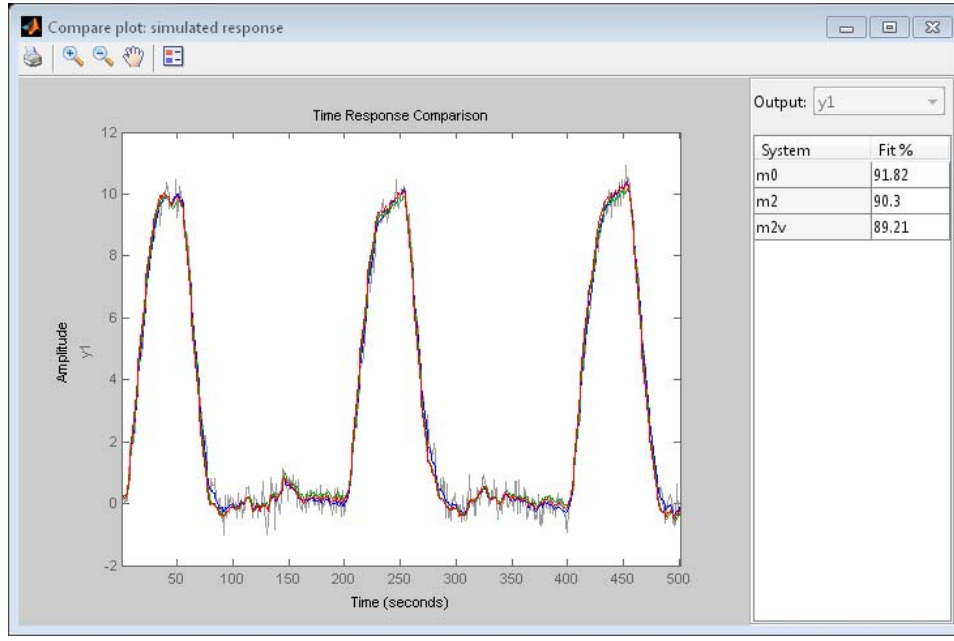
```
m2v = procest(dat2v, m2_init, opt)
[getpvec(m0), getpvec(m2), getpvec(m2v)]
```

```
m2v =
Process model with transfer function:
                  1+Tz*s
  G(s) = Kp * ------------------ * exp(-Td*s)
```

```
                s(1+Tp1*s)(1+Tp2*s)

           Kp = -2.8056
          Tp1 = 2
          Tp2 = 4586.1
           Td = 2.252
           Tz = -1630

Parameterization:
    'P2DIZ'
   Number of free coefficients: 5
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "dat2v".
Fit to estimation data: 89.61% (prediction focus)
FPE: 0.1897, MSE: 0.1675

ans =

   1.0e+03 *

    0.0010   -0.0009   -0.0028
    0.0010    0.0020    0.0020
    0.0050    1.3884    4.5861
    0.0022    0.0031    0.0023
    0.0030   -1.5191   -1.6300
```

This model has much worse parameter values. On the other hand, it performs nearly identically to the true system m0 when tested on the other data set dat2:

```
compare(dat2,m0,m2,m2v)
```



Fixing Known Parameters During Estimation

Suppose we know from other sources that one time constant is 1:

```
m2v.Structure.Tp1.Value = 1;
m2v.Structure.Tp1.Free = false;
```

We can fix this value, while estimating the other parameters:

```
m2v = procest(dat2v,m2v)
%
```

```
m2v =
Process model with transfer function:
                    1+Tz*s
  G(s) = Kp * ------------------- * exp(-Td*s)
              s(1+Tp1*s)(1+Tp2*s)
```

```
         Kp = -8.4621
        Tp1 = 1
        Tp2 = 13964
         Td = 4.201
         Tz = -1630.2

Parameterization:
    'P2DIZ'
   Number of free coefficients: 4
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "dat2v".
Fit to estimation data: 90.74% (prediction focus)
FPE: 0.1372, MSE: 0.1332
```

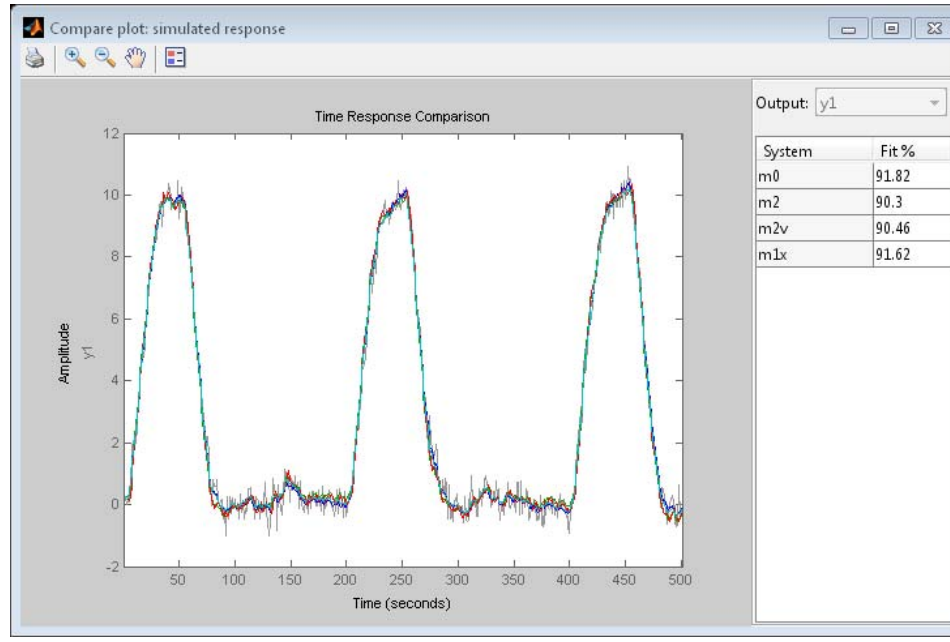
As observed, fixing `Tp1` to its known value dramatically improves the estimates of the remaining parameters in model `m2v`.

This also indicates that simple approximation should do well on the data:

```
m1x_init = idproc('P2D'); % simpler structure (no zero, no integrator)
m1x_init.Structure.Td.Maximum = 2;
m1x = procest(dat2v, m1x_init)
compare(dat2,m0,m2,m2v,m1x)
```

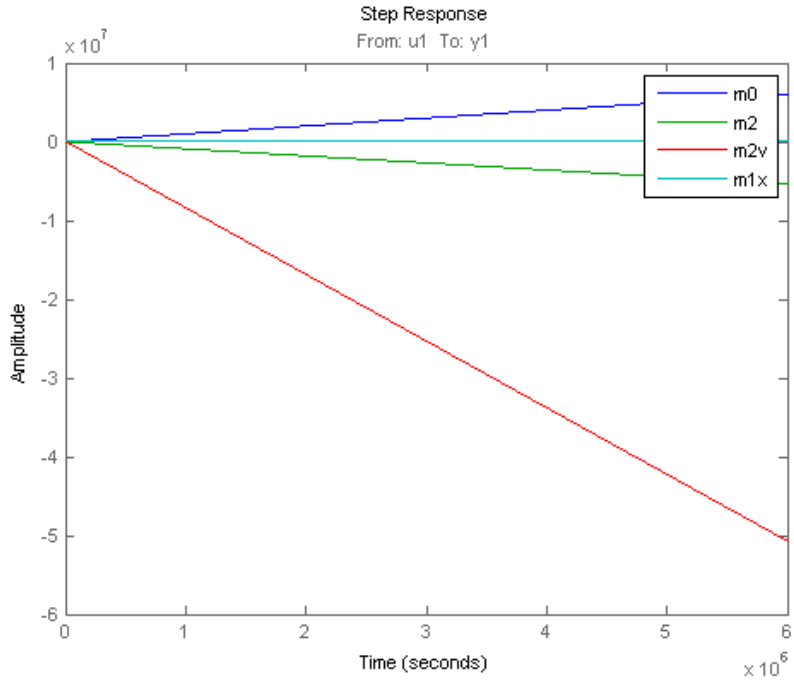
```
m1x =
Process model with transfer function:
                Kp
  G(s) = ----------------- * exp(-Td*s)
         (1+Tp1*s)(1+Tp2*s)

         Kp = 14815
        Tp1 = 15070
        Tp2 = 3.1505
         Td = 1.833

Parameterization:
    'P2D'
```

```
   Number of free coefficients: 4
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "dat2v".
Fit to estimation data: 91.23% (prediction focus)
FPE: 0.1209, MSE: 0.1193
```



Thus, the simpler model is able to estimate system output pretty well. However, `m1x` does not contain any integration, so the open loop long time range behavior will be quite different:

```
step(m0,m2,m2v,m1x)
legend('m0','m2','m2v','m1x')
bdclose('iddempr2')
```



Additional Information

For more information on identification of dynamic systems with System Identification Toolbox visit the System Identification Toolbox product information page.

Determining Model Order and Delay

Estimation requires you to specify the model order and delay. Many times, these values are not known. You can determine the model order and delay in one of the following ways:

- Guess their values by visually inspecting the data or based on the prior knowledge of the system.
- Estimate delay as a part of `idproc` or `idtf` model estimation. These models treat delay as an estimable parameter and you can determine their values by the estimation commands `procest` and `tfest`, respectively. However automatic estimation of delays can cause errors. Therefore, it is recommended that you analyze the data for delays in advance.
- To estimate delays, you can also use one of the following tools:
  - Estimate delay using `delayest`. The choice of the order of the underlying ARX model and the lower/upper bound on the value of the delay to be estimated influence the value returned by `delayest`.
  - Compute impulse response using `impulseest`. Plot the impulse response with a confidence interval of sufficient standard deviations (usually 3). The delay is indicated by the number of response samples that are inside the statistically zero region (marked by the confidence bound) before the response goes outside that region.
  - Select the model order in `n4sid` by specifying the model order as a vector.
  - Choose the model order of an ARX model using `arxstruc` or `ivstruc` and `selstruc`. These command select the number of poles, zeros and delay.

  See “Model Structure Selection: Determining Model Order and Input Delay” on page 3-196 for an example of using these tools.

Model Structure Selection: Determining Model Order and Input Delay

This example shows some methods for choosing and configuring the model structure. Estimation of a model using measurement data requires selection of a model structure (such as state-space or transfer function) and its order (e.g., number of poles and zeros) in advance. This choice is influenced by prior knowledge about the system being modeled, but can also be motivated by an analysis of data itself. This example describes some options for determining model orders and input delay.

Introduction

Choosing a model structure is usually the first step towards its estimation. There are various possibilities for structure - state-space, transfer functions and polynomial forms such as ARX, ARMAX, OE, BJ etc. If you do not have detailed prior knowledge of your system, such as its noise characteristics and indication of feedback, the choice of a reasonable structure may not be obvious. Also for a given choice of structure, the order of the model needs to be specified before the corresponding parameters are estimated. System Identification Toolbox™ offers some tools to assist in the task of model order selection.

The choice of a model order is also influenced by the amount of delay. A good idea of the input delay simplifies the task of figuring out the orders of other model coefficients. Discussed below are some options for input delay determination and model structure and order selection.

Choosing and Preparing Example Data for Analysis

This example uses the hair dryer data, also used by iddemo1 ("Estimating Simple Models from Real Laboratory Process Data"). The process consists of air being fanned through a tube. The air is heated at the inlet of the tube, and the input is the voltage applied to the heater. The output is the temperature at the outlet of the tube.

Let us begin by loading the measurement data and doing some basic preprocessing:

```
load dry2
```

Form a data set for estimation of the first half, and a reference set for validation purposes of the second half:

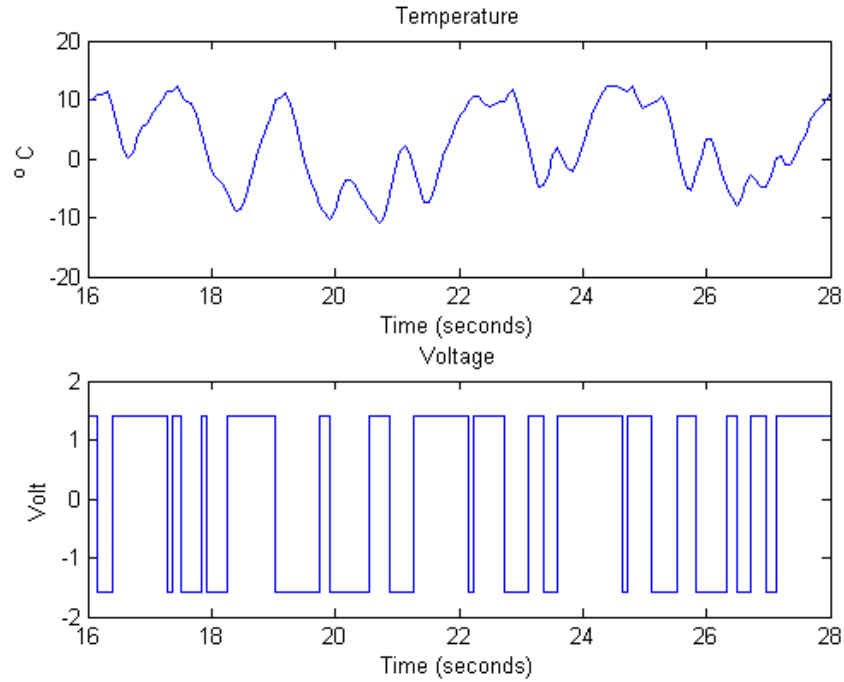
```
ze = dry2(1:500);
zr = dry2(501:1000);
```

Detrend each of the sets:

```
ze = detrend(ze);
zr = detrend(zr);
```

Let us look at a portion of the estimation data:

```
plot(ze(200:350))
```



Estimating Input Delay

There are various options available for determining the time delay from input to output. These are:

- Using the DELAYEST utility.
- Using a non-parametric estimate of the impulse response, using IMPULSEEST.
- Using the state-space model estimator N4SID with a number of different orders and finding the delay of the 'best' one.

Using delayest:

Let us discuss the above options in detail. Function delayest returns an estimate of the delay for a given choice of orders of numerator and denominator polynomials. This function evaluates an ARX structure:

```
y(t) + a1*y(t-1) + ...  + ana*y(t-na) = b1*u(t-nk) +
...+bnb*u(t-nb-nk+1)
```

with various delays and chooses the delay value that seems to return the best fit. In this process, chosen values of na and nb are used.

```
delay = delayest(ze) % na = nb = 2 is used, by default
```

```
delay =

     3
```

A value of 3 is returned by default. But this value may change a bit if the assumed orders of numerator and denominator polynomials (2 here) is changed. For example:

```
delay = delayest(ze,5,4)
```

```
delay =

     2
```

returns a value of 2. To gain insight into how `delayest` works, let us evaluate the loss function for various choices of delays explicitly. We select a second order model (`na=nb=2`), which is the default for `delayest`, and try out every time delay between 1 and 10. The loss function for the different models are computed using the validation data set:

```
V = arxstruc(ze,zr,struc(2,2,1:10));
```

We now select that delay that gives the best fit for the validation data:

```
[nn,Vm] = selstruc(V,0); % nn is given as [na nb nk]
```

The chosen structure was:

```
nn
```

```
nn =

     2     2     3
```

which show the best model has a delay of nn(3) = 3.

We can also check how the fit depends on the delay. This information is returned in the second output `Vm`. The logarithms of a quadratic loss function are given as the first row, while the indexes `na`, `nb` and `nk` are given as a column below the corresponding loss function.

```
Vm
```

```
Vm =

  Columns 1 through 7

   -0.1283   -1.3142   -1.8787   -0.2339    0.0084    0.0900    0.1957
    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000
    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000
```

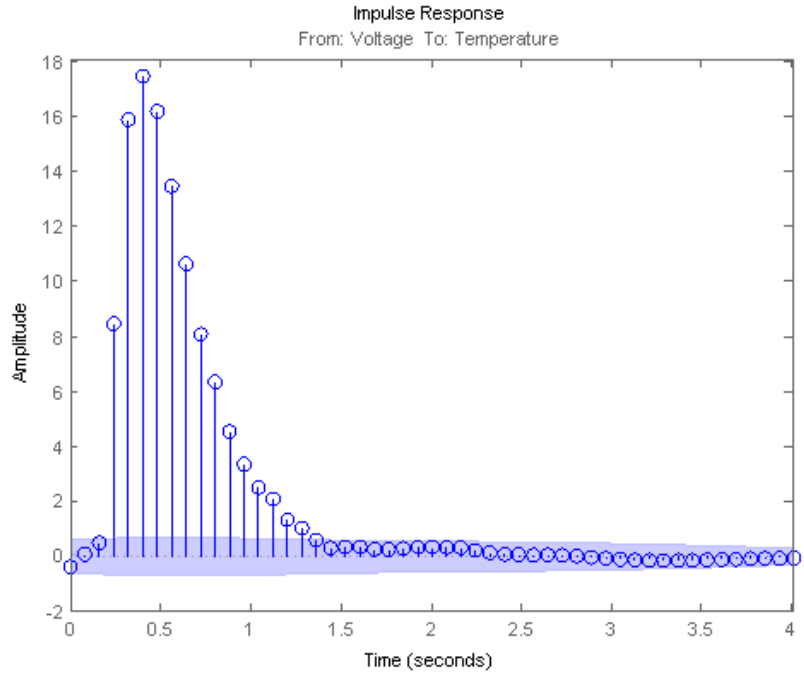
```
    1.0000    2.0000    3.0000    4.0000    5.0000    6.0000    7.0000

  Columns 8 through 10

    0.2082    0.1728    0.1631
    2.0000    2.0000    2.0000
    2.0000    2.0000    2.0000
    8.0000    9.0000   10.0000
```

The choice of 3 delays is thus rather clear, since the corresponding loss is minimum.

Using `impulse`

To gain a better insight into the dynamics, let us compute the impulse response of the system. We will use the function `impulseest` to compute a non-parametric impulse response model. We plot this response with a confidence interval represented by 3 standard deviations.

```
FIRModel = impulseest(ze);
clf
h = impulseplot(FIRModel);
showConfidence(h,3)
```

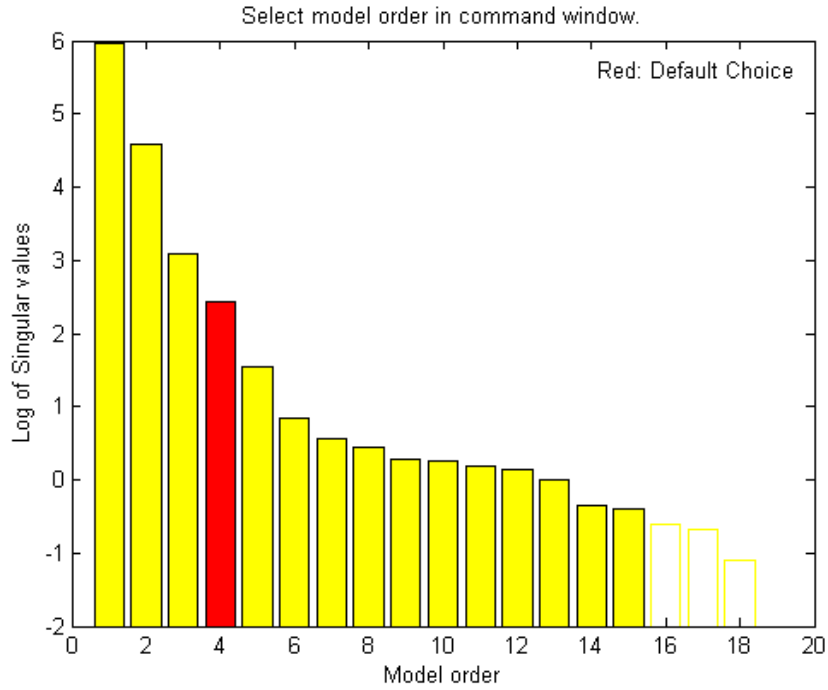


The filled light-blue region shows the confidence interval for the insignificant response in this estimation. There is a clear indication that the impulse response "takes off" (leaves the uncertainty region) after 3 samples. This points to a delay of three intervals.

Using `n4sid` based state-space evaluation

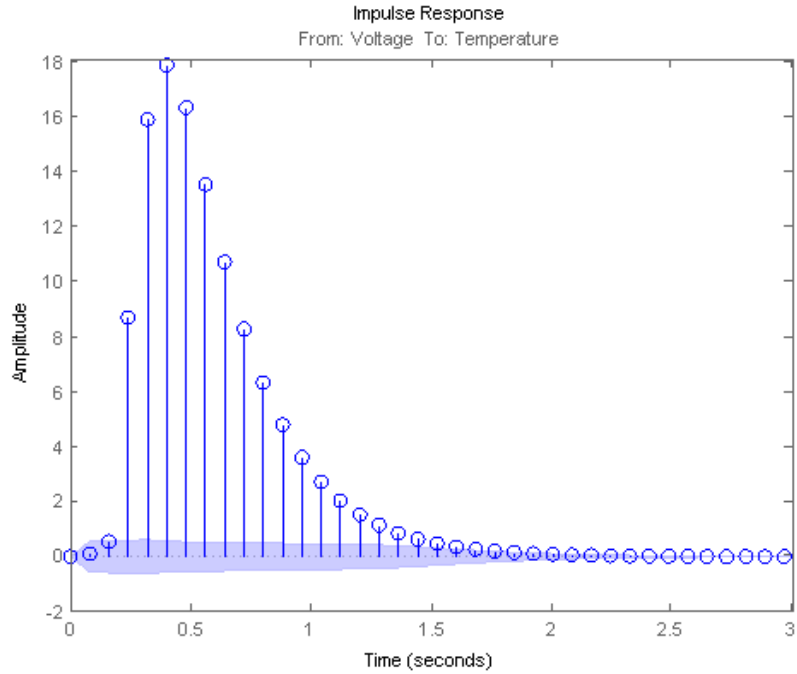
We may also estimate a family of parametric models to find the delay corresponding to the "best" model. In case of state-space models, a range of orders may be evaluated simultaneously and user gets prompted to choose the best order. Execute the following command to invoke `n4sid` in an interactive mode:

```
m = n4sid(ze,1:15); % All orders between 1 and 15.
```



The plot indicates an order of 4 as the best value. For this choice, let us compute the impulse response of the model `m`:

```
m = n4sid(ze,4);
showConfidence(impulseplot(m),3)
```



As with non-parametric impulse response, there is a clear indication that the delay from input to output is of three samples.

Choosing a Reasonable Model Structure

In lack of any prior knowledge, it is advisable to try out various available choices and use the one that seems to work the best. State-space models may be a good starting point since only the number of states needs to be specified in order to estimate a model. Also, a range of orders may be evaluated quickly, using `n4sid`, for determining the best order, as described in the next section. For polynomial models, a similar advantage is realized using the `arx` estimator. Output-error (OE) models may also be good choice for a starting polynomial model because of their simplicity.

Determining Model Order

Once you have decided upon a model structure to use, the next task is to determine the order(s). In general, the aim should be to not use a model order higher than necessary. This can be determined by analyzing the improvement in %fit as a function of model order. When doing this, it is advisable to use a separate, independent dataset for validation. Choosing an independent validation data set (`zr` in our example) would improve the detection of over-fitting.

In addition to a progressive analysis of multiple model orders, explicit determination of optimum orders can be performed for some model structures. Functions `arxstruc` and `selstruc` may be used for choosing the best order for ARX models. For our example, let us check the fit for all 100 combinations of up to 10 `b`-parameters and up to 10 `a`-parameters, all with a delay value of 3:

```
V = arxstruc(ze,zr,struc(1:10,1:10,3));
```

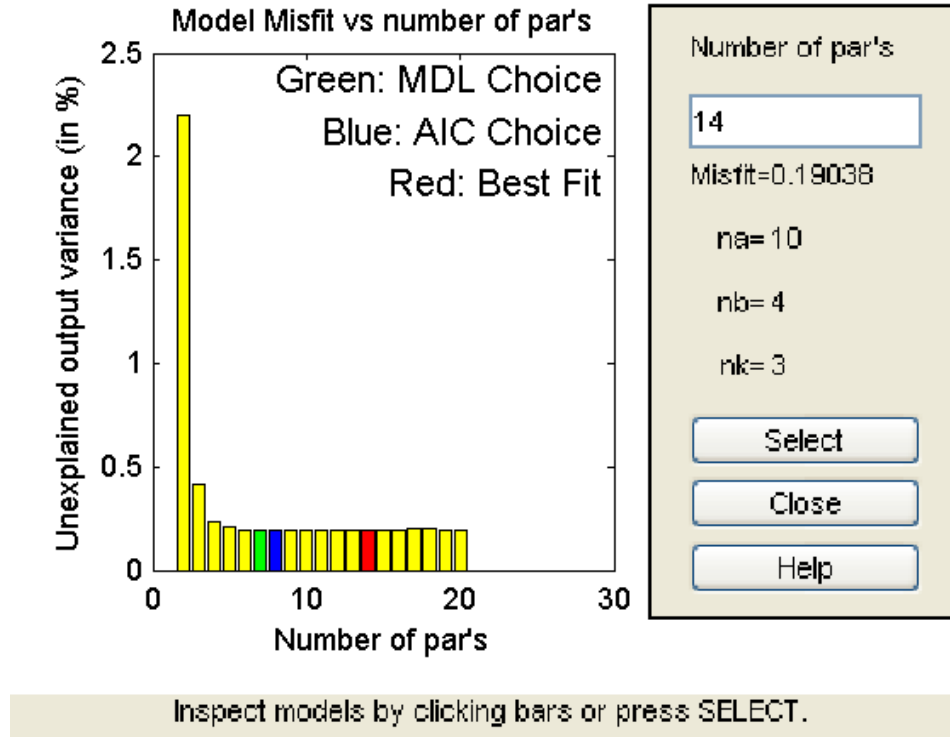
The best fit for the validation data set is obtained for:

```
nn = selstruc(V,0)
```

```
nn =

    10     4     3
```

Let us check how much the fit is improved for the higher order models. For this, we use the function `selstruc` with only one input. In this case, a plot showing the fit as a function of the number of parameters used is generated. The user is also prompted to enter the number of parameters. The routine then selects a structure with these many parameters that gives the best fit. Note that several different model structures use the same number of parameters. Execute the following command to choose a model order interactively:

```
nns = selstruc(V) %invoke selstruc in an interactive mode
```



The best fit is thus obtained for nn = [4 4 3], while we see that the improved fit compared to nn = [2 2 3] is rather marginal.

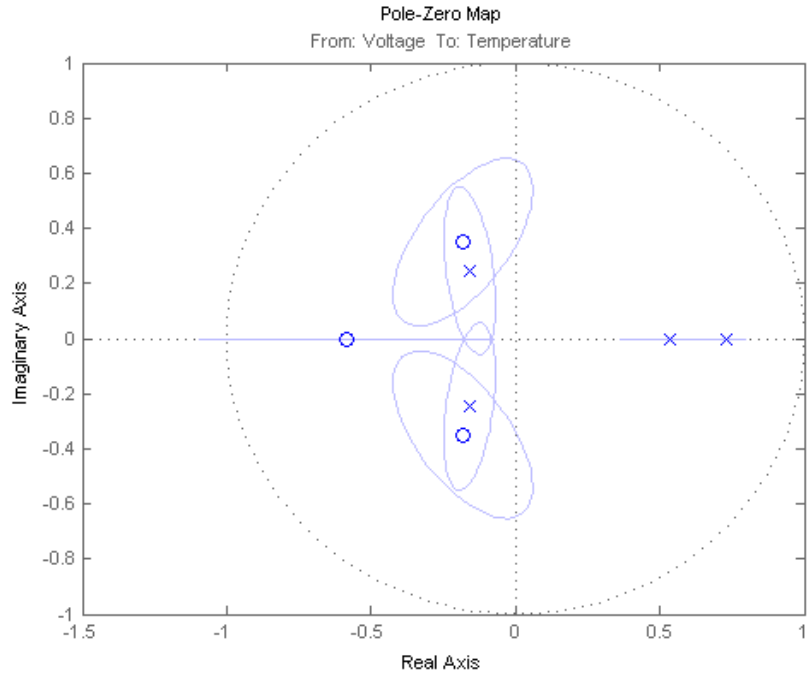
We may also approach this problem from the direction of reducing a higher order model. If the order is higher than necessary, then the extra parameters are basically used to "model" the measurement noise. These "extra" poles are estimated with a lower level of accuracy (large confidence interval). If their are cancelled by a zero located nearby, then it is an indication that this pole-zero pair may not be required to capture the essential dynamics of the system.

For our example, let us compute a 4th order model:

```
th4 = arx(ze,[4 4 3]);
```

Let us check the pole-zero configuration for this model. We can also include confidence regions for the poles and zeros corresponding to 3 standard deviations, in order to determine how accurately they are estimated and also how close the poles and zeros are to each other.

```
h = iopzplot(th4);
showConfidence(h,3)
```

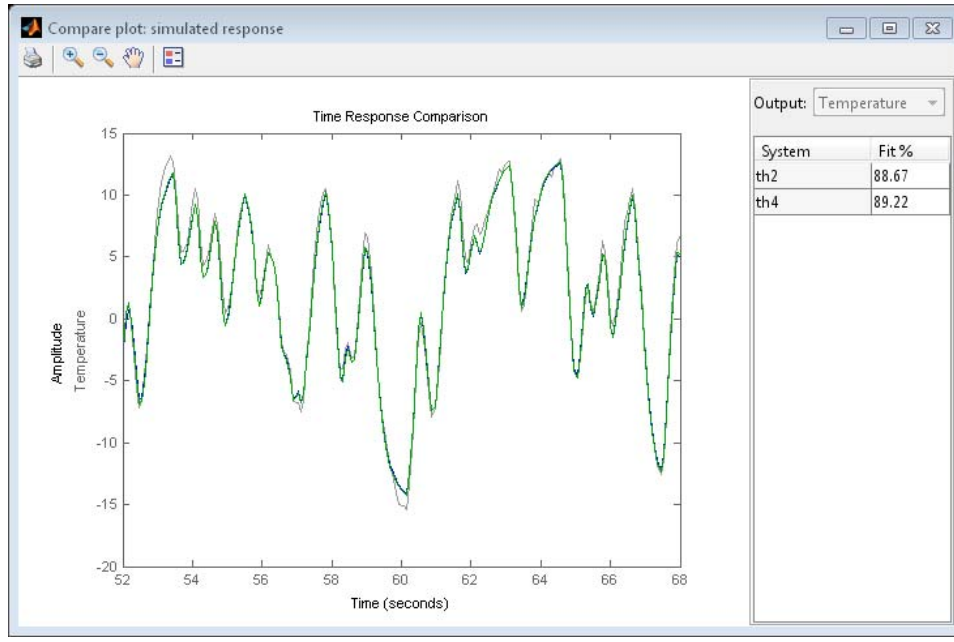


The confidence intervals for the two complex-conjugate poles and zeros overlap, indicating they are likely to cancel each other. Hence, a second order model might be adequate. Based on this evidence, let us compute a 2nd order ARX model:

```
th2 = arx(ze,[2 2 3]);
```

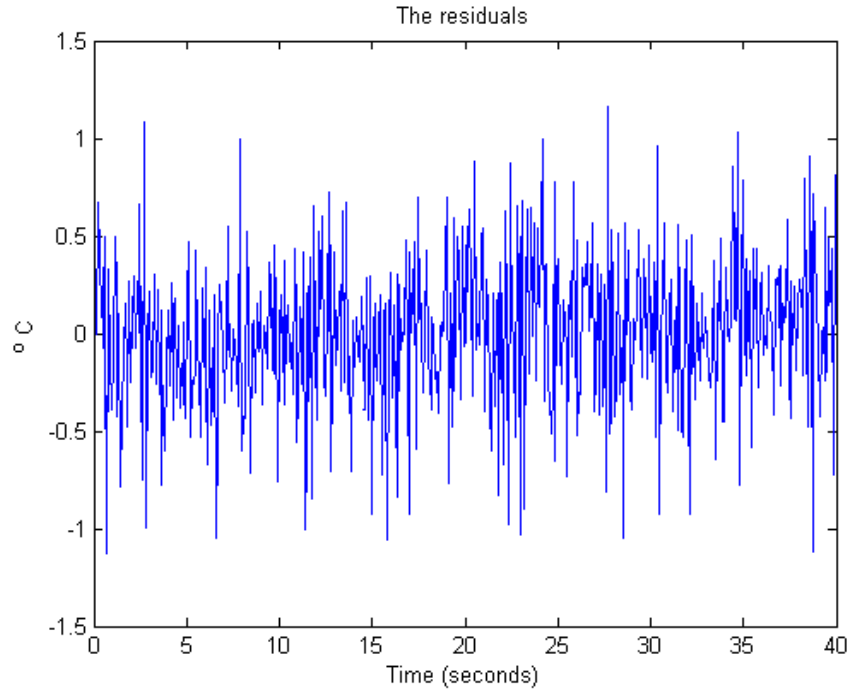
We can test how well this model (`th2`) is capable of reproducing the validation data set. To compare the simulated output from the two models with the actual output (plotting the mid 200 data points) we use the `compare` utility:

```
compare(zr(150:350),th2,th4)
```



The plot indicates that there was no significant loss of accuracy in reducing the order from 4 to 2. We can also check the residuals ("leftovers") of this model, i.e., what is left unexplained by the model.

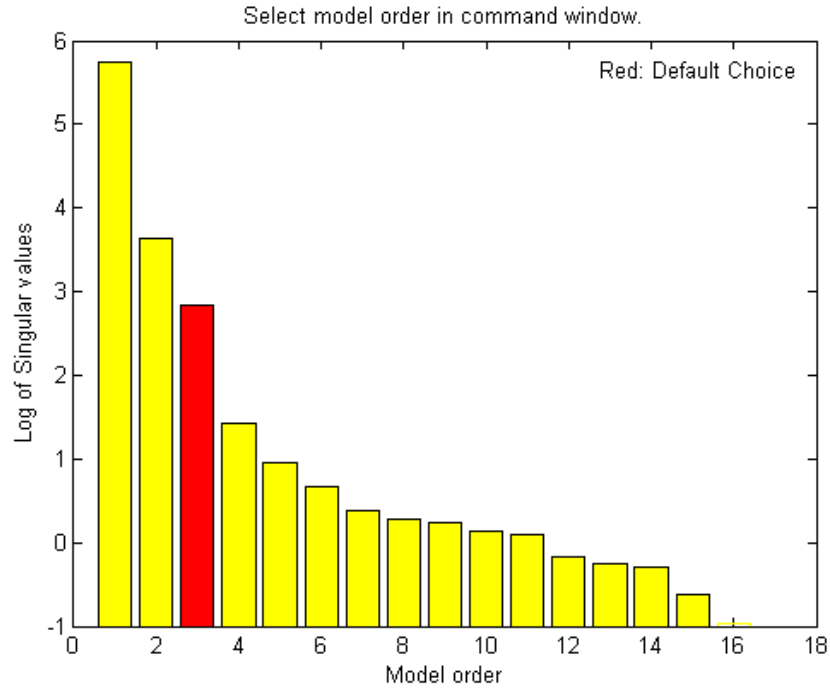
```
e = resid(ze,th2);
plot(e(:,1,[])), title('The residuals')
```



We see that the residuals are quite small compared to the signal level of the output, that they are reasonably well (although not perfectly) uncorrelated with the input and among themselves. We can thus be (provisionally) satisfied with the model `th2`.

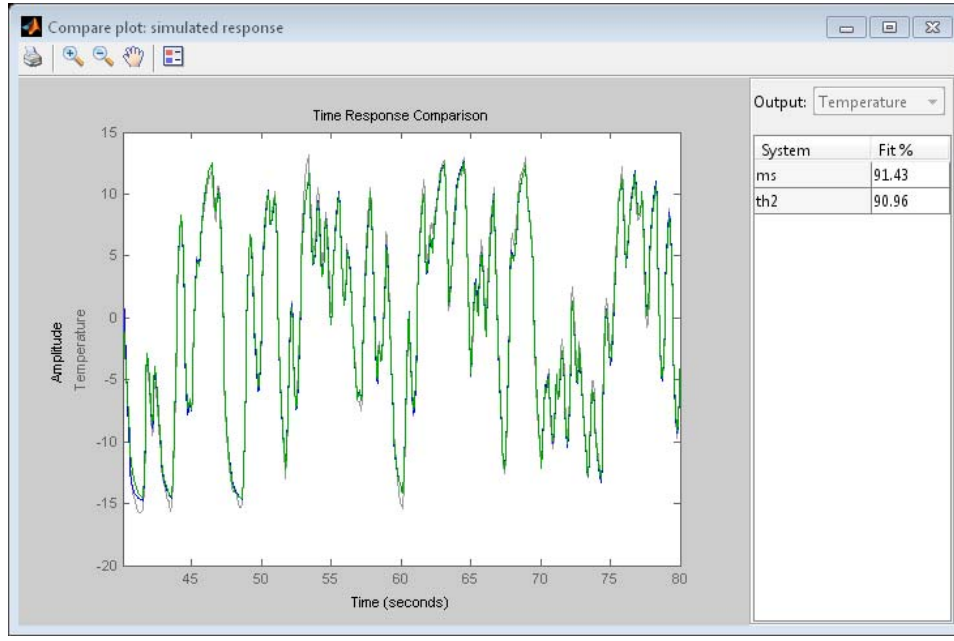
Let us now check if we can determine the model order for a state-space structure. As before, we know the delay is 3 samples. We can try all orders from 1 to 15 with a fixed delay of 3 in `n4sid`. Execute the following command to try various orders and choose one interactively.

```
ms = n4sid(ze,[1:15],'InputDelay',2); %n4sid estimation with variable orders
```



The default order, indicated in the figure above, is 3, that is in good agreement with our earlier findings. Finally, we compare how the state-space model `ms` and the ARX model `th2` compare in reproducing the measured output of the validation data:

```
ms = n4sid(ze,3,'InputDelay',2);
compare(zr,ms,th2)
```



The comparison plot indicates that the two models are practically identical.

Conclusions

This example described some options for choosing a reasonable model order. Determining delay in advance can simplify the task of choosing orders. With ARX and state-space structures, we have some special tools (`arx` and `n4sid` estimators) for automatically evaluating a whole set of model orders, and choosing the best one among them. The information revealed by this exercise (using utilities such as `arxstruc`, `selstruc`, `n4sid` and `delayest`) could be used as a starting point when estimating models of other structures, such as BJ and ARMAX.

Additional Information

For more information on identification of dynamic systems with System Identification Toolbox visit the System Identification Toolbox product information page.

Frequency Domain Identification: Estimating Models Using Frequency Domain Data

This example shows how to estimate models using frequency domain data. The estimation and validation of models using frequency domain data work the same way as they do with time domain data. This provides a great amount of flexibility in estimation and analysis of models using time and frequency domain as well as spectral (FRF) data. You may simultaneously estimate models using data in both domains, compare and combine these models. A model estimated using time domain data may be validated using spectral data or vice-versa.

Frequency domain data can not be used for estimation or validation of nonlinear models.

Introduction

Frequency domain experimental data are common in many applications. It could be that the data was collected as frequency response data (frequency functions: FRF) from the process using a frequency analyzer. It could also be that it is more practical to work with the input's and output's Fourier transforms (FFT of time-domain data), for example to handle periodic or band-limited data. (A band-limited continuous time signal has no frequency components above the Nyquist frequency). In System Identification Toolbox, frequency domain I/O data are represented the same way as time-domain data, i.e., using `iddata` objects. The 'Domain' property of the object must be set to 'Frequency'. Frequency response data are represented as complex vectors or as magnitude/phase vectors as a function of frequency. IDFRD objects in the toolbox are used to encapsulate FRFs, where a user specifies the complex response data and a frequency vector. Such IDDATA or IDFRD objects (and also FRD objects of Control System Toolbox) may be used seamlessly with any estimation routine (such as `procest`, `tfest` etc).

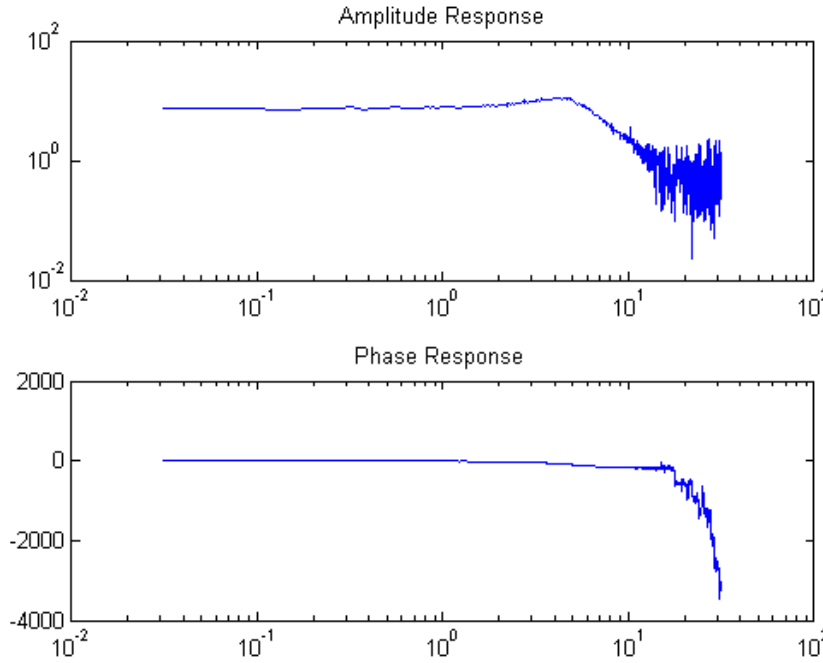
Inspecting Frequency Domain Data

Let us begin by loading some frequency domain data:

```
load demofr
```

This MAT-file contains frequency response data at frequencies `W`, with the amplitude response `AMP` and the phase response `PHA`. Let us first have a look at the data:

```
subplot(211), loglog(W,AMP),title('Amplitude Response')
subplot(212), semilogx(W,PHA),title('Phase Response')
```



This experimental data will now be stored as an IDFRD object. First transform amplitude and phase to a complex valued response:

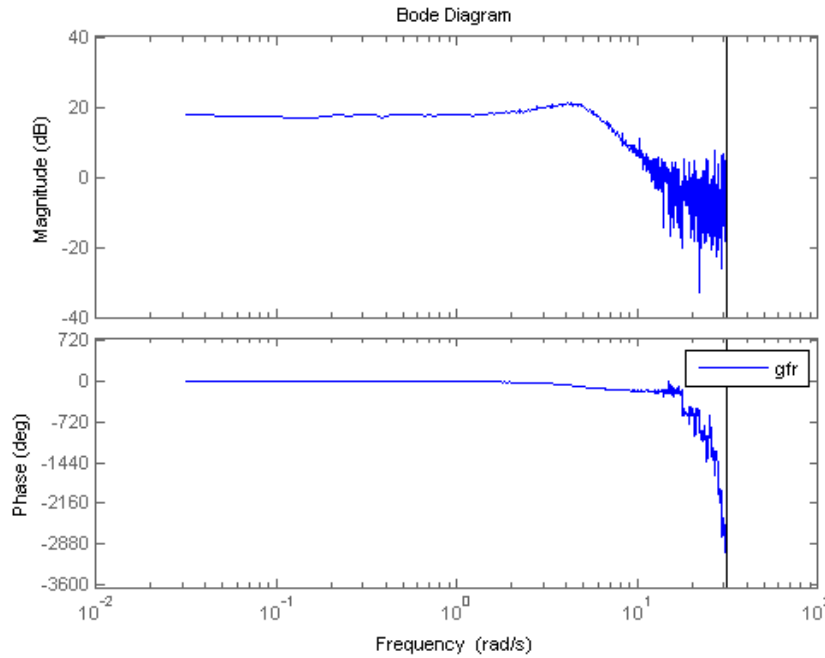
```
zfr = AMP.*exp(1i*PHA*pi/180);
Ts = 0.1;
gfr = idfrd(zfr,W,Ts);
```

`Ts` is the sampling interval of the underlying data. If the data corresponds to continuous time, for example since the input has been band-limited, use Ts = 0.

Note: If you have the Control System Toolbox™, you could use an FRD object instead of the IDFRD object. IDFRD has options for more information, like disturbance spectra and uncertainty measures which are not available in FRD objects.

The IDFRD object `gfr` now contains the data, and it can be plotted and analyzed in different ways. To view the data, we may use `plot` or `bode`:

```
clf
bode(gfr), legend('gfr')
```



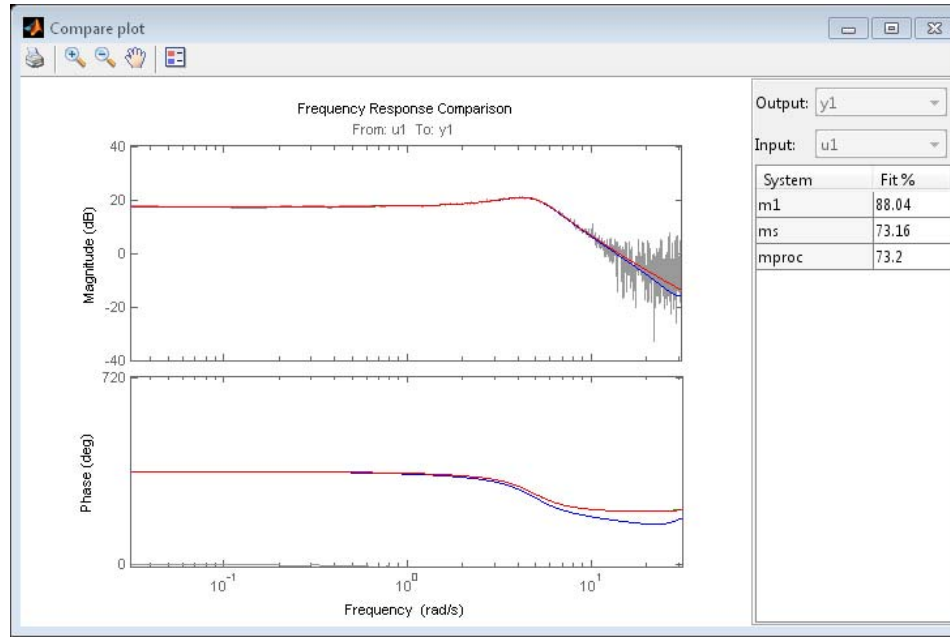
Estimating Models Using Frequency Response (FRF) Data

To estimate models, you can now use `gfr` as a data set with all the commands of the toolbox in a transparent fashion. The only restriction is that noise models cannot be built. This means that for polynomial models only OE (output-error models) apply, and for state-space models, you have to fix `K = 0`.

```
m1 = oe(gfr,[2 2 1]) % Discrete-time Output error (transfer function) model
ms = ssest(gfr) % Continuous-time state-space model with default choice of
mproc = process(gfr,'P2UDZ') % 2nd-order, continuous-time model with underd
compare(gfr,m1,ms,mproc)


m1 =
Discrete-time OE model:  y(t) = [B(z)/F(z)]u(t) + e(t)
  B(z) = 0.9982 z^-1 + 0.4974 z^-2

  F(z) = 1 - 1.499 z^-1 + 0.6998 z^-2

Sample time: 0.1 seconds

Parameterization:
   Polynomial orders:   nb=2   nf=2   nk=1
   Number of free coefficients: 4
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainti

Status:
Estimated using OE on frequency response data "gfr".
Fit to estimation data: 88.04% (prediction focus)
FPE: 0.2501, MSE: 0.2494

ms =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
           x1       x2
   x1  -0.9005    6.636
   x2   -3.308   -2.669

  B =
         u1
   x1   -32.9
   x2  -28.33

  C =
```

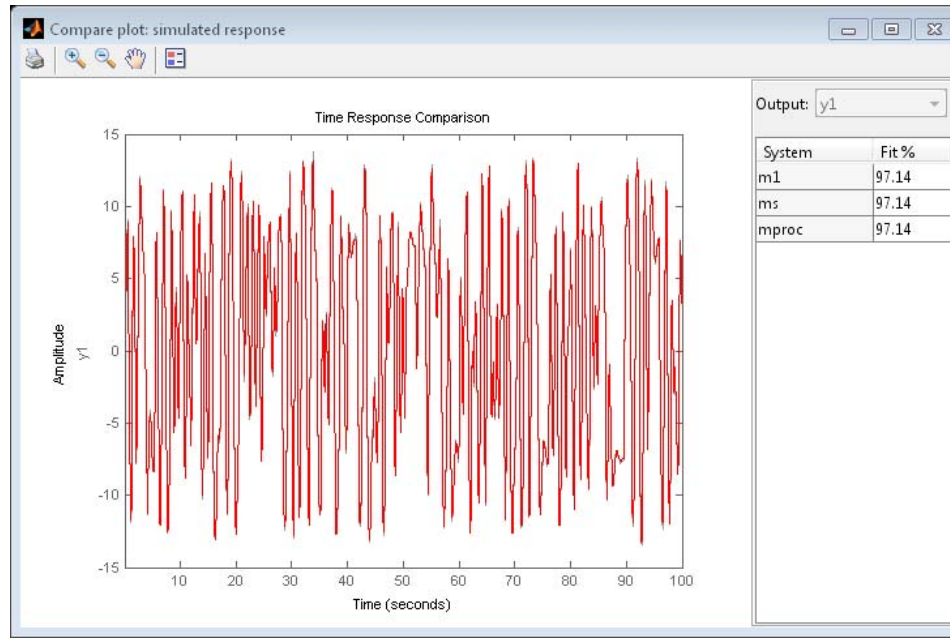
```
           x1       x2
   y1  -0.5073    0.499

  D =
       u1
   y1   0

  K =
       y1
   x1   0
   x2   0

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 8
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainti

Status:
Estimated using SSEST on frequency response data "gfr".
Fit to estimation data: 73.16% (prediction focus)
FPE: 0.2511, MSE: 1.255

mproc =
Process model with transfer function:
                     1+Tz*s
  G(s) = Kp * ---------------------- * exp(-Td*s)
              1+2*Zeta*Tw*s+(Tw*s)^2

         Kp = 7.4612
         Tw = 0.20246
       Zeta = 0.36238
         Td = 0
         Tz = 0.013617

Parameterization:
    'P2DUZ'
   Number of free coefficients: 5
   Use "getpvec", "getcov" for parameters and their uncertainties.
```

```
Status:
Estimated using PROCEST on frequency response data "gfr".
Fit to estimation data: 73.2% (prediction focus)
FPE: 0.2504, MSE: 1.251
```



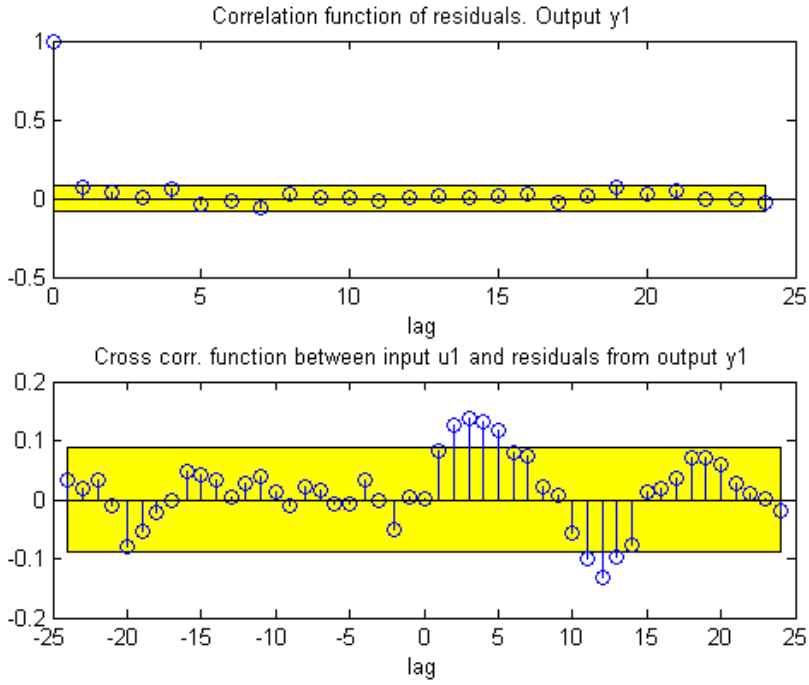
As shown above a variety of linear model types may be estimated in both continuous and discrete time domains, using spectral data. These models may be validated using, time-domain data. The time-domain I/O data set `ztime`, for example, is collected from the same system, and can be used for validation of `m1`, `ms` and `mproc`:

```
compare(ztime,m1,ms,mproc) %validation in a different domain
```



We may also look at the residuals to affirm the quality of the model using the validation data `ztime`. As observed, the residuals are almost white:

```
resid(mproc,ztime) % Residuals plot
```

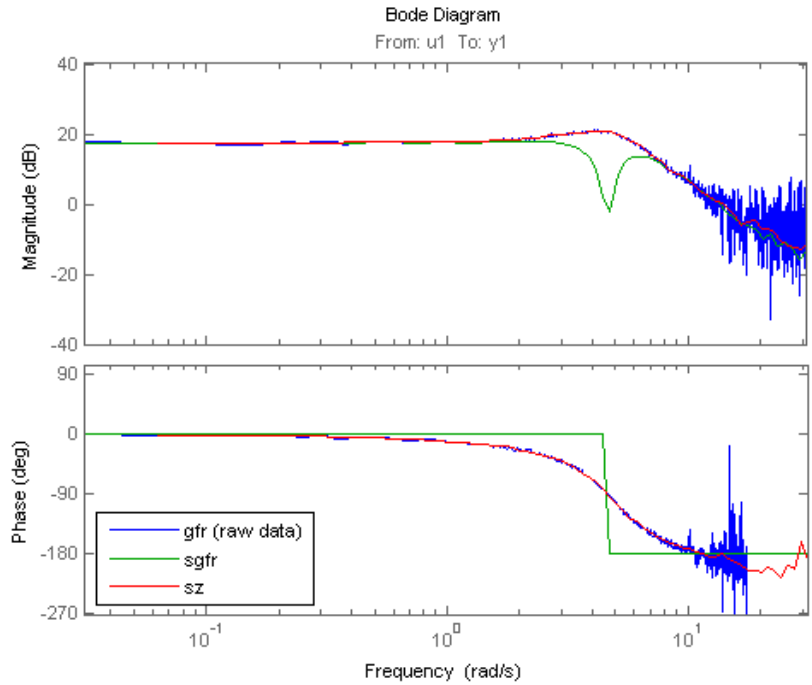


Condensing Data Using SPAFDR

An important reason to work with frequency response data is that it is easy to condense the information with little loss. The command SPAFDR allows you to compute smoothed response data over limited frequencies, for example with logarithmic spacing. Here is an example where the `gfr` data is condensed to 100 logarithmically spaced frequency values. With a similar technique, also the original time domain data can be condensed:

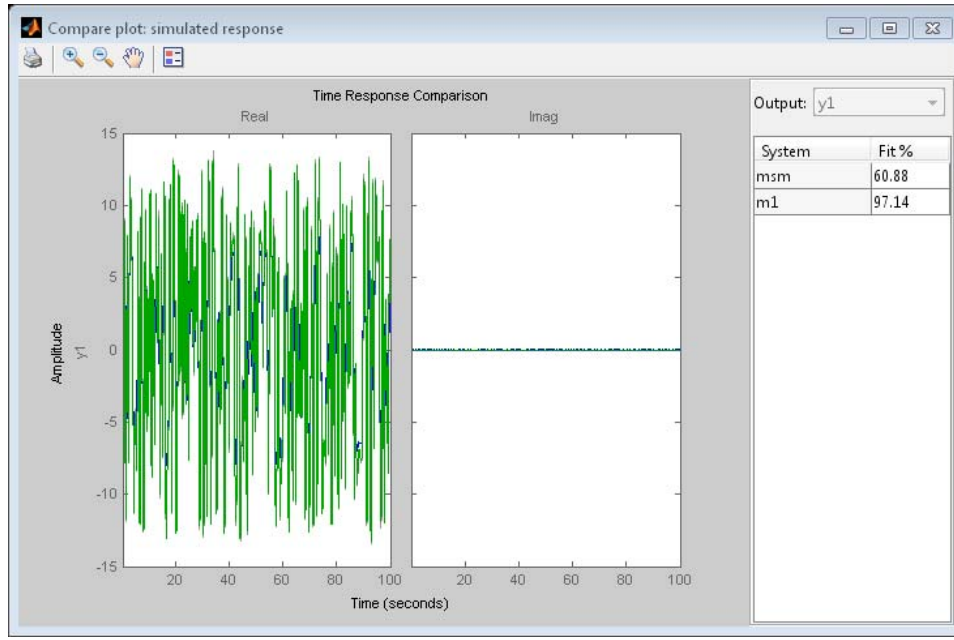
```
sgfr = spafdr(gfr) % spectral estimation with frequency-dependent resolutio
sz = spafdr(ztime); % spectral estimation using time-domain data
clf
bode(gfr,sgfr,sz)
axis([pi/100 10*pi, -272 105])
legend('gfr (raw data)','sgfr','sz','location','southwest')
```

```
sgfr =
IDFRD model.
Contains Frequency Response Data for 1 output(s) and 1 input(s), and the sp
Response data and disturbance spectra are available at 200 frequency points

Sample time: 0.1 seconds
Output channels: 'y1'
Input channels: 'u1'
Status:
Estimated using SPAFDR on frequency response data "gfr".
```



The Bode plots show that the information in the smoothed data has been taken well care of. Now, these data records with 100 points can very well be used for model estimation. For example:

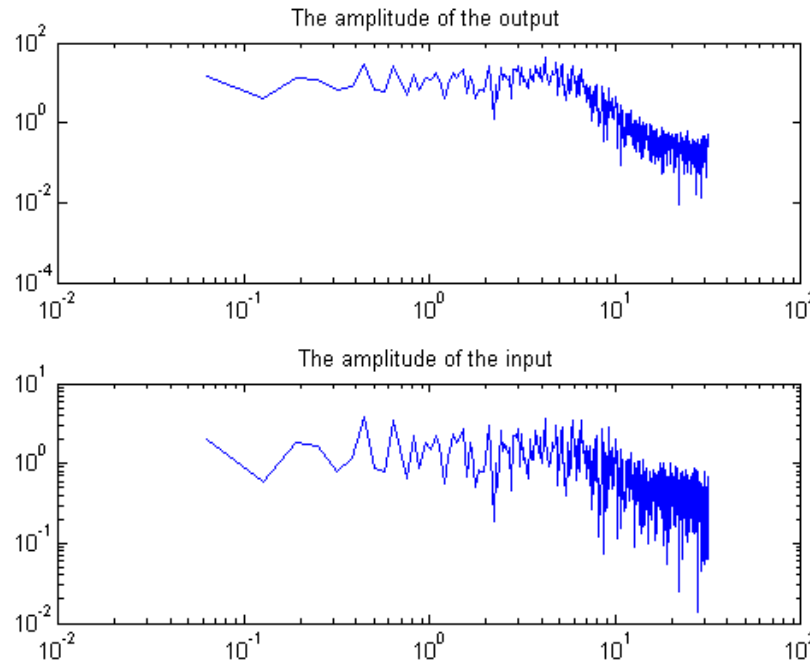
```
msm = oe(sgfr,[2 2 1]);
compare(ztime,msm,m1) % msm has the same accuracy as M1 (based on 1000 poin
```



Estimation Using Frequency-Domain I/O Data

It may be that the measurements are available as Fourier transforms of inputs and output. Such frequency domain data from the system are given as the signals Y and U. In loglog plots they look like

```
Wfd = (0:500)'*10*pi/500;
subplot(211),loglog(Wfd,abs(Y)),title('The amplitude of the output')
subplot(212),loglog(Wfd,abs(U)),title('The amplitude of the input')
```



The frequency response data is essentially the ratio between `Y` and `U`. To collect the frequency domain data as an IDDATA object, do as follows:

```
ZFD = iddata(Y,U,'ts',0.1,'Domain','Frequency','Freq',Wfd)
```

```
ZFD =

Frequency domain data set with responses at 501 frequencies,
ranging from 0 to 31.416 rad/seconds
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
```

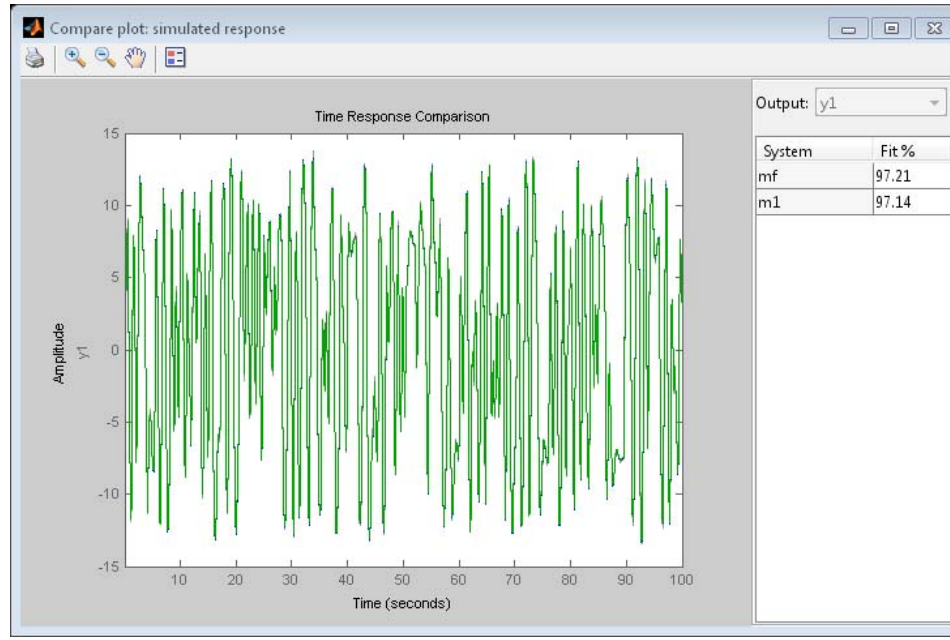
```
   u1
```

Now, again the frequency domain data set `ZFD` can be used as data in all estimation routines, just as time domain data and frequency response data:

```
mf = ssest(ZFD)
compare(ztime,mf,m1)
```

```
mf =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
          x1      x2
   x1  -1.501   6.791
   x2  -3.115  -2.059

  B =
          u1
   x1  -28.11
   x2  -33.39

  C =
           x1       x2
   y1  -0.5844   0.4129

  D =
       u1
   y1   0

  K =
       y1
   x1   0
   x2   0

Parameterization:
   FREE form (all coefficients in A, B, C free).
```

```
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 8
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainti

Status:
Estimated using SSEST on frequency domain data "ZFD".
Fit to estimation data: 97.21% (prediction focus)
FPE: 0.04263, MSE: 0.04184
```



Transformations Between Data Representations (Time - Frequency)

Time and frequency domain input-output data sets can be transformed to either domain by using FFT and IFFT. These commands are adapted to IDDATA objects:

```
dataf = fft(ztime)
datat = ifft(dataf)
```

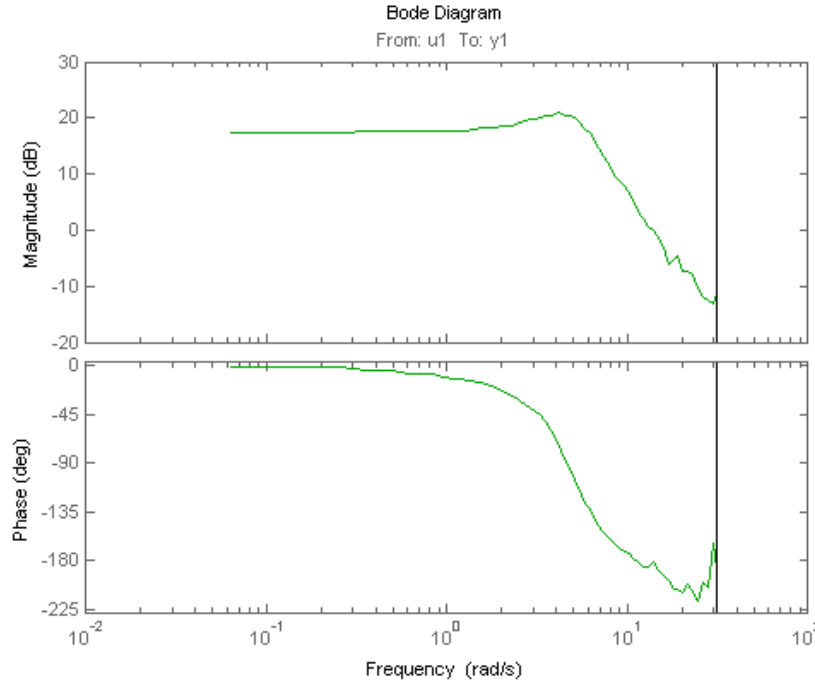
```
dataf =

Frequency domain data set with responses at 501 frequencies,
ranging from 0 to 31.416 rad/seconds
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1


datat =

Time domain data set with 1000 samples.
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

Time and frequency domain input-output data can be transformed to frequency response data by SPAFDR, SPA and ETFE:

```
g1 = spafdr(ztime)
g2 = spafdr(ZFD);
bode(g1,g2)
```

```
g1 =
IDFRD model.
Contains Frequency Response Data for 1 output(s) and 1 input(s), and the sp
Response data and disturbance spectra are available at 100 frequency points

Sample time: 0.1 seconds
```

```
Output channels: 'y1'
Input channels: 'u1'
Status:
Estimated using SPAFDR on time domain data "ztime".
```



Frequency response data can also be transformed to more smoothed data (less resolution and less data) by SPAFDR and SPA;

```
g3 = spafdr(gfr);
```

Frequency response data can be transformed to frequency domain input-output signals by the command IDDATA:

```
gfd = iddata(g3)
```

```
gfd =
```

```
Frequency domain data set with responses at 200 frequencies,
ranging from -31.416 to 31.416 rad/seconds
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

Using Continuous-time Frequency-domain Data to Estimate Continuous-time Models

Time domain data can naturally only be stored and dealt with as discrete-time, sampled data. Frequency domain data have the advantage that continuous time data can be represented correctly. Suppose that the underlying continuous time signals have no frequency information above the Nyquist frequency, e.g. because they are sampled fast, or the input has no frequency component above the Nyquist frequency. Then the Discrete Fourier transforms (DFT) of the data also are the Fourier transforms of the continuous time signals, at the chosen frequencies. They can therefore be used to directly fit continuous time models. In fact, this is the correct way of using band-limited data for model fit.

This will be illustrated by the following example.

Consider the continuous time system:

$$G(s) = \frac{1}{s^2 + s + 1}$$

```
m0 = idpoly(1,1,1,1,[1 1 1],'ts',0)
```

```
m0 =
Continuous-time OE model:  y(t) = [B(s)/F(s)]u(t) + e(t)
  B(s) = 1
```

```
  F(s) = s^2 + s + 1

Parameterization:
   Polynomial orders:   nb=1   nf=2   nk=0
   Number of free coefficients: 3
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainti

Status:
Created by direct construction or transformation. Not estimated.
```

Choose an input with low frequency contents that is fast sampled:

```
rng(235,'twister');
u = idinput(500,'sine',[0 0.2]);
u = iddata([],u,0.1,'intersamp','bl');
```

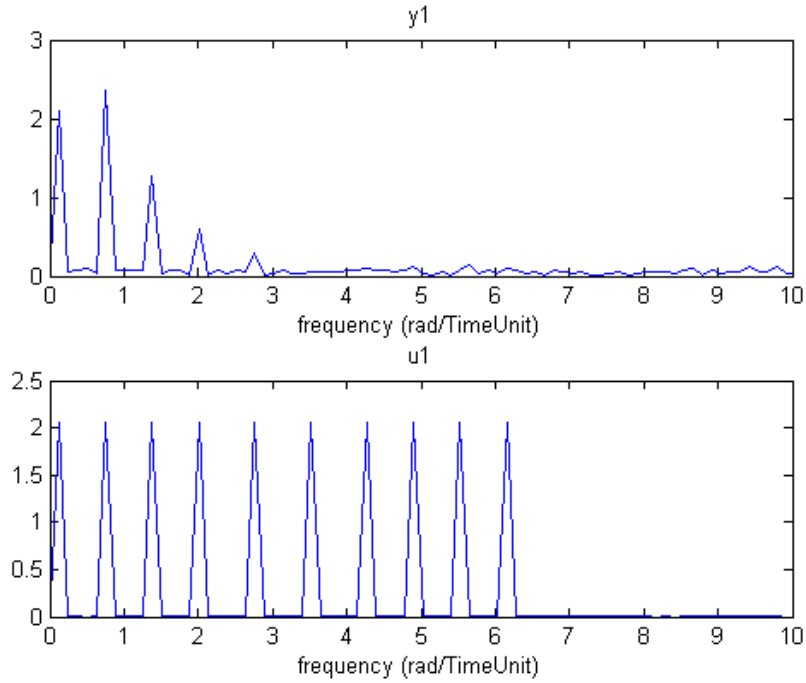
0.1 is the sampling interval, and `'bl'` indicates that the input is band-limited, i.e. in continuous time it consists of sinusoids with frequencies below half the sampling frequency. Correct simulation of such a system should be done in the frequency domain:

```
uf = fft(u);
uf.ts = 0; % Denoting that the data is continuous time
yf = sim(m0,uf);
%
% Add some noise to the data:
yf.y = yf.y + 0.05*(randn(size(yf.y))+1i*randn(size(yf.y)));
dataf = [yf uf] % This is now a continuous time frequency domain data set.


dataf =

Frequency domain data set with responses at 251 frequencies,
ranging from 0 to 31.416 rad/seconds
Sample time: 0 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
```

u1

Look at the data:

```
plot(dataf)
axis([0 10 0 2.5])
```



Using `dataf` for estimation will by default give continuous time models: State-space:

```
m4 = ssest(dataf,2); %Second order continuous-time model
```

For a polynomial model with `nb = 2` numerator coefficient and `nf = 2` estimated denominator coefficients use:

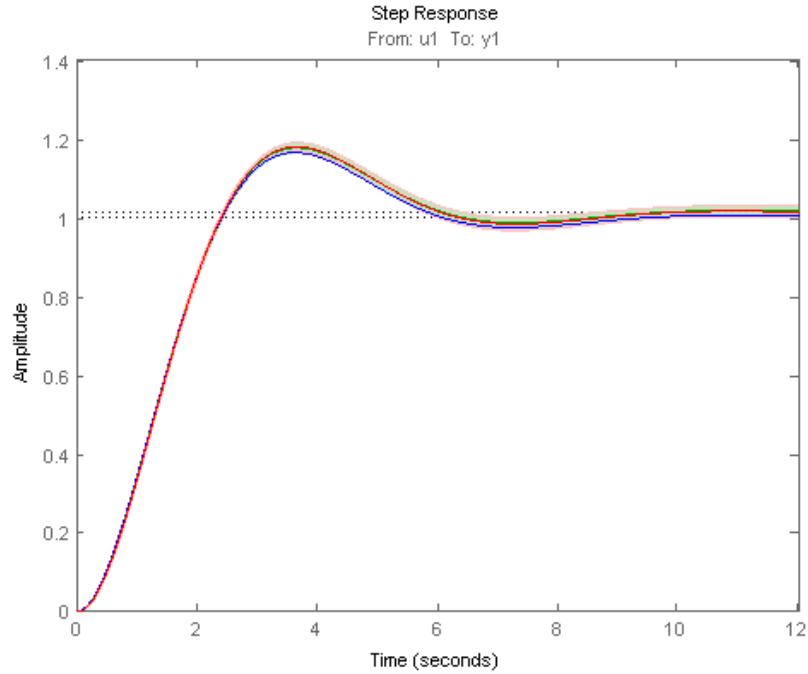
```
nb = 2;
```

```
nf = 2;
m5 = oe(dataf,[nb nf])
```

```
m5 =
Continuous-time OE model:  y(t) = [B(s)/F(s)]u(t) + e(t)
  B(s) = -0.01761 s + 1

  F(s) = s^2 + 0.9873 s + 0.9902

Parameterization:
   Polynomial orders:   nb=2   nf=2   nk=0
   Number of free coefficients: 4
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainti

Status:
Estimated using OE on frequency domain data "dataf".
Fit to estimation data: 70.15% (prediction focus)
FPE: 0.00482, MSE: 0.004723
```

Compare step responses with uncertainty of the true system `m0` and the models `m4` and `m5`:

```
clf
h = stepplot(m0,m4,m5);
showConfidence(h,1)
% *Blue: m0, Green: m4, Red: m5.*
% The confidence intervals are shown with patches.
```



Although it was not necessary in this case, it is generally advised to focus the fit to a limited frequency band (low pass filter the data) when estimating using continuous time data. The system has a bandwidth of about 3 rad/s, and was excited by sinusoids up to 6.2 rad/s. A reasonable frequency range to focus the fit to is then [0 7] rad/s:

```
m6 = ssest(dataf,2,ssestOptions('Focus',[0 7])) % state space model
```

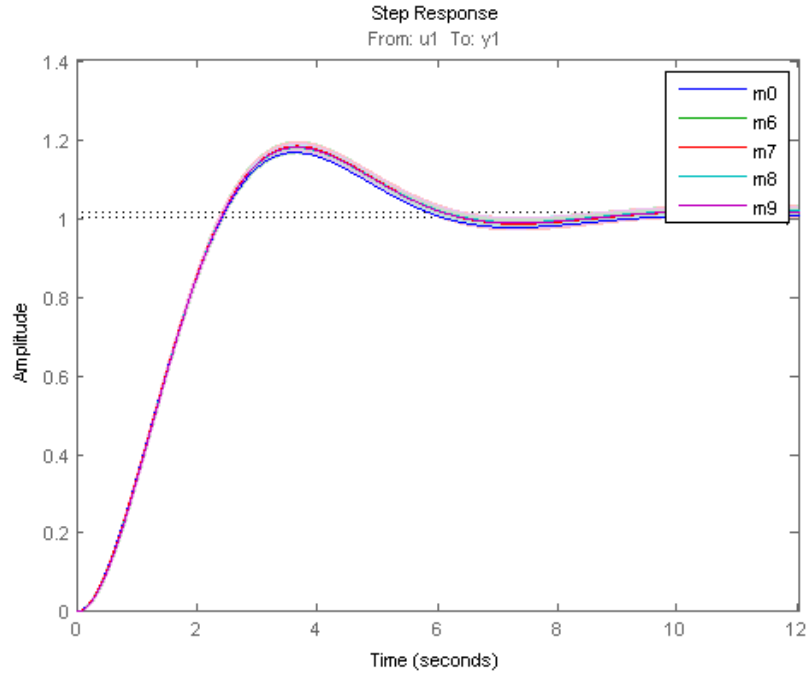
```
m6 =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
             x1        x2
```

```
   x1  -0.2714   -1.809
   x2   0.4407  -0.7282

  B =
            u1
   x1   0.5873
   x2  -0.3025

  C =
          x1       x2
   y1  0.7565   1.526

  D =
       u1
   y1   0

  K =
       y1
   x1   0
   x2   0

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 8
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainti

Status:
Estimated using SSEST on frequency domain data "dataf".
Fit to estimation data: 87.08% (filter focus)
FPE: 0.004213, MSE: 0.003716

m7 = oe(dataf,[1 2],oeOptions('Focus',[0 7])) % polynomial model of Output


m7 =
Continuous-time OE model:  y(t) = [B(s)/F(s)]u(t) + e(t)
  B(s) = 0.9866
```

```
  F(s) = s^2 + 0.9791 s + 0.9761

Parameterization:
   Polynomial orders:   nb=1   nf=2   nk=0
   Number of free coefficients: 3
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainti

Status:
Estimated using OE on frequency domain data "dataf".
Fit to estimation data: 87.04% (filter focus)
FPE: 0.004008, MSE: 0.003742

opt = procestOptions('SearchMethod','lsqnonlin','Focus',[0 7]); % Requires
m8 = procest(dataf,'P2UZ',opt)  % process model with underdamped poles


m8 =
Process model with transfer function:
                     1+Tz*s
  G(s) = Kp * ----------------------
              1+2*Zeta*Tw*s+(Tw*s)^2

         Kp = 1.0124
         Tw = 1.0019
       Zeta = 0.5021
         Tz = -0.017474

Parameterization:
    'P2UZ'
   Number of free coefficients: 4
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on frequency domain data "dataf".
Fit to estimation data: 87.08% (filter focus)
FPE: 0.003947, MSE: 0.003716

opt = tfestOptions('SearchMethod','lsqnonlin','Focus',[0 7]); % Requires Op
m9 = tfest(dataf,2,opt) % transfer function with 2 poles
```

```
m9 =

  From input "u1" to output "y1":
    -0.01647 s + 1.003
  ----------------------
  s^2 + 0.9948 s + 0.9922

Continuous-time identified transfer function.

Parameterization:
   Number of poles: 2   Number of zeros: 1
   Number of free coefficients: 4
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties

Status:
Estimated using TFEST on frequency domain data "dataf".
Fit to estimation data: 87.08% (filter focus)
FPE: 0.004067, MSE: 0.003717
```

```
h = stepplot(m0,m6,m7,m8,m9);
showConfidence(h,1)
legend('show')
```



Conclusions

We saw how time, frequency and spectral data can seamlessly be used to estimate a variety of linear models in both continuous and discrete time domains. The models may be validated and compared in domains different from the ones they were estimated in. The data formats (time, frequency and spectrum) are interconvertible, using methods such as `fft`, `ifft`, `spafdr` and `spa`. Furthermore, direct, continuous-time estimation is achievable by using `tfest`, `ssest` and `procest` estimation routines. The seamless use of data in any domain for estimation and analysis is an important feature of System Identification Toolbox.

Additional Information

For more information on identification of dynamic systems with System Identification Toolbox visit the System Identification Toolbox product information page.

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

This example shows how to estimate parameters in user-defined model structures. Such structures are specified by IDGREY (linear state-space) or IDNLGREY (nonlinear state-space) models. We shall investigate how to assign structure, fix parameters and create dependencies among them.

Experiment Data

We shall investigate data produced by a (simulated) dc-motor. We first load the data:

```
load dcmdata
who
```

```
Your variables are:

text  u     y
```

The matrix `y` contains the two outputs: `y1` is the angular position of the motor shaft and `y2` is the angular velocity. There are 400 data samples and the sampling interval is 0.1 seconds. The input is contained in the vector `u`. It is the input voltage to the motor.

```
z = iddata(y,u,0.1); % The IDDATA object
z.InputName = 'Voltage';
z.OutputName = {'Angle';'AngVel'};
plot(z(:,1,:))
```

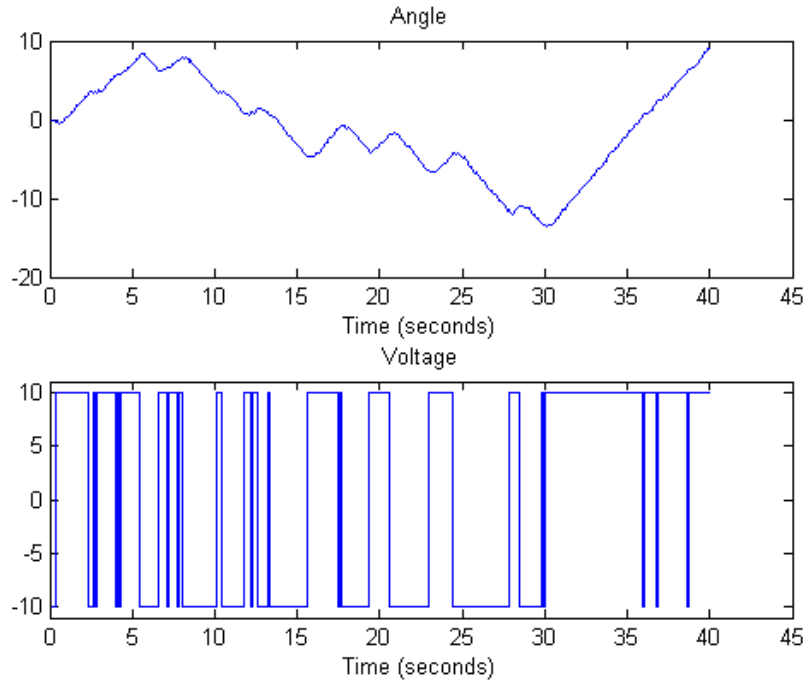


Figure: Measurement Data: Voltage to Angle

```
plot(z(:,2,:))
```

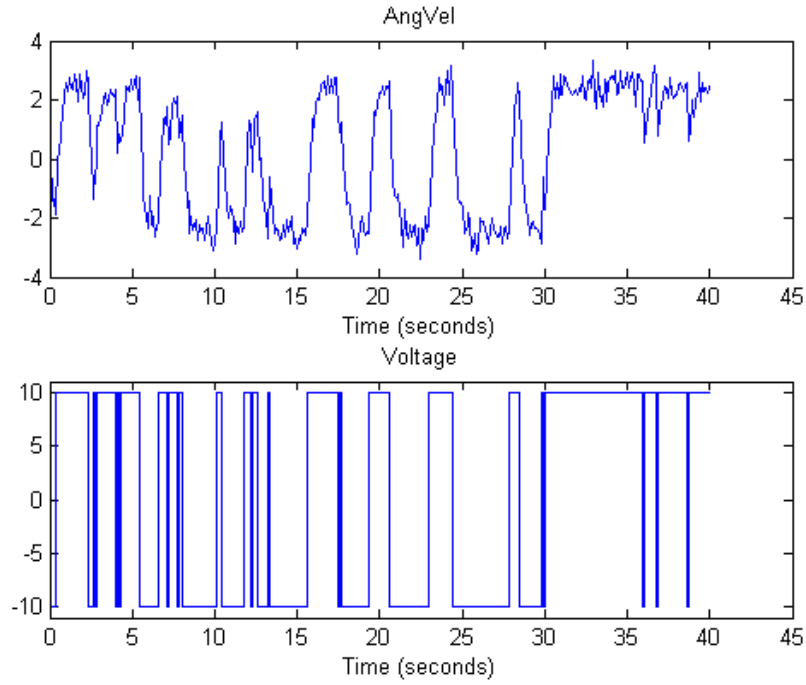


Figure: Measurement Data: Voltage to Angle

Model Structure Selection

```
d/dt x = A x + B u + K e
   y   = C x + D u + e
```

We shall build a model of the dc-motor. The dynamics of the motor is well known. If we choose x1 as the angular position and x2 as the angular velocity it is easy to set up a state-space model of the following character neglecting disturbances: (see Example 4.1 in Ljung(1999):

```
          | 0     1  |       | 0   |
d/dt x =  |          | x  +  |     | u
          | 0   -th1 |       | th2 |

         | 1  0 |
```

```
y =  |      | x
     | 0  1 |
```

The parameter `th1` is here the inverse time-constant of the motor and `th2` is such that `th2/th1` is the static gain from input to the angular velocity. (See Ljung(1987) for how `th1` and `th2` relate to the physical parameters of the motor). We shall estimate these two parameters from the observed data. The model structure (parameterized state space) described above can be represented in MATLAB using IDSS and IDGREY models. These models let you perform estimation of parameters using experimental data.

Specification of a Nominal (Initial) Model

If we guess that th1=1 and th2 = 0.28 we obtain the nominal or initial model

```
A = [0 1; 0 -1]; %initial guess for A(2,2) is -1
B = [0; 0.28]; %initial guess for B(2) is 0.28
C = eye(2);
D = zeros(2,1);
```

and we package this into an IDSS model object:

```
ms = idss(A,B,C,D);
```

The model is characterized by its matrices, their values, which elements are free (to be estimated) and upper and lower limits of those:

```
ms.Structure.a
```

```
ans =

       Name: 'a'
      Value: [2x2 double]
    Minimum: [2x2 double]
    Maximum: [2x2 double]
       Free: [2x2 logical]
      Scale: [2x2 double]
       Info: [2x2 struct]
```

```
1x1 param.Continuous


ms.Structure.a.Value
ms.Structure.a.Free


ans =

     0     1
     0    -1


ans =

     1     1
     1     1
```

Specification of Free (Independent) Parameters Using IDSS Models

So we should now mark that it is only A(2,2) and B(2,1) that are free parameters to be estimated.

```
ms.Structure.a.Free = [0 0; 0 1];
ms.Structure.b.Free = [0; 1];
ms.Structure.c.Free = 0; % scalar expansion used
ms.Structure.d.Free = 0;
set(ms,'Ts',0);  % This defines the model to be continuous
```

The Initial Model

```
ms % Initial model


ms =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)
```

```
  A =
       x1  x2
   x1   0   1
   x2   0  -1

  B =
         u1
   x1     0
   x2  0.28

  C =
       x1  x2
   y1   1   0
   y2   0   1

  D =
       u1
   y1   0
   y2   0

  K =
       y1  y2
   x1   0   0
   x2   0   0

Parameterization:
   STRUCTURED form (some fixed coefficients in  A, B, C).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 2
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainti

Status:
Created by direct construction or transformation. Not estimated.
```

Estimation of Free Parameters of the IDSS Model

The prediction error (maximum likelihood) estimate of the parameters is now computed by:

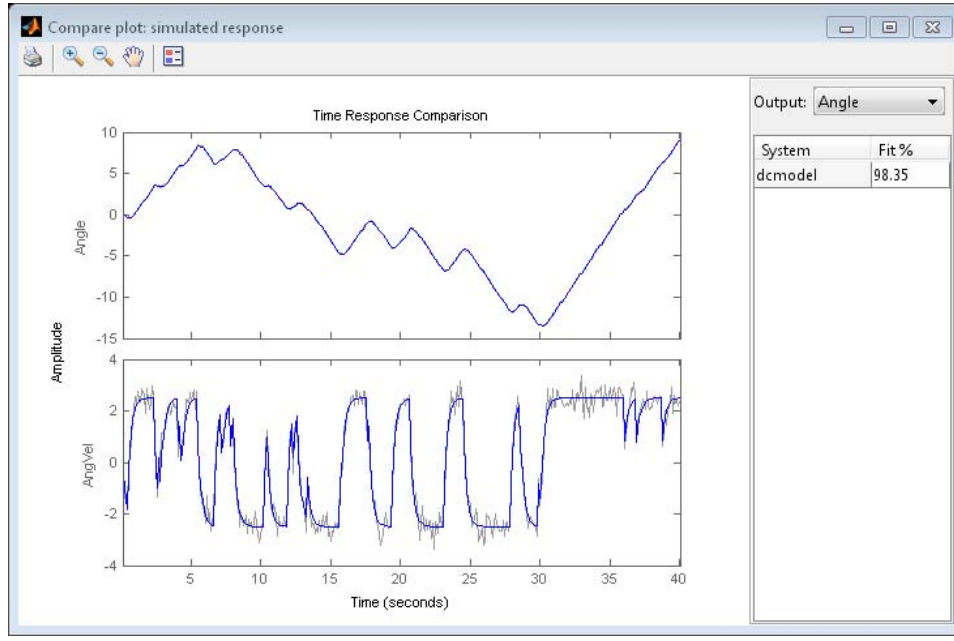
```
dcmodel = ssest(z,ms,ssestOptions('Display','on'));
dcmodel
```

```
dcmodel =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
           x1      x2
   x1       0       1
   x2       0  -4.013

  B =
       Voltage
   x1        0
   x2    1.002

  C =
           x1  x2
   Angle    1   0
   AngVel   0   1

  D =
           Voltage
   Angle         0
   AngVel        0

  K =
        Angle  AngVel
   x1       0       0
   x2       0       0

Parameterization:
   STRUCTURED form (some fixed coefficients in  A, B, C).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 2
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainti
```

```
Status:
Estimated using SSEST on time domain data "z".
Fit to estimation data: [98.35;84.42]% (prediction focus)
FPE: 0.001071, MSE: 0.05967
```

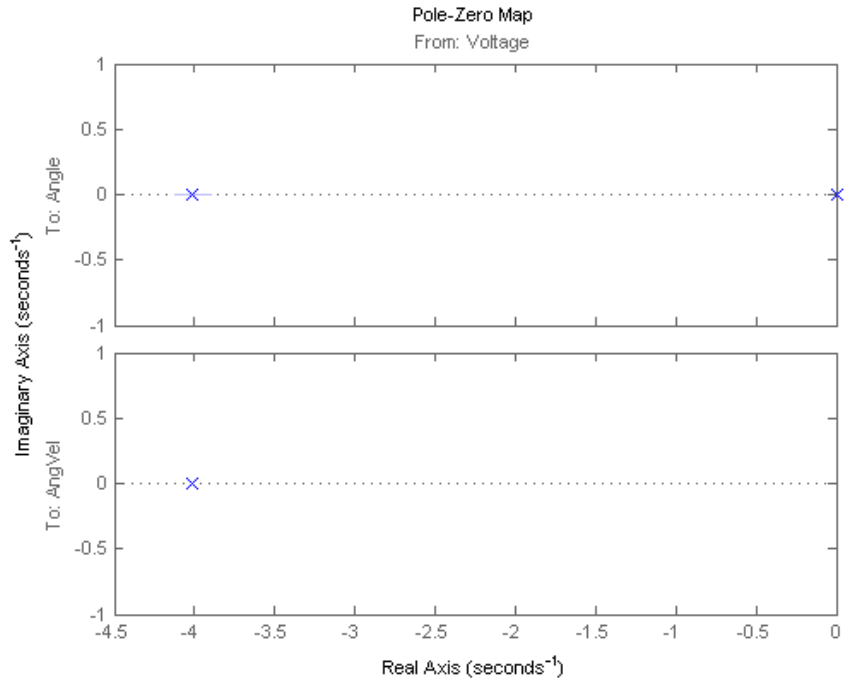
The estimated values of the parameters are quite close to those used when the data were simulated (-4 and 1). To evaluate the model's quality we can simulate the model with the actual input by and compare it with the actual output.

```
compare(z,dcmodel);
```



We can now, for example plot zeros and poles and their uncertainty regions. We will draw the regions corresponding to 3 standard deviations, since the model is quite accurate. Note that the pole at the origin is absolutely certain, since it is part of the model structure; the integrator from angular velocity to position.

```
clf
showConfidence(iopzplot(dcmodel),3)
```



Now, we may make various modifications. The 1,2-element of the A-matrix (fixed to 1) tells us that `x2` is the derivative of `x1`. Suppose that the sensors are not calibrated, so that there may be an unknown proportionality constant. To include the estimation of such a constant we just "let loose" `A(1,2)` and re-estimate:

```
dcmodel2 = dcmodel;
dcmodel2.Structure.a.Free(1,2) = 1;
dcmodel2 = pem(z,dcmodel2,ssestOptions('Display','on'));
```

The resulting model is

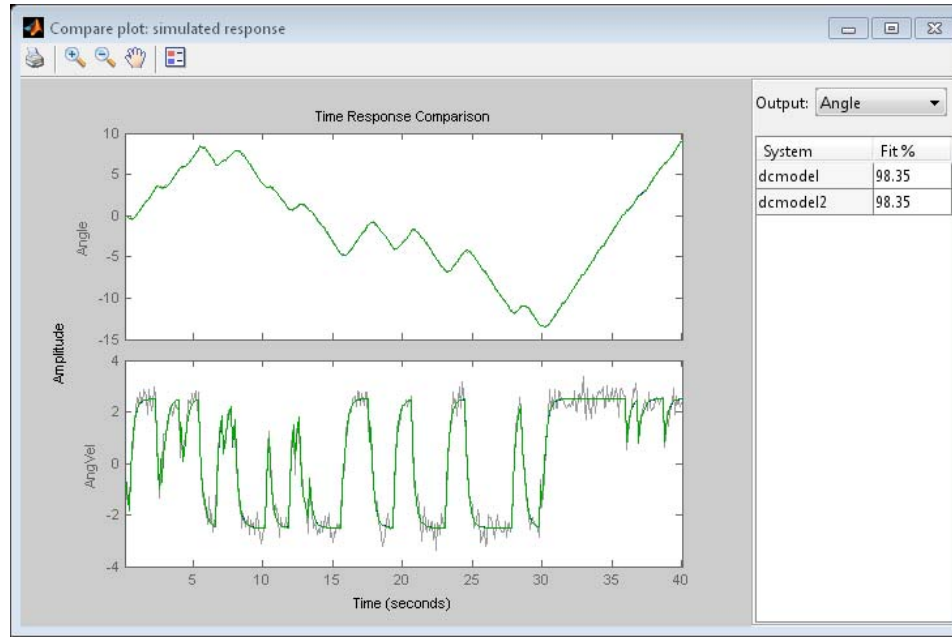
```
dcmodel2
```

```
dcmodel2 =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
           x1      x2
   x1       0  0.9975
   x2       0  -4.011

  B =
       Voltage
   x1        0
   x2    1.004

  C =
           x1  x2
   Angle    1   0
   AngVel   0   1

  D =
           Voltage
   Angle         0
   AngVel        0

  K =
        Angle  AngVel
   x1       0       0
   x2       0       0

Parameterization:
   STRUCTURED form (some fixed coefficients in  A, B, C).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 3
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainti

Status:
Estimated using SSEST on time domain data "z".
```

```
Fit to estimation data: [98.35;84.42]% (prediction focus)
FPE: 0.001076, MSE: 0.05966
```

We find that the estimated `A(1,2)` is close to 1. To compare the two model we use the `compare` command:

```
compare(z,dcmodel,dcmodel2)
```



Specification of Models with Coupled Parameters Using IDGREY Objects

Suppose that we accurately know the static gain of the dc-motor (from input voltage to angular velocity, e.g. from a previous step-response experiment. If the static gain is `G`, and the time constant of the motor is t, then the state-space model becomes

```
          |0     1|     |  0  |
d/dt x =  |       |x +  |     | u
          |0  -1/t|     | G/t |

          |1   0|
```

```
   y   =  |     | x
          |0   1|
```

With `G` known, there is a dependence between the entries in the different matrices. In order to describe that, the earlier used way with "Free" parameters will not be sufficient. We thus have to write a MATLAB file which produces the A, B, C, and D, and optionally also the K and X0 matrices as outputs, for each given parameter vector as input. It also takes auxiliary arguments as inputs, so that the user can change certain things in the model structure, without having to edit the file. In this case we let the known static gain `G` be entered as such an argument. The file that has been written has the name 'motor.m'.

```
type motor
```

```
function [A,B,C,D,K,X0] = motor(par,ts,aux)
%MOTOR ODE file representing the dynamics of a motor.
%
%   [A,B,C,D,K,X0] = MOTOR(Tau,Ts,G)
%   returns the State Space matrices of the DC-motor with
%   time-constant Tau (Tau = par) and known static gain G. The sample
%   time is Ts.
%
%   This file returns continuous-time representation if input argument Ts
%   is zero. If Ts>0, a discrete-time representation is returned. To make
%   the IDGREY model that uses this file aware of this flexibility, set the
%   value of Structure.FcnType property to 'cd'. This flexibility is useful
%   for conversion between continuous and discrete domains required for
%   estimation and simulation.
%
%   See also IDGREY, IDDEMO7.

%   L. Ljung
%   Copyright 1986-2011 The MathWorks, Inc.
%   $Revision: 1.1.10.2 $  $Date: 2011/12/22 18:22:34 $

t = par(1);
G = aux(1);
```

```
A = [0 1;0 -1/t];
B = [0;G/t];
C = eye(2);
D = [0;0];
K = zeros(2);
X0 = [0;0];
if ts>0 % Sample the model with sample time Ts
   s = expm([[A B]*ts; zeros(1,3)]);
   A = s(1:2,1:2);
   B = s(1:2,3);
end
```

We now create an IDGREY model object corresponding to this model structure: The assumed time constant will be

```
par_guess = 1;
```

We also give the value 0.25 to the auxiliary variable `G` (gain) and sampling interval.

```
aux = 0.25;
dcmm = idgrey('motor',par_guess,'cd',aux,0);
```

The time constant is now estimated by

```
dcmm = greyest(z,dcmm,greyestOptions('Display','on'));
```

We have thus now estimated the time constant of the motor directly. Its value is in good agreement with the previous estimate.

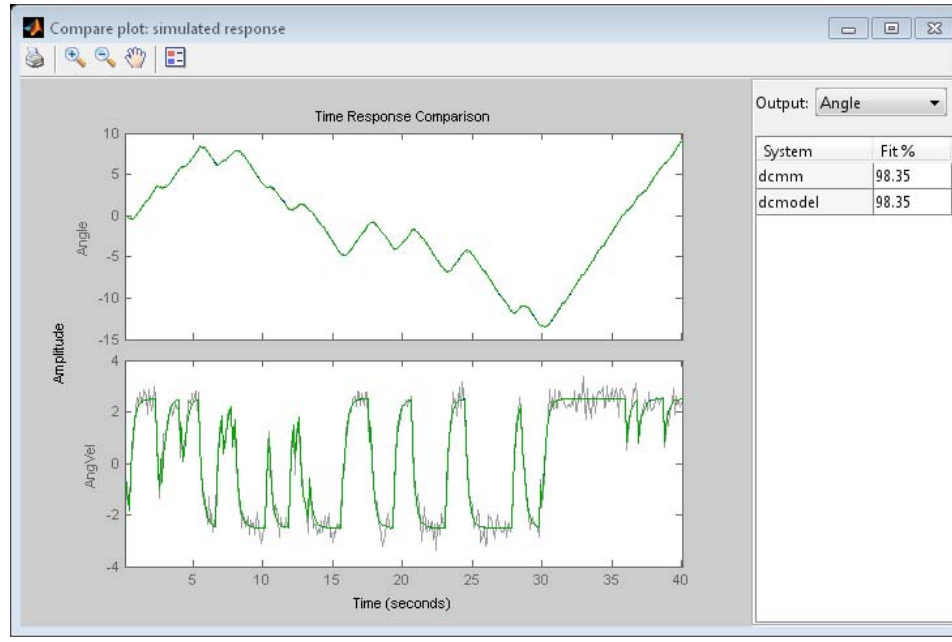
```
dcmm
```

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

  A =
           x1      x2
```

```
    x1       0       1
    x2       0  -4.006

  B =
         Voltage
    x1         0
    x2     1.001

  C =
            x1  x2
   Angle     1   0
   AngVel    0   1

  D =
           Voltage
   Angle         0
   AngVel        0

  K =
         Angle  AngVel
    x1       0       0
    x2       0       0

  Model parameters:
   Par1 = 0.2496

Parameterization:
   ODE Function: motor
   (parameterizes both continuous- and discrete-time equations)
   Disturbance component: parameterized by the ODE function
   Initial state: parameterized by the ODE function
   Number of free coefficients: 1
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using GREYEST on time domain data "z".
Fit to estimation data: [98.35;84.42]% (prediction focus)
FPE: 0.00107, MSE: 0.05971
```

With this model we can now proceed to test various aspects as before. The syntax of all the commands is identical to the previous case. For example, we can compare the idgrey model with the other state-space model:

```
compare(z,dcmm,dcmodel)
```



They are clearly very close.

Estimating Multivariate ARX Models

The state-space part of the toolbox also handles multivariate (several outputs) ARX models. By a multivariate ARX-model we mean the following:

```
A(q) y(t) = B(q) u(t) + e(t)
```

Here A(q) is a ny | ny matrix whose entries are polynomials in the delay operator 1/q. The k-l element is denoted by:

$$a_{kl}(q)$$

where:

$$a_{kl}(q) = 1 + a_1 q^{-1} + ... + a_{nakl} q^{-nakl} q$$

It is thus a polynomial in `1/q` of degree `nakl`.

Similarly B(q) is a ny | nu matrix, whose kj-element is:

$$b_{kj}(q) = b_0 q^{-nkk} + b_1 q^{-nkkj-1} + ... + b_{nbkj} q^{-nkkj-nbkj}$$

There is thus a delay of `nkkj` from input number `j` to output number `k`. The most common way to create those would be to use the ARX-command. The orders are specified as: `nn = [na nb nk]` with `na` being a `ny-by-ny` matrix whose `kj`-entry is `nakj`; `nb` and `nk` are defined similarly.

Let's test some ARX-models on the dc-data. First we could simply build a general second order model:

```
dcarx1 = arx(z,'na',[2,2;2,2],'nb',[2;2],'nk',[1;1])
```

```
dcarx1 =
Discrete-time ARX model:
  Model for output "Angle": A(z)y_1(t) = - A_i(z)y_i(t) + B(z)u(t) + e_1(t)
    A(z) = 1 - 0.5545 z^-1 - 0.4454 z^-2

    A_2(z) = -0.03548 z^-1 - 0.06405 z^-2

    B(z) = 0.004243 z^-1 + 0.006589 z^-2

  Model for output "AngVel": A(z)y_2(t) = - A_i(z)y_i(t) + B(z)u(t) + e_2(t
    A(z) = 1 - 0.2005 z^-1 - 0.2924 z^-2

    A_1(z) = 0.01849 z^-1 - 0.01937 z^-2

    B(z) = 0.08642 z^-1 + 0.03877 z^-2

Sample time: 0.1 seconds

Parameterization:
```

```
   Polynomial orders:   na=[2 2;2 2]   nb=[2;2]   nk=[1;1]
   Number of free coefficients: 12
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainti

Status:
Estimated using ARX on time domain data "z".
Fit to estimation data: [97.87;83.44]% (prediction focus)
FPE: 0.002197, MSE: 0.06999
```

The result, `dcarx1`, is stored as an IDPOLY model, and all previous commands apply. We could for example explicitly list the ARX-polynomials by:

```
dcarx1.a
```

```
ans =

    [1x3 double]    [1x3 double]
    [1x3 double]    [1x3 double]
```

as cell arrays where e.g. the {1,2} element of dcarx1.a is the polynomial A(1,2) described earlier, relating y2 to y1.

We could also test a structure, where we know that `y1` is obtained by filtering `y2` through a first order filter. (The angle is the integral of the angular velocity). We could then also postulate a first order dynamics from input to output number 2:

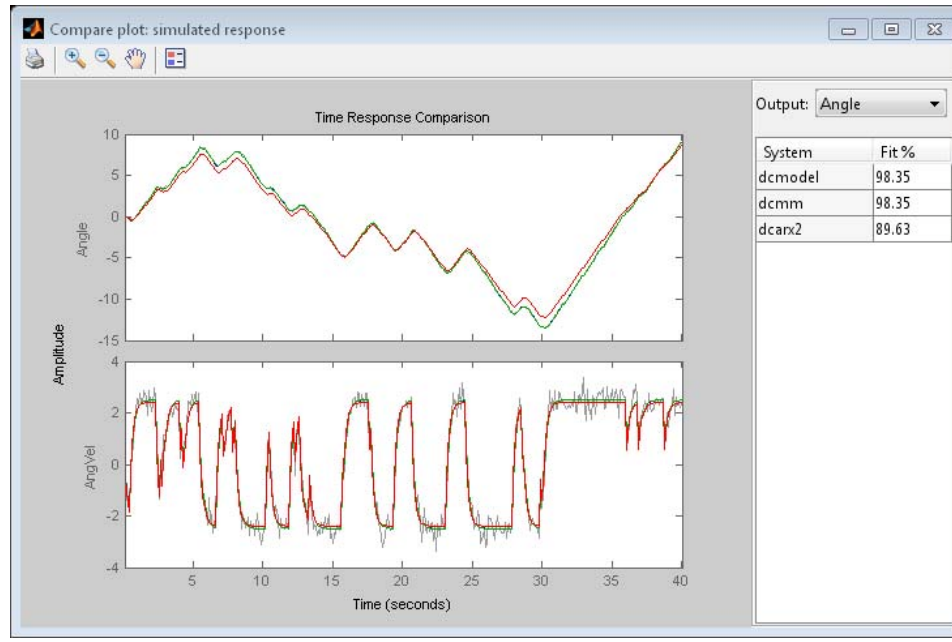
```
na = [1 1; 0 1];
nb = [0 ; 1];
nk = [1 ; 1];
dcarx2 = arx(z,[na nb nk])
```

```
dcarx2 =
Discrete-time ARX model:
  Model for output "Angle": A(z)y_1(t) = - A_i(z)y_i(t) + B(z)u(t) + e_1(t)
    A(z) = 1 - 0.9992 z^-1
```

```
    A_2(z) = -0.09595 z^-1

    B(z) = 0

  Model for output "AngVel": A(z)y_2(t) = B(z)u(t) + e_2(t)
    A(z) = 1 - 0.6254 z^-1

    B(z) = 0.08973 z^-1

Sample time: 0.1 seconds

Parameterization:
   Polynomial orders:   na=[1 1;0 1]   nb=[0;1]   nk=[1;1]
   Number of free coefficients: 4
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainti

Status:
Estimated using ARX on time domain data "z".
Fit to estimation data: [97.52;81.46]% (prediction focus)
FPE: 0.003468, MSE: 0.08862
```

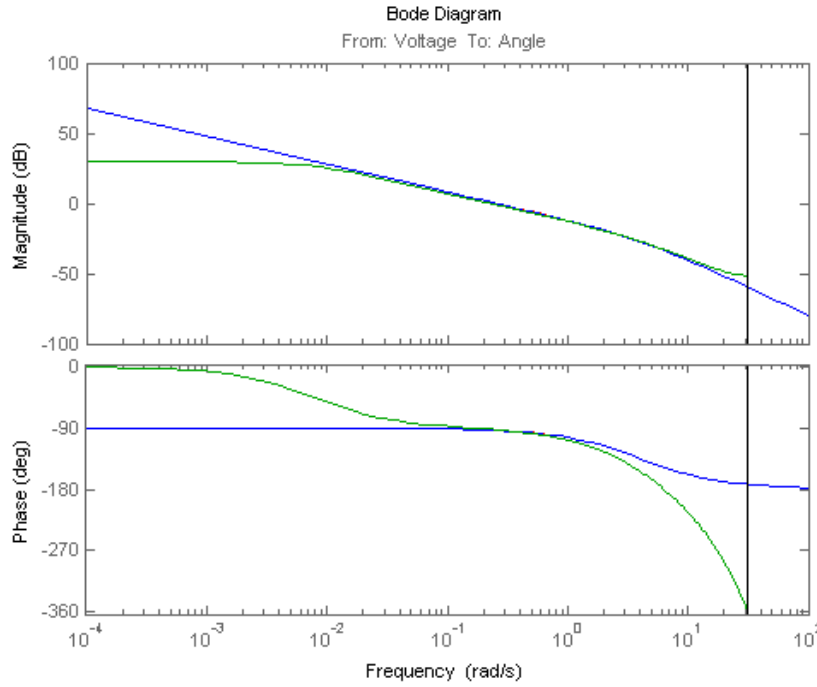
To compare the different models obtained we use

```
compare(z,dcmodel,dcmm,dcarx2)
```



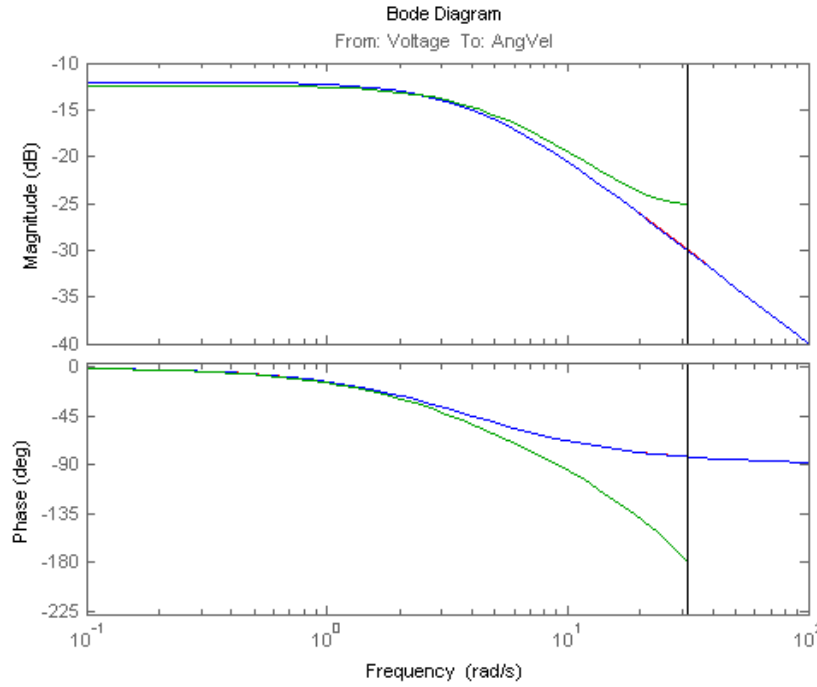
Finally, we could compare the bodeplots obtained from the input to output one for the different models by using `bode`: First output:

```
dcmm2 = idss(dcmm); % convert to IDSS for subreferencing
bode(dcmodel(1,1),'r',dcmm2(1,1),'b',dcarx2(1,1),'g')
```



Second output:

```
bode(dcmodel(2,1),'r',dcmm2(2,1),'b',dcarx2(2,1),'g')
```



The two first models are more or less in exact agreement. The ARX-models are not so good, due to the bias caused by the non-white equation error noise. (We had white measurement noise in the simulations).

Conclusions

Estimation of models with pre-selected structures can be performed using System Identification toolbox. In state-space form, parameters may be fixed to their known values or constrained to lie within a prescribed range. If relationship between parameters or other constraints need to be specified, IDGREY objects may be used. IDGREY models evaluate a user-specified MATLAB file for estimating state-space system parameters. Multi-variate ARX models offer another option for quickly estimating multi-output models with user-specified structure.

Additional Information

For more information on identification of dynamic systems with System Identification Toolbox visit the System Identification Toolbox product information page.

4

Nonlinear Black-Box Model Identification

About Nonlinear Model Identification

In this section...

What Are Nonlinear Models?

Dynamic models in System Identification Toolbox software are mathematical relationships between the system's inputs *u(t)* and outputs *y(t)*. You can use these relationships to compute the current output from previous inputs and outputs. The general form of a model in discrete time is:

$$y(t) = f(u(t - 1), y(t - 1), u(t - 2), y(t - 2), \ldots)$$

Such a model is nonlinear if the function *f* is a nonlinear function. *f* may represent arbitrary nonlinearities, such as switches and saturations.

When to Fit Nonlinear Models

In practice, all systems are nonlinear and the output is a nonlinear function of the input variables. However, a linear model is often sufficient to accurately describe the system dynamics. In most cases, you should first try to fit linear models.

Here are some scenarios when you might need the additional flexibility of nonlinear models:

- "Linear Model Is Not Good Enough" on page 4-3
- "Physical System Is Weakly Nonlinear" on page 4-3
- "Physical System Is Inherently Nonlinear" on page 4-3
- "Linear and Nonlinear Dynamics Are Captured Separately" on page 4-4

Linear Model Is Not Good Enough

You might need nonlinear models when a linear model provides a poor fit to the measured output signals and cannot be improved by changing the model structure or order. Nonlinear models have more flexibility in capturing complex phenomena than the linear models of similar orders.

Physical System Is Weakly Nonlinear

From physical insight or data analysis, you might know that a system is weakly nonlinear. In such cases, you can estimate a linear model and then use this model as an initial model for nonlinear estimation. Nonlinear estimation can improve the fit by using nonlinear components of the model structure to capture the dynamics not explained by the linear model. For more information, see "Using Linear Model for Nonlinear ARX Estimation" on page 4-28 and "Using Linear Model for Hammerstein-Wiener Estimation" on page 4-63.

Physical System Is Inherently Nonlinear

You might have physical insight that your system is nonlinear. Certain phenomena are inherently nonlinear in nature, including dry friction in mechanical systems, actuator power saturation, and sensor nonlinearities in electro-mechanical systems. You can try modeling such systems using the Hammerstein-Wiener model structure, which lets you interconnect linear models with static nonlinearities. For more information, see "Identifying Hammerstein-Wiener Models" on page 4-48.

Nonlinear models might be necessary to represent systems that operate over a range of operating points. In some cases, you might fit several linear models, where each model is accurate at specific operating conditions. You can also try using the nonlinear ARX model structure with tree partitions to model such systems. For more information, see "Identifying Nonlinear ARX Models" on page 4-8.

If you know the nonlinear equations describing a system, you can represent this system as a nonlinear grey-box model and estimate the coefficients from experimental data. In this case, the coefficients are the parameters of the model. For more information, see "Grey-Box Model Estimation".

Before fitting a nonlinear model, try transforming your input and output variables such that the relationship between the transformed variables becomes linear. For example, you might be dealing with a system that has current and voltage as inputs to an immersion heater, and the temperature of the heated liquid as an output. In this case, the output depends on the inputs via the power of the heater, which is equal to the product of current and voltage. Instead of fitting a nonlinear model to two-input and one-output data, you can create a new input variable by taking the product of current and voltage and then fitting a linear model to the single-input/single-output data.

Linear and Nonlinear Dynamics Are Captured Separately

You might have multiple data sets that capture the linear and nonlinear dynamics separately. For example, one data set with low amplitude input (excites the linear dynamics only) and another data set with high amplitude input (excites the nonlinear dynamics). In such cases, first estimate a linear model using the first data set. Next, use the model as an initial model to estimate a nonlinear model using the second data set. For more information, see "Using Linear Model for Nonlinear ARX Estimation" on page 4-28 and "Using Linear Model for Hammerstein-Wiener Estimation" on page 4-63.

Available Nonlinear Models

System Identification Toolbox supports these nonlinear models:

- "Nonlinear ARX Models" on page 4-4
- "Hammerstein-Wiener Models" on page 4-5
- "Nonlinear State-Space Models" on page 4-5

Nonlinear ARX Models

Nonlinear ARX models extend the linear ARX models to the nonlinear case and have this structure:

$$y(t) = f(y(t - 1), ..., y(t - na), u(t - nk), ..., u(t - nk - nb + 1))$$

where the function f depends on a finite number of previous inputs u and outputs y. na is the number of past output terms and nb is the number of past

input terms used to predict the current output. *nk* is the delay from the input to the output, specified as the number of samples.

Typically, you use nonlinear ARX models as black-box structures. The nonlinear function of the nonlinear ARX model is a flexible nonlinearity estimator with parameters that need not have physical significance.

System Identification Toolbox software uses `idnlarx` objects to represent nonlinear ARX models. For more information about estimation, see "Nonlinear ARX Models".

Hammerstein-Wiener Models

Hammerstein-Wiener models describe dynamic systems using one or two static nonlinear blocks in series with a linear block. The linear block is a discrete transfer function and represents the dynamic component of the model.

You can use the Hammerstein-Wiener structure to capture physical nonlinear effects in sensors and actuators that affect the input and output of a linear system, such as dead zones and saturation. Alternatively, use Hammerstein-Wiener structures as black box structures that do not represent physical insight into system processes.

System Identification Toolbox software uses `idnlhw` objects to represent Hammerstein-Wiener models. For more information about estimation, see "Hammerstein-Wiener Models".

Nonlinear State-Space Models

Nonlinear state-space models have this representation:

$$\dot{x}(t) = F\left(x(t), u(t)\right)$$
$$y(t) = H\left(x(t), u(t)\right)$$

where F and H can have any parameterization. A nonlinear ordinary differential equation of high order can be represented as a set of first order equations. You use the `idnlgrey` object to specify the structures of such models based on physical insight about your system. The parameters of such models typically have physical interpretations.

For more information about estimating nonlinear state-space models, see "Grey-Box Model Estimation".

Preparing Data for Nonlinear Identification

Estimating nonlinear ARX and Hammerstein-Wiener models requires uniformly sampled time-domain data. Your data can have one or more input and output channels.

For time-series data, you can only fit nonlinear ARX models and nonlinear state-space models.

Tip Whenever possible, use different data sets for model estimation and validation.

Before estimating models, import your data into the MATLAB workspace and do *one* of the following:

- **In the System Identification Tool GUI.** Import data into the GUI, as described in "Importing Data into the GUI" on page 2-17.
- **At the command line.** Represent your data as an `iddata` object, as described in the corresponding reference page.

You can analyze data quality and preprocess data by interpolating missing values, filtering to emphasize a specific frequency range, or resampling using a different sample time (see "Ways to Prepare Data for System Identification" on page 2-6).

Data detrending can be useful in certain cases, such as before modeling the relationship between the change in input and the change in output about an operating point. However, most applications do not require you to remove offsets and linear trends from the data before nonlinear modeling.

Identifying Nonlinear ARX Models

In this section...

Nonlinear ARX Model Extends the Linear ARX Structure

A nonlinear ARX model can be understood as an extension of a linear model. A linear SISO ARX model has this structure:

$$y(t) + a_1 y(t-1) + a_2 y(t-2) + \ldots + a_{na} y(t-na) = b_1 u(t) + b_2 u(t-1) + \ldots + b_{nb} u(t-nb+1) + e(t)$$

where the input delay nk is zero to simplify the notation.

This structure implies that the current output $y(t)$ is predicted as a weighted sum of past output values and current and past input values. Rewriting the equation as a product:

$$y_p(t) = [-a_1, -a_2, \ldots, -a_{na}, b_1, b_2, .., b_{nb}] * [y(t-1), y(t-2), \ldots, y(t-na), u(t), u(t-1), \ldots, u(t-nb-1)]^T$$

where $y(t-1), y(t-2), \ldots, y(t-na), u(t), u(t-1), \ldots, u(t-nb-1)$ are delayed input and output variables, called *regressors*. The linear ARX model thus predicts the current output y_p as a weighted sum of its regressors.

This structure can be extended to create a nonlinear form as:

- Instead of the weighted sum that represents a linear mapping, the nonlinear ARX model has a more flexible nonlinear mapping function:

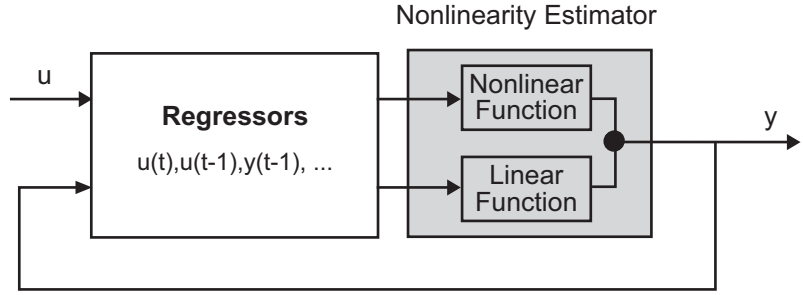
  $$y_p(t) = f(y(t-1), y(t-2), y(t-3), \ldots, u(t), u(t-1), u(t-2), \ldots)$$

  where f is a nonlinear function. Inputs to f are model regressors. When you specify the nonlinear ARX model structure, you can choose one of several available nonlinear mapping functions in this toolbox (see "Nonlinearity Estimators for Nonlinear ARX Models" on page 4-10).

- Nonlinear ARX regressors can be both delayed input-output variables and more complex, nonlinear expressions of delayed input and output variables. Examples of such nonlinear regressors are $y(t\text{-}1)^2$, $u(t\text{-}1)*y(t\text{-}2)$, $\tan(u(t\text{-}1))$, and $u(t\text{-}1)*y(t\text{-}3)$.

Structure of Nonlinear ARX Models

This block diagram represents the structure of a nonlinear ARX model in a simulation scenario:



The nonlinear ARX model computes the output y in two stages:

1. Computes regressors from the current and past input values and past output data.

In the simplest case, regressors are delayed inputs and outputs, such as *u(t-1)* and *y(t-3)*—called *standard* regressors. You can also specify *custom* regressors, which are nonlinear functions of delayed inputs and outputs. For example, tan(*u*(*t*-1)) or *u*(*t*-1)***y*(*t*-3).

By default, all regressors are inputs to both the linear and the nonlinear function blocks of the nonlinearity estimator. You can choose a subset of regressors as inputs to the nonlinear function block.

2 The nonlinearity estimator block maps the regressors to the model output using a combination of nonlinear and linear functions. You can select from available nonlinearity estimators, such as tree-partition networks, wavelet networks, and multi-layer neural networks. You can also exclude either the linear or the nonlinear function block from the nonlinearity estimator.

The nonlinearity estimator block can include linear and nonlinear blocks in parallel. For example:

$$F(x) = L^T(x - r) + d + g\left(Q(x - r)\right)$$

x is a vector of the regressors. $L^T(x) + d$ is the output of the linear function block and is affine when d≠0. d is a scalar offset. $g\left(Q(x - r)\right)$ represents the output of the nonlinear function block. r is the mean of the regressors x. Q is a projection matrix that makes the calculations well conditioned. The exact form of $F(x)$ depends on your choice of the nonlinearity estimator.

Estimating a nonlinear ARX model computes the model parameter values, such as L, r, d, Q, and other parameters specifying g. Resulting models are `idnlarx` objects that store all model data, including model regressors and parameters of the nonlinearity estimator. See the `idnlarx` reference page for more information.

Nonlinearity Estimators for Nonlinear ARX Models

System Identification Toolbox software provides several nonlinearity estimators $F(x)$ for nonlinear ARX models. For more information about $F(x)$, see “Structure of Nonlinear ARX Models” on page 4-9.

Each nonlinearity estimator corresponds to an object class in this toolbox. When you estimate nonlinear ARX models in the GUI, System Identification Toolbox creates and configures objects based on these classes. You can also create and configure nonlinearity estimators at the command line.

Most nonlinearity estimators represent the nonlinear function as a summed series of nonlinear units, such as wavelet networks or sigmoid functions. You can configure the number of nonlinear units n for estimation. For a detailed description of each estimator, see the references page of the corresponding nonlinearity class.

Nonlinearity	Class	Structure	Comments
Wavelet network (default)	`wavenet`	$g(x) = \sum_{k=1}^{n} \alpha_k \kappa\left(\beta_k (x - \gamma_k)\right)$ where $\kappa(s)$ is the wavelet function.	By default, the estimation algorithm determines the number of units n automatically.
One layer sigmoid network	`sigmoidnet`	$g(x) = \sum_{k=1}^{n} \alpha_k \kappa\left(\beta_k (x - \gamma_k)\right)$ where $\kappa(s) = \left(e^s + 1\right)^{-1}$ is the sigmoid function. β_k is a row vector such that $\beta_k(x - \gamma_k)$ is a scalar.	Default number of units n is 10.

Nonlinearity	Class	Structure	Comments
Tree partition	`treepartition`	Piecewise linear function over partitions of the regressor space defined by a binary tree.	The estimation algorithm determines the number of units automatically. Try using tree partitions for modeling data collected over a range of operating conditions.
F is linear in x	`linear`	This estimator produces a model that is similar to the linear ARX model, but offers the additional flexibility of specifying custom regressors.	Use to specify custom regressors as the nonlinearity estimator and exclude a nonlinearity mapping function.
Multilayered neural network	`neuralnet`	Uses as a `network` object created using the Neural Network Toolbox™ software.	
Custom network (user-defined)	`customnet`	Similar to sigmoid network but you specify $\kappa(s)$.	(For advanced use) Uses the unit function that you specify.

Ways to Configure Nonlinear ARX Estimation

Configurable Elements of Nonlinear ARX Structure

You can adjust various elements of the nonlinear ARX model structure and fit different models to your data.

Configure model regressors by:

- Specifying model order and delay, which creates the set of standard regressors.

  For a definition, see "Nonlinear ARX Order and Delay" on page 4-14.

- Creating custom regressors.

  Custom regressors are arbitrary functions of past inputs and outputs, such as products, powers, and other MATLAB expressions of input and output variables. You can specify custom regressors in addition to or instead of standard regressors for greater flexibility in modeling your data.

- Including a subset of regressors in the nonlinear function of the nonlinear estimator block.

  Selecting which regressors are inputs to the nonlinear function reduces model complexity and keeps the estimation well-conditioned.

- Initializing using a linear ARX model.

  You can perform this operation only at the command line. The initialization configures the nonlinear ARX model to use standard regressors, which the toolbox computes using the orders and delays of the linear model. See "Using Linear Model for Nonlinear ARX Estimation" on page 4-28.

Configure the nonlinearity estimator block by:

- Specifying and configuring the nonlinear function, including the number of units.
- Excluding the nonlinear function from the nonlinear estimator such that $F(x) = L^T(x) + d$.
- Excluding the linear function from the nonlinear estimator such that $F(x) = g(Q(x - r))$.

Note You cannot exclude the linear function from tree partitions and neural networks.

See these topics for detailed steps to change the model structure:

- "How to Estimate Nonlinear ARX Models in the GUI" on page 4-16
- "How to Estimate Nonlinear ARX Models at the Command Line" on page 4-19

Default Nonlinear ARX Structure

Estimate a nonlinear ARX model with default configuration by one of the following:

- Specifying only model order and input delay. Specifying the order automatically creates standard regressors.
- Specifying a linear ARX model. The linear model sets the model orders and linear function of the nonlinear model. You can perform this operation only at the command line.

By default:

- The nonlinearity estimator is a wavelet network (see the `wavenet` reference page).

  This nonlinearity often provides satisfactory results and uses a fast estimation method.
- All of the standard regressors are inputs to the linear and nonlinear functions of the wavelet network.

Nonlinear ARX Order and Delay

The order and delay of nonlinear ARX models are positive integers:

- *na* — Number of past output terms used to predict the current output.
- *nb* — Number of past input terms used to predict the current output.

- *nk* — Delay from input to the output in terms of the number of samples.

The meaning of *na*, *nb*, and *nk* is similar to linear ARX model parameters. Orders are scalars for SISO data, and matrices for MIMO data. If you are not sure how to specify the order and delay, you can estimate them as described in "Preliminary Step – Estimating Model Orders and Input Delays" on page 3-53. Such an estimate is based on linear ARX models and only provides initial guidance—the best orders for a linear ARX model might not be the best orders for a nonlinear ARX model.

System Identification Toolbox software computes standard regressors using model orders.

For example, if you specify this order and delay for a SISO model with input *u* and output *y*:

na=2, *nb*=3, and *nk*=5

the toolbox computes standard regressors *y(t-2)*, *y(t-1)*, *u(t-5)*, *u(t-6)*, and *u(t-7)*.

You can specify custom regressors in addition to standard regressors, as described in "How to Estimate Nonlinear ARX Models in the GUI" on page 4-16 and "How to Estimate Nonlinear ARX Models at the Command Line" on page 4-19.

Estimation Algorithm for Nonlinear ARX Models

The estimation algorithm depends on your choice of nonlinearity estimator and other properties of the `idnlarx` class. You can set algorithm properties both in the GUI and at the command line.

Focus property of idnlarx class. By default, estimating nonlinear ARX models minimizes one-step prediction errors, which corresponds to `Focus` value of `Prediction`.

If you want a model that is optimized for reproducing simulation behavior, try setting the `Focus` value to `Simulation`. In this case, you cannot use `treepartition` and `neuralnet` because these nonlinearity estimators are not differentiable. Minimization of simulation error requires differentiable

nonlinear functions. Simulation error minimization takes more time than one-step-ahead prediction error minimization.

Common algorithm properties of idnlarx class.

- `MaxIter` — Maximum number of iterations.
- `SearchMethod` — Search method for minimization of prediction or simulation errors, such as Gauss-Newton and Levenberg-Marquardt line search, and Trust-region reflective Newton approach. By default, the algorithm uses a combination of these methods.
- `Tolerance` — Condition for terminating iterative search when the expected improvement of the parameter values is less than a specified value.
- `Display` — Shows progress of iterative minimization in the MATLAB Command Window.

How to Estimate Nonlinear ARX Models in the GUI

Prerequisites

- Learn about the nonlinear ARX model structure (see "Structure of Nonlinear ARX Models" on page 4-9).
- Import data into the System Identification Tool GUI (see "Preparing Data for Nonlinear Identification" on page 4-7).
- (Optional) Choose a nonlinearity estimator in "Nonlinearity Estimators for Nonlinear ARX Models" on page 4-10.

1. In the System Identification Tool GUI, select **Estimate > Nonlinear models** to open the Nonlinear Models dialog box.
2. In the **Configure** tab, select `Nonlinear ARX` from the **Model type** list.
3. (Optional) Edit the **Model name** by clicking the pencil icon. The name of the model should be unique to all nonlinear ARX models in the System Identification Tool GUI.

4 (Optional) If you want to refine a previously estimated model, click **Initialize** to select a previously estimated model from the **Initial Model** list.

Note Refining a previously estimated model starts with the parameter values of the initial model and uses the same model structure. The algorithm uses default estimation settings unless you specify to use the initial model settings, or change these settings.

The **Initial Model** list includes models that:

- Exist in the System Identification Tool GUI.
- Have the same number of inputs and outputs as the dimensions of the estimation data (selected as **Working Data** in the System Identification Tool GUI).

5 Keep the default settings in the Nonlinear Models dialog box that specify the model structure and the algorithm, or modify these settings:

Note For more information about available options, click **Help** in the Nonlinear Models dialog box to open the GUI help.

What to Configure	Options in Nonlinear Models GUI	Comment
Model order	In the **Regressors** tab, edit the **No. of Terms** corresponding to each input and output channel.	Model order *na* is the output number of terms and *nb* is the input number of terms.
Input delay	In the **Regressors** tab, edit the **Delay** corresponding to an input channel.	If you do not know the input delay value, click **Infer Input Delay**. This action opens the Infer Input Delay

What to Configure	Options in Nonlinear Models GUI	Comment
		dialog box to suggest possible delay values.
Regressors	In the **Regressors** tab, click **Edit Regressors**.	This action opens the Model Regressors dialog box. Use this dialog box to create custom regressors or to include specific regressors in the nonlinear block.
Nonlinearity estimator	In the **Model Properties** tab.	To use all standard and custom regressors in the linear block only, you can exclude the nonlinear block by setting **Nonlinearity** to `None`.
Estimation algorithm	In the **Estimate** tab, click **Algorithm Options**.	

6 Click **Estimate** to add this model to the System Identification Tool GUI.

The **Estimate** tab displays the estimation progress and results.

7 Validate the model response by selecting the desired plot in the **Model Views** area of the System Identification Tool GUI. For more information about validating models, see "How to Plot Nonlinear ARX Plots Using the GUI" on page 4-34.

If you get a poor fit, try changing the model structure or algorithm configuration in step 5.

You can export the model to the MATLAB workspace by dragging it to **To Workspace** in the System Identification Tool GUI.

How to Estimate Nonlinear ARX Models at the Command Line

Prerequisites

- Learn about the nonlinear ARX model structure in "Structure of Nonlinear ARX Models" on page 4-9.
- Prepare your data, as described in "Preparing Data for Nonlinear Identification" on page 4-7.
- (Optional) Estimate model orders and delays the same way you would for linear ARX models. See "Preliminary Step – Estimating Model Orders and Input Delays" on page 3-53.
- (Optional) Choose a nonlinearity estimator in "Nonlinearity Estimators for Nonlinear ARX Models" on page 4-10.
- (Optional) Estimate or construct an linear ARX model for initialization of nonlinear ARX model. See "Using Linear Model for Nonlinear ARX Estimation" on page 4-28.

Estimate model using nlarx.

Use `nlarx` to both construct and estimate a nonlinear ARX model. After each estimation, validate the model by comparing it to other models and simulating or predicting the model response.

Basic Estimation

Start with the simplest estimation using *`m`* = `nlarx(`*`data`*`,[`*`na nb nk`*`])`. For example:

```
m = nlarx(data,[2 2 1]) % na=nb=2 and nk=1
```

By default, the nonlinearity estimator is the wavelet network (see the `wavenet` reference page), which takes all standard regressors as inputs to its linear and nonlinear functions. `m` is an `idnlarx` object.

For MIMO systems, *`nb`*, *`nf`*, and *`nk`* are *ny*-by-*nu* matrices. See the `nlarx` reference page for more information about MIMO estimation.

Specify a different nonlinearity estimator (for example, sigmoid network):

```
M = nlarx(data,[2 2 1],'sigmoid')
```

Set the `Focus` property of the `idnlarx` object estimation to simulation error minimization:

```
M = nlarx(data,[2 2 1],'sigmoid','Focus','simulation')
```

Configure model regressors.

Standard Regressors

Change the model order to create a model structure with different model regressors, which are delayed input and output variables that are inputs to the nonlinearity estimator.

Custom Regressors

Explore including custom regressors in the nonlinear ARX model structure. Custom regressors are in addition to the standard model regressors (see "Nonlinear ARX Order and Delay" on page 4-14).

Use `polyreg` or `customreg` to construct custom regressors in terms of model input-output variables. You can specify custom regressors using the `CustomRegressors` property of the `idnlarx` class or `addreg` to append custom regressors to an existing model.

For example, generate regressors as polynomial functions of inputs and outputs:

```
  load iddata1
  m = nlarx(z1,[2 2 1],'sigmoidnet');
  getreg(m) % displays all regressors
% Generate polynomial regressors up to order 2:
  reg = polyreg(m)
% Append polynomial regressors to CustomRegressors:
  m = addreg(m,reg);
  getreg(m) % now includes polynomial regressors
```

You can also specify arbitrary functions of input and output variables. For example:

```
  load iddata1
  m = nlarx(z1,[2 2 1],'sigmoidnet',...
            'CustomReg',{'y1(t-1)^2','y1(t-2)*u1(t-3)'});
  getreg(m) % displays all regressors
% Append polynomial regressors to CustomRegressors:
  m = addreg(m,reg);
  getreg(m) % polynomial regressors
```

Manipulate custom regressors using the `CustomRegressors` property of the `idnlarx` class. For example, to get the function handle of the first custom regressor in the array:

```
CReg1 = m.CustomReg(1).Function;
```

To view the regressor expression as a string, use:

```
 m.CustomReg(1).Display
```

You can exclude all standard regressors and use only custom regressors in the model structure by setting `na=nb=nk=0`:

```
m = nlarx(data,[0 0 0],'CustomReg',{'y1(t-1)^2','y1(t-2)*u1(t-3)'})
```

In advanced applications, you can specify advanced estimation options for nonlinearity estimators. For example, `wavenet` and `treepartition` classes provide the `Options` property for setting such estimation options.

Linear and nonlinear regressors.

By default, all model regressors enter as inputs to both linear and nonlinear function blocks of the nonlinearity estimator. To reduce model complexity and keep the estimation well-conditioned, use a subset of regressors as inputs to the nonlinear function of the nonlinear estimator block.

For example, specify a nonlinear ARX model to be linear in past outputs and nonlinear in past inputs:

```
m = nlarx(data,[2 2 1]) % all standard regressors are
                        % inputs to the nonlinear function
```

```
getreg(m) % lists all standard regressors
m = nlarx(data,[4 4 1],sigmoidnet,'nlreg',[5 6 7 8])
```

This example uses `getreg` to determine the index of each regressor from the complete list of all model regressors. Only regressor numbers 5 through 8 are inputs to the nonlinear function—`getreg` shows that these regressors are functions of the input variable `u1`. `nlreg` is an abbreviation for the `NonlinearRegressors` property of the `idnlarx` class.

Alternatively, include only input regressors in the nonlinear function block using:

```
m = nlarx(data,[4 4 1],sigmoidnet,'nlreg','input')
```

When you are not sure which regressors to include as inputs to the nonlinear function block, specify to search during estimation for the optimum regressor combination:

```
m = nlarx(data,[4 4 1],sigmoidnet,'nlreg','search')
```

After estimation, use `m.NonlinearRegressors` to view which regressors were selected by the automatic regressor search. This search typically takes a long time, and you can display the search progress using:

```
m = nlarx(data,[4 4 1],sigmoidnet,'nlreg','search',...
                                  'Display', 'on')
```

Configure the nonlinearity estimator.

Specify the nonlinearity estimator directly in the estimation command as:

- A string of the nonlinearity name, which uses the default nonlinearity configuration.

  ```
  m = nlarx(data, [2 2 1],'sigmoidnet')
  ```

  or

  ```
  m = nlarx(data,[2 2 1],'sig') % abbreviated string
  ```

- Nonlinearity object.

```
m = nlarx(data,[2 2 1],wavenet('num',5))
```

This estimation uses a nonlinear ARX model with a wavelet nonlinearity that has 5 units.

To construct the nonlinearity object before providing it as an input to the nonlinearity estimator:

```
w = wavenet('num', 5);
m = nlarx(data,[2 2 1],w)
```

or

```
w = wavenet;
w.NumberOfUnits = 5;
m = nlarx(data,[2 2 1],w)
```

For MIMO systems, you can specify a different nonlinearity for each output. For example, to specify `sigmoidnet` for the first output and `wavenet` for the second output:

```
M = nlarx(data,[na nb nk],[sigmoidnet; wavenet])
```

If you want the same nonlinearity for all output channels, specify one nonlinearity.

This table summarizes values that specify nonlinearity estimators.

Nonlinearity	Value (Default Nonlinearity Configuration)	Class
Wavelet network (default)	`'wavenet'` or `'wave'`	`wavenet`
One layer sigmoid network	`'sigmoidnet'` or `'sigm'`	`sigmoidnet`
Tree partition	`'treepartition'` or `'tree'`	`treepartition`
F is linear in x	`'linear'` or `[ ]`	`linear`

Additional available nonlinearities include multilayered neural networks and custom networks that you create.

Specify a multilayered neural network using:

```
m = nlarx(data,[na nb nk],NNet)
```

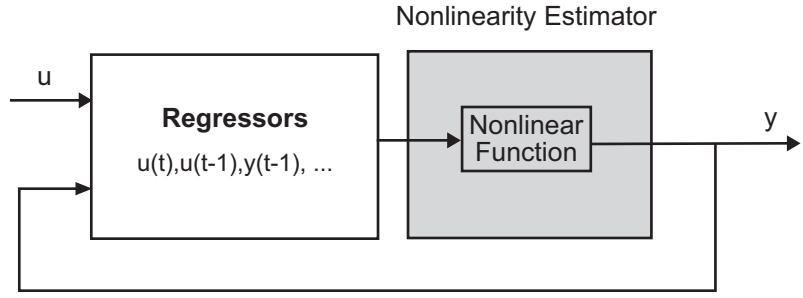
where `NNet` is the neural network object you create using the Neural Network Toolbox software. See the `neuralnet` reference page.

Specify a custom network by defining a function called `gaussunit.m`, as described in the `customnet` reference page. Define the custom network object `CNetw` and estimate the model:

```
CNetw = customnet(@gaussunit);
m = nlarx(data,[na nb nk],CNetw)
```

Include only nonlinear function in nonlinearity estimator.

If your model includes `wavenet`, `sigmoidnet`, and `customnet` nonlinearity estimators, you can exclude the linear function using the `LinearTerm` property of the nonlinearity estimator. The nonlinearity estimator becomes $F(x) = g(Q(x-r))$.



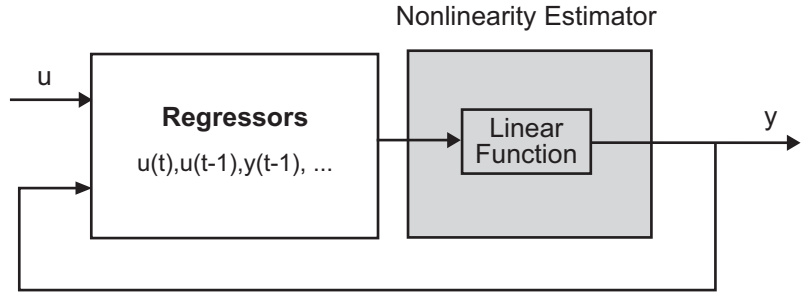
For example:

```
SNL = sigmoidnet('LinearTerm','off'')
m = nlarx(data,[2 2 1],SNL);
```

Note You cannot exclude the linear function from tree partition and neural network nonlinearity estimators.

Include only linear function in nonlinearity estimator.

Configure the nonlinear ARX structure to include only the linear function in the nonlinearity estimator by setting the nonlinearity to `linear`. In this case, $F(x)=L^T(x)+d$ is a weighted sum of model regressors plus an offset. Such models provide a bridge between purely linear ARX models and fully flexible nonlinear models.



In the simplest case, a model with only standard regressors is linear (affine). For example, this structure:

```
m = nlarx(data,[na nb nk],'linear');
```

is similar to the linear ARX model:

```
lin_m = arx(data,[na nb nk]);
```

However, the nonlinear ARX model `m` is more flexible than the linear ARX model `lin_m` because it contains the offset term, d. This offset term provides the additional flexibility of capturing signal offsets, which is not available in linear models.

A popular nonlinear ARX configuration in many applications uses polynomial regressors to model system nonlinearities. In such cases, the system is considered to be a linear combination of products of (delayed) input and output variables. Use the `polyreg` command to easily generate combinations of regressor products and powers.

For example, suppose that you know the output y(t) of a system to be a linear combination of $(y(t-1))^2$ and $y(t-2)*u(t-3)$. To model such as system, use:

```
M = nlarx(data,[0 0 0],'linear',...
                       'CustomReg',{'y1(t-1)^2','y1(t-2)*u1(t-3)'})
```

`M` has no standard regressors and the nonlinearity in the model is described only by the custom regressors.

Iteratively refine the model.

If your model structure includes nonlinearities that support iterative search (see "Estimation Algorithm for Nonlinear ARX Models" on page 4-15), you can use `pem` to refine model parameters:

```
m = nlarx(data,[2 2 1],'sigmoidnet')
m2 = pem(data,m)
```

You can also use `nlarx` to refine the original model:

```
m1 = nlarx(data, [2 2 1],'sigmoidnet','wavenet');
m2 = nlarx(data,m1) % can repeatedly run this command
```

Check the search termination criterion `m.EstimationInfo.WhyStop`. If `WhyStop` indicates that the estimation reached the maximum number of iterations, try repeating the estimation and possibly specifying a larger value for the `MaxIter` `idnlarx` property:

```
m2 = pem(data,m1,'MaxIter',30) % runs 30 more iterations
                               % starting from m1
```

When the `m.EstimationInfo.WhyStop` value is `Near (local) minimum, (norm( g) < tol` or `No improvement along the search direction with line search`, validate your model to see if this model adequately fits the data. If not, the solution might be stuck in a local minimum of the cost-function surface. Try adjusting the `Algorithm.Tolerance` property value of the `idnlarx` class or the `Algorithm.SearchMethod` property, and repeat the estimation. You can also try perturbing the parameters of the last model using `init` (called *randomization*) and refining the model using `pem`:

```
M1 = nlarx(data, [2 2 1], `sigm'); % original model
M1p = init(M1); % randomly perturbs parameters about nominal values
M2 = pem(data, M1p);  % estimates parameters of perturbed model
```

You can display the progress of the iterative search in the MATLAB Command Window using the `Display` property of the `idnlarx` class:

```
M2= pem(data,M1p,'Display','On')
```

What if you cannot get a satisfactory model?

If you do not get a satisfactory model after many trials with various model structures and algorithm settings, it is possible that the data is poor. For example, your data might be missing important input or output variables and does not sufficiently cover all the operating points of the system.

Nonlinear black-box system identification usually requires more data than linear model identification to gain enough information about the system.

Example – Using nlarx to Estimate Nonlinear ARX Models

Use `nlarx` to estimate a nonlinear ARX model for measured input/output data.

1. Prepare the data for estimation:

   ```
   load twotankdata
   z = iddata(y, u, 0.2);
   ze = z(1:1000); zv = z(1001:3000);
   ```

2. Estimate several models using different model orders, delays, and nonlinearity settings:

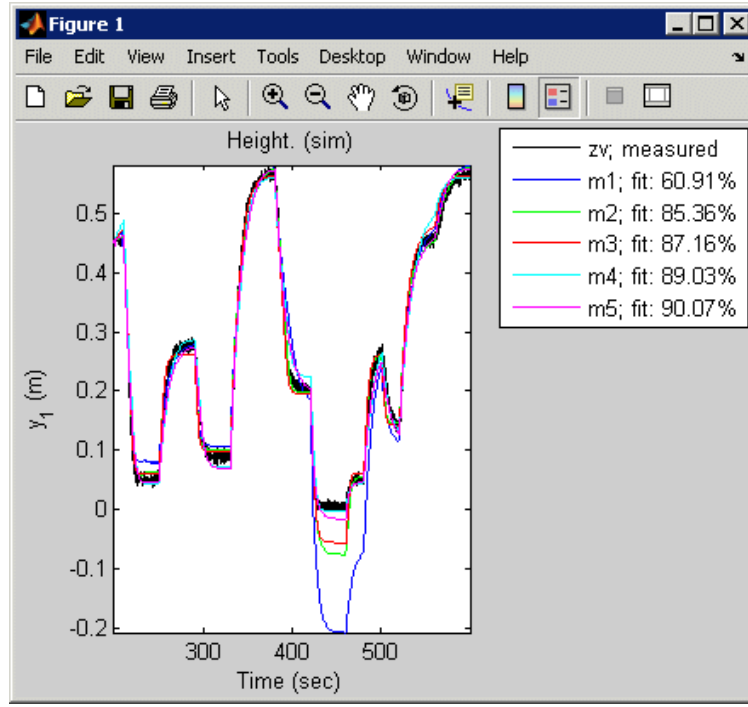
   ```
   m1 = nlarx(ze,[2 2 1]);
   m2 = nlarx(ze,[2 2 3]);
   m3 = nlarx(ze,[2 2 3],wavenet('num',8));
   m4 = nlarx(ze,[2 2 3],wavenet('num',8),...
                         'nlr', [1 2]);
   ```

   An alternative way to perform the estimation is to configure the model structure first, and then to estimate this model:

   ```
   m5 = idnlarx([2 2 3],sigmoidnet('num',14),'nlr',[1 2])
   m5 = pem(ze,m5);
   ```

3. Compare the resulting models by plotting the model outputs with the measured output:

```
compare(zv, m1,m2,m3,m4,m5)
```



The following examples

Using Linear Model for Nonlinear ARX Estimation

About Using Linear Models

You can use an ARX structure polynomial model (`idpoly` with only *A* and *B* as active polynomials) for nonlinear ARX estimation.

Tip To learn more about when to use linear models, see "When to Fit Nonlinear Models" on page 4-2.

Typically, you create a linear ARX model using the `arx` command. You can provide the linear model only at the command line when constructing (see `idnlarx`) or estimating (see `nlarx`) a nonlinear ARX model.

The software uses the linear model for initializing the nonlinear ARX estimation:

- Assigns the linear model orders as initial values of nonlinear model orders (`na` and `nb` properties of the `idnlarx` object) and delays (`nk` property) to compute standard regressors in the nonlinear ARX model structure.
- Uses the *A* and *B* polynomials of the linear model to compute the linear function of the nonlinearity estimators (`LinearCoef` parameter of the nonlinearity estimator object), except for neural network nonlinearity estimator.

During estimation, the estimation algorithm uses these values to further adjust the nonlinear model to the data. The initialization always provides a better fit to the estimation data than the linear ARX model.

How to Initialize Nonlinear ARX Estimation Using Linear ARX Models

Estimate a nonlinear ARX model initialized using a linear model by typing

```
m = nlarx(data,LinARXModel)
```

`LinARXModel` is an `idpoly` object of ARX structure. *`m`* is an `idnlarx` object. *`data`* is a time-domain `iddata` object.

By default, the nonlinearity estimator is the wavelet network (`wavenet` object). This network takes all standard regressors computed using orders and delay of *`LinARXModel`* as inputs to its linear and nonlinear functions. The software computes the `LinearCoef` parameter of the `wavenet` object using the `A` and `B` polynomials of the linear ARX model.

Tip When you use the same data set, a nonlinear ARX model initialized using a linear ARX model produces a better fit than the linear ARX model.

Specify a different nonlinearity estimator, for example a sigmoid network:

```
m = nlarx(data,LinARXModel,'sigmoid')
```

Set the `Focus` property of the `idnlarx` object estimation to simulation error minimization:

```
m = nlarx(data,LinARXModel,'sigmoid','Focus','simulation')
```

After each estimation, validate the model by comparing the simulated response to the data. To improve the fit of the nonlinear ARX model, adjust various elements of the nonlinear ARX structure. For more information, see "Ways to Configure Nonlinear ARX Estimation" on page 4-12.

Estimate Nonlinear ARX Models Using Linear ARX Models

This example shows how to estimate nonlinear ARX models by using linear ARX models.

1. Load the estimation data.

   ```
   load throttledata.mat
   ```

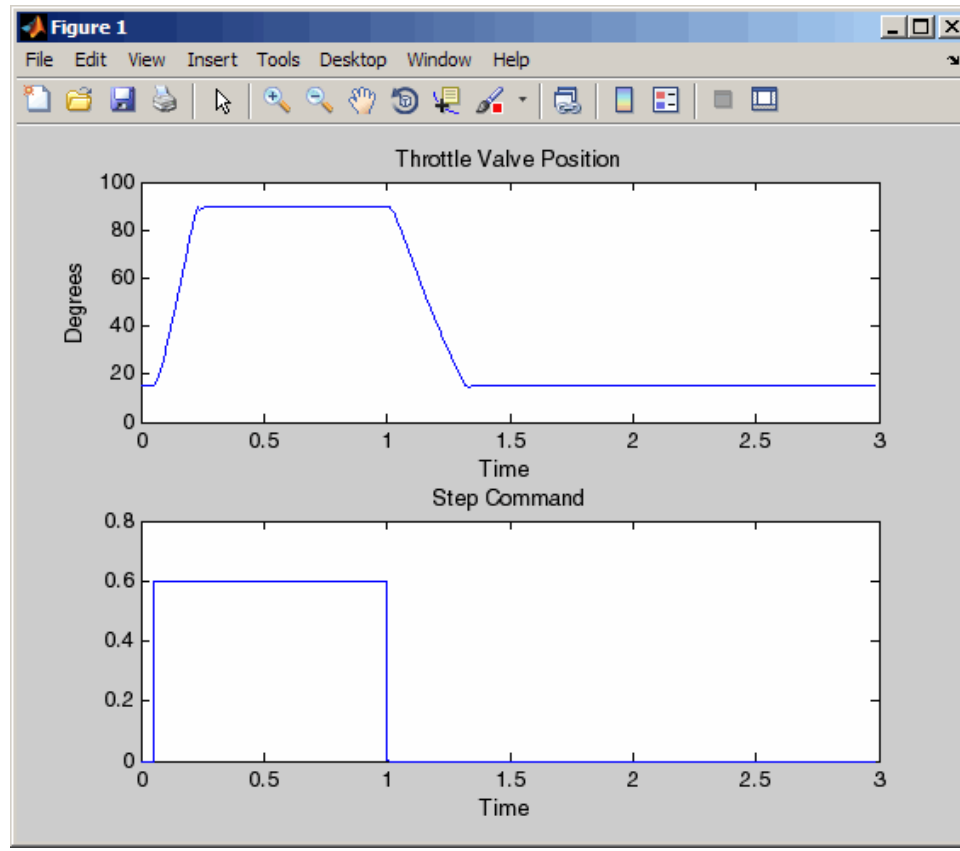
   This command loads the data object `ThrottleData` into the MATLAB workspace. The object contains input and output samples collected from an engine throttle system, sampled at a rate of 100 Hz.

   A DC motor controls the opening angle of the butterfly valve in the throttle system. A step signal (in volts) drives the DC motor. The output is the angular position (in degrees) of the valve.

2. Plot the data to view and analyze the data characteristics.

   ```
   plot(ThrottleData)
   ```

In the normal operating range of 15 90 degrees, the input and output variables have a linear relationship, as shown in the following figure. You use a linear model of low order to model this relationship.



In the throttle system, a hard stop limits the valve position to 90 degrees, and a spring brings the valve to 15 degrees when the DC motor is turned off. These physical components introduce nonlinearities that a linear model cannot capture.

3 Estimate an ARX model to model the linear behavior of this single-input single-output system in the normal operating range.

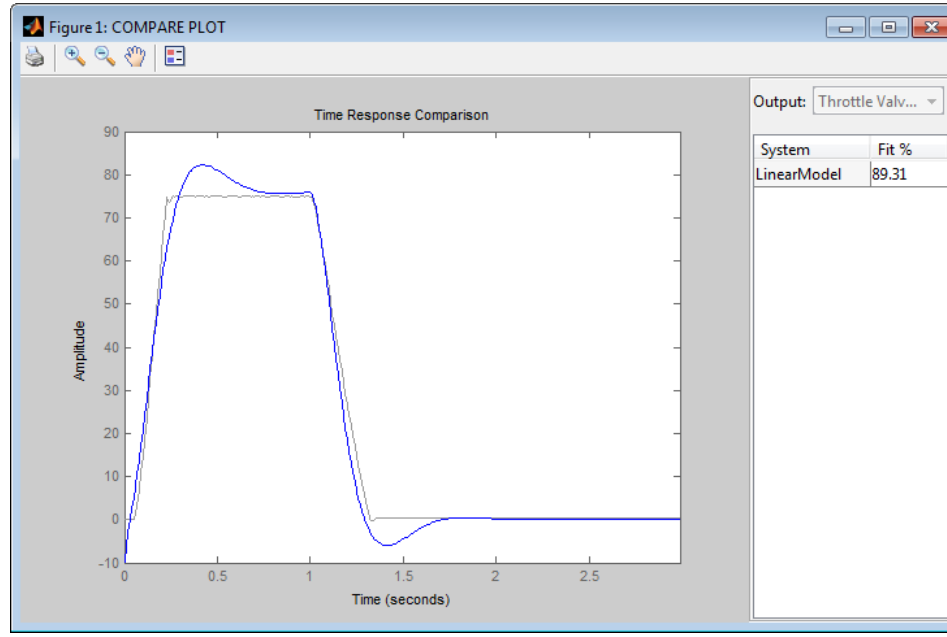
```
% Detrend the data because linear models cannot capture offsets.
```

```
Tr = getTrend(ThrottleData);
Tr.OutputOffset = 15;
DetrendedData = detrend(ThrottleData,Tr);

% Estimate a linear ARX model with na=2, nb=1, nk=1.
opt = arxOptions('Focus','simulation');
LinearModel = arx(DetrendedData,[2 1 1],opt);
```

4 Compare the simulated model response with estimation data.

```
compare(DetrendedData, LinearModel)
```

The linear model captures the rising and settling behavior in the linear operating range but does not account for output saturation at 90 degrees, as shown in the next figure.



5 Estimate a nonlinear ARX model to model the output saturation.

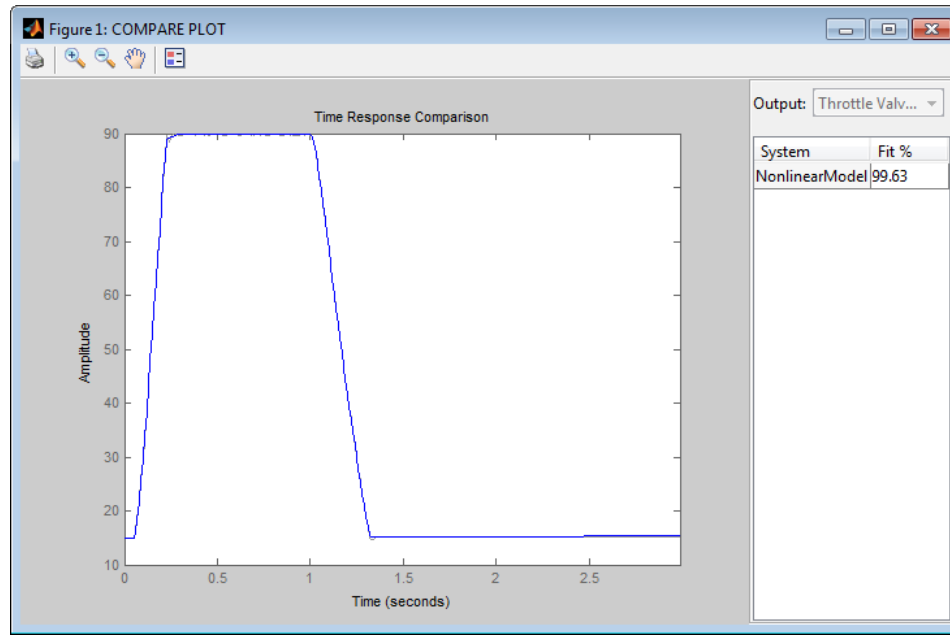
```
NonlinearModel = nlarx(ThrottleData, LinearModel, 'sigmoidnet',...
 'Focus', 'Simulation');
```

The software uses the orders and delay of the linear model for the orders of the nonlinear model. In addition, the software computes the linear function of `sigmoidnet` nonlinearity estimator.

6 Compare the nonlinear model with data.

```
compare(ThrottleData, NonlinearModel)
```

The model captures the nonlinear effects (output saturation) and improves the overall fit to data, as shown in the next figure.



Validating Nonlinear ARX Models

• "Axis Limits, Legend, and 3-D Rotation" on page 4-38

About Nonlinear ARX Plots

The Nonlinear ARX plot displays the characteristics of model nonlinearities as a function of one or two regressors. For more information about estimating nonlinear ARX models, see "Identifying Nonlinear ARX Models" on page 4-8.

Examining a nonlinear ARX plot can help you gain insight into which regressors have the strongest effect on the model output. Understanding the relative importance of the regressors on the output can help you decide which regressors should be included in the nonlinear function.

Furthermore, you can create several nonlinear models for the same data set using different nonlinearity estimators, such a wavelet network and tree partition, and then compare the nonlinear surfaces of these models. Agreement between nonlinear surfaces increases the confidence that these nonlinear models capture the true dynamics of the system.

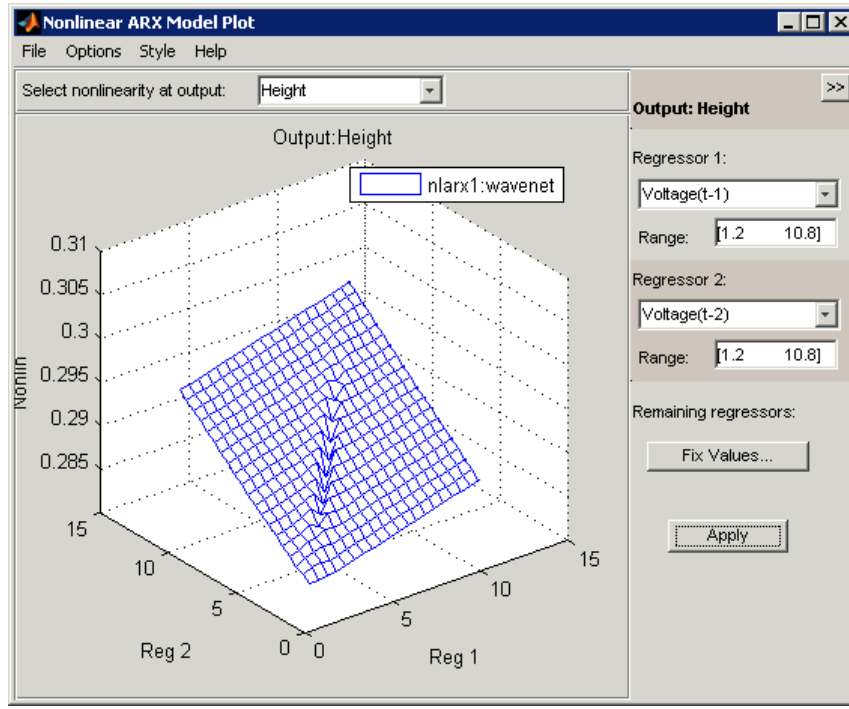
How to Plot Nonlinear ARX Plots Using the GUI

You can plot linear and nonlinear blocks of nonlinear ARX models.

To create a nonlinear ARX plot in the System Identification Tool GUI, select the **Nonlinear ARX** check box in the **Model Views** area. For general information about creating and working with plots, see "Working with Plots" on page 12-13.

Note The **Nonlinear ARX** check box is unavailable if you do not have a nonlinear ARX model in the Model Board.

The following figure shows a sample nonlinear ARX plot.



How to Validate Nonlinear ARX Models at the Command Line

You can use the following approaches to validate nonlinear ARX models at the command line:

Compare Model Output to Measured Output

Compare estimated models using `compare`. Use an independent validation data set whenever possible. For more information about validating models, see “Model Validation”.

For example, compare linear and nonlinear ARX models of same order:

```
load iddata1
LM = arx(z1,[2 2 1]) % estimates linear ARX model
M = nlarx(z1,[2 2 1],'sigmoidnet') % estimates nonlinear ARX model
compare(z1,LM,M) % compares responses of LM and M
```

```
% against measured data
```

Compare the performance of several models using the properties `M.EstimationInfo.FPE` (final prediction error) and `M.EstimationInfo.LossFcn` (value of loss function at estimation termination). Smaller values typically indicate better performance. However, `m.EstimationInfo.FPE` values might be unreliable when the model contains a large number of parameters relative to the estimation data size.

Simulate and Predict Model Response

Use `sim(idnlarx)` and `predict` to simulate and predict model response, respectively. To compute the step response of the model, use `step`. See the corresponding reference page for more information.

Analyze Residuals

Residuals are differences between the one-step-ahead predicted output from the model and the measured output from the validation data set. Thus, residuals represent the portion of the validation data output not explained by the model. Use `resid` to compute and plot the residuals.

Plot Nonlinearity

Use `plot` to view the shape of the nonlinearity. For example:

```
plot(M)
```

where `M` is the nonlinear ARX (`idnlarx`) model. The plot command opens the Nonlinear ARX Model Plot window. For more information about working with this plot window, see “Configuring the Nonlinear ARX Plot” on page 4-37 and “Axis Limits, Legend, and 3-D Rotation” on page 4-38.

If the shape of the plot looks like a plane for all the chosen regressor values, then the model is probably linear in those regressors. In this case, you can remove the corresponding regressors from nonlinear block by specifying the `M.NonlinearRegressors` property and repeat the estimation.

You can use additional `plot` arguments to specify the following information:

- Include multiple nonlinear ARX models on the plot.
- Configure the regressor values for computing the nonlinearity values.

For detailed information about `plot`, type the following command at the prompt:

```
help idnlarx/plot
```

Check Iterative Search Termination Conditions

If your `idnlarx` model structure uses iterative search to minimize prediction or simulation errors, use `M.EstimationInfo` to display the estimation termination conditions. `M` is the estimated `idnlarx` model. For example, check the `WhyStop` field of the `EstimationInfo` property, which describes why the estimation stopped—the algorithm might have reached the maximum number of iterations or the required tolerance value. For more information about iterative search, see "Estimation Algorithm for Nonlinear ARX Models" on page 4-15.

Configuring the Nonlinear ARX Plot

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool GUI. Active models display a thick line inside the Model Board icon.

To configure the plot:

1. If your model contains multiple output, select the output channel in the **Select nonlinearity at output** list. Selecting the output channel displays the nonlinearity values that correspond to this output channel.

2. If the regressor selection options are not visible, click  to expand the Nonlinear ARX Model Plot window.

3. Select **Regressor 1** from the list of available regressors. In the **Range** field, enter the range of values to include on the plot for this regressor. The regressor values are plotted on the **Reg1** axis.

4. Specify a second regressor for a 3-D plot by selecting one of the following types of options:

- Select **Regressor 2** to display three axes. In the **Range** field, enter the range of values to include on the plot for this regressor. The regressor values are plotted on the **Reg2** axis.
- Select <none> in the **Regressor 2** list to display only two axes.

5 To fix the values of the regressor that are not displayed, click **Fix Values**. In the Fix Regressor Values dialog box, double-click the **Value** cell to edit the constant value of the corresponding regressor. The default values are determined during model estimation. Click **OK**.

6 In the Nonlinear ARX Model Plot window, click **Apply** to update the plot.

7 To change the grid of the regressor space along each axis, **Options > Set number of samples**, and enter the number of samples to use for each regressor. Click **Apply** and then **Close**.

For example, if the number of samples is 20, each regressor variable contains 20 points in its specified range. For a 3-D plots, this results in evaluating the nonlinearity at 20 x 20 = 400 points.

Axis Limits, Legend, and 3-D Rotation

The following table summarizes the commands to modify the appearance of the Nonlinear ARX plot.

Changing Appearance of the Nonlinear ARX Plot

Action	Command
Change axis limits.	Select **Options > Set axis limits** to open the Axis Limits dialog box, and edit the limits. Click **Apply**.
Hide or show the legend.	Select **Style > Legend**. Select this option again to show the legend.
(Three axes only) Rotate in three dimensions.	Select **Style > 3D Rotate** and drag the axes on the plot to

Changing Appearance of the Nonlinear ARX Plot (Continued)

Action	Command
Note Available only when you have selected two regressors as independent variables.	a new orientation. To disable three-dimensional rotation, select **Style > 3D Rotate** again.

Using Nonlinear ARX Models

Simulation and Prediction

Use `sim(idnlarx)` to simulate the model output, and `predict` to predict the model output. To compare models to measured output and to each other, use `compare`.

Simulation and prediction commands provide default handling of the model's initial conditions, or initial state values. See the `idnlarx` reference page for a definition of the nonlinear ARX model states.

This toolbox provides several options to facilitate how you specify initial states. For example, you can use `findstates(idnlarx)` and `data2state(idnlarx)` to compute state values based on operating conditions or the requirement to maximize fit to measured output.

To learn more about how `sim` and `predict` compute the model output, see "How the Software Computes Nonlinear ARX Model Output" on page 4-40.

Linearization

Compute linear approximation of nonlinear ARX models using `linearize(idnlarx)` or `linapp`.

`linearize` provides a first-order Taylor series approximation of the system about an operation point (also called *tangent linearization*). `linapp` computes a linear approximation of a nonlinear model for a given input data. For more

information, see the “Linear Approximation of Nonlinear Black-Box Models” on page 4-79.

You can compute the operating point for linearization using `findop(idnlarx)`.

After computing a linear approximation of a nonlinear model, you can perform linear analysis and control design on your model using Control System Toolbox commands. For more information, see “Using Identified Models for Control Design Applications” on page 10-2 and “Using System Identification Toolbox Software with Control System Toolbox Software” on page 10-6.

Simulation and Code Generation Using Simulink

You can import estimated Nonlinear ARX models into the Simulink software using the Nonlinear ARX block (IDNLARX Model) from the System Identification Toolbox block library. Import the `idnlarx` object from the workspace into Simulink using this block to simulate the model output.

The IDNLARX Model block supports code generation with Simulink Coder™ software, using both generic and embedded targets. Code generation does not work when the model contains `customnet` or `neuralnet` nonlinearity estimator, or custom regressors.

How the Software Computes Nonlinear ARX Model Output

In most applications, `sim(idnlarx)` and `predict` are sufficient for computing the simulated and predicted model response, respectively. This advanced topic describes how the software evaluates the output of nonlinearity estimators and uses this output to compute the model response.

Evaluating Nonlinearities

Evaluating the predicted output of a nonlinearity for a specific regressor value x requires that you first extract the nonlinearity F and regressors from the model:

```
F = get(m,'Nonlinearity') % equivalent to F = m.nl
x = getreg(m,'all',data) % computes regressors
```

Evaluate $F(x)$:

```
y = evaluate(F,x)
```

where `x` is a row vector of regressor values.

You can also evaluate predicted output values at multiple time instants by evaluating F for several regressor vectors simultaneously:

```
y = evaluate(F,[x1;x2;x3])
```

Low-Level Simulation and Prediction of Sigmoid Network

This example shows how the software computes the simulated and predicted output of the model as a result of evaluating the output of its nonlinearity estimator for given regressor values.

Estimating and Exploring a Nonlinear ARX Model

1 Estimate nonlinear ARX model with sigmoid network nonlinearity:

```
load twotankdata
estData = iddata(y,u,0.2,'Tstart',0);
M = nlarx(estData,[1 1 0],'sig');
```

2 Inspect the model properties and estimation result:

```
present(M)
```

which provides information about input and output variables, regressors, and nonlinearity estimator:

```
Input name: u1
Output name: y1
Standard regressors corresponding to the orders
  na = 1, nb = 1, nk = 0
No custom regressor
Nonlinear regressors:
  y1(t-1)
  u1(t)
Nonlinearity estimator: sigmoidnet with 10 units
```

3 Inspect the nonlinearity estimator:

```
NL = M.Nonlinearity % equivalent to M.nl
class(NL)   % nonlinearity class
display(NL) % equivalent to NL
```

Inspect the sigmoid network parameter values:

```
NL.Parameters
```

Prediction of Output

The model output is:

$$y1(t)=f(y1(t\text{-}1),u1(t))$$

where f is the sigmoid network function. The model regressors $y1(t\text{-}1)$ and $u1(t)$ are inputs to the nonlinearity estimator. Time t is a discrete variable representing kT, where $k = 0, 1,\ldots$, and T is the sampling interval. In this example, T=`0.2` second.

The output prediction equation is:

$$yp(t)=f(y1\_meas(t\text{-}1),u1\_meas(t))$$

where $yp(t)$ is the predicted value of the response at time t. $y1\_meas(t\text{-}1)$ and $u1\_meas(t)$ are the measured output and input values at times t-1 and t, respectively.

Computing the predicted response includes:

- Computing regressor values from input-output data.
- Evaluating the nonlinearity for given regressor values.

To compute the predicted value of the response using initial conditions and current input:

1 Estimate model from data and get nonlinearity parameters:

```
load twotankdata
```

```
estData = iddata(y,u,0.2,'Tstart',0);
M = nlarx(estData,[1 1 0],'sig');
NL = M.Nonlinearity;
```

2 Specify zero initial states:

```
x0 = 0;
```

The model has one state because there is only one delayed term `y1(t-1)`. The number of states is equal to `sum(getDelayInfo(M))`.

3 Compute the predicted output at time t=0.

```
RegValue = [0,estData.u(1)] % input to nonlinear function f
yp_0 = evaluate(NL,RegValue)
```

`RegValue` is the vector of regressors at `t=0`. The predicted output is $yp(t=0)=f(y1\_meas(t=-1),u1\_meas(t=0))$. In terms of MATLAB variables, this output is f(`0,estData.u(1)`), where

- $y1\_meas(t=-1)$ is the initial state `x0 (=0)`.
- $u1\_meas(t=0)$ is the value of the input at t=0, which is the first input data sample `estData.u(1)`.

4 Compute the predicted output at time t=1.

```
RegValue = [estData.y(1),estData.u(2)];
yp_1 = evaluate(NL,RegValue)
```

The predicted output is $yp(t=1)=f(y1\_meas(t=0),u1\_meas(t=1))$. In terms of MATLAB variables, this output is f(`estData.y(1),estData.u(2)`), where

- $y1\_meas(t=0)$ is the measured output value at t=0, which is to `estData.y(1)`.
- $u1\_meas(t=1)$ is the second input data sample `estData.u(2)`.

5 Perform one-step-ahead prediction at all time values for which data is available.

```
RegMat = getreg(M,[],estData,x0);
yp = evaluate(NL,RegMat)
```

This code obtains a matrix of regressors `RegMat` for all the time samples using `getreg`. `RegMat` has as many rows as there are time samples, and as many columns as there are regressors in the model—two, in this example.

These steps are equivalent to the predicted response computed in a single step using `predict`:

```
yp = predict(M,estData,1,'InitialState',x0)
```

Simulation of Output

The model output is:

$$y1(t)=f(y1(t\text{-}1),u1(t))$$

where f is the sigmoid network function. The model regressors $y1(t\text{-}1)$ and $u1(t)$ are inputs to the nonlinearity estimator. Time t is a discrete variable representing kT, where k = 0, 1,..., and T is the sampling interval. In this example, T=`0.2` second.

The simulated output is:

$$ys(t)=f(ys(t\text{-}1),u1\_meas(t))$$

where $ys(t)$ is the simulated value of the response at time t. The simulation equation is the same as the prediction equation, except that the past output value $ys(t\text{-}1)$ results from the simulation at the previous time step, rather than the measured output value.

Computing the simulated response includes:

- Computing regressor values from input-output data using simulated output values.
- Evaluating the nonlinearity for given regressor values.

To compute the simulated value of the response using initial conditions and current input:

1. Estimate model from data and get nonlinearity parameters:

```
load twotankdata
estData = iddata(y,u,0.2,'Tstart',0);
M = nlarx(estData,[1 1 0],'sig');
NL = M.Nonlinearity
```

2 Specify zero initial states:

```
x0 = 0;
```

The model has one state because there is only one delayed term `y1(t-1)`. The number of states is equal to `sum(getDelayInfo(M))`.

3 Compute the simulated output at time t=0, $ys(t=0)$.

```
RegValue = [0,estData.u(1)]
ys_0 = evaluate(NL,RegValue)
```

`RegValue` is the vector of regressors at t=0. $ys(t=0)=f(y1(t=-1),u1\_meas(t=0))$. In terms of MATLAB variables, this output is f(`0,estData.u(1)`), where

- $y1(t=-1)$ is the initial state `x0 (=0)`.
- $u1\_meas(t=0)$ is the value of the input at t=0, which is the first input data sample `estData.u(1)`.

4 Compute the simulated output at time t=1, $ys(t=1)$.

```
RegValue = [ys_0,estData.u(2)];
ys_1 = evaluate(NL,RegValue)
```

The simulated output $ys(t=1)=f(ys(t=0),u1\_meas(t=1))$. In terms of MATLAB variables, this output is f(`ys_0,estData.u(2)`), where

- $ys(t=0)$ is the simulated value of the output at t=0.
- $u1\_meas(t=1)$ is the second input data sample `estData.u(2)`.

5 Compute the simulated output at time t=2:

```
RegValue = [ys_1,estData.u(3)];
ys_2 = evaluate(NL,RegValue)
```

Unlike for output prediction, you cannot use `getreg` to compute regressor values for all time values. You must compute regressors values at each time

sample separately because the output samples required for forming the regressor vector are available iteratively, one sample at a time.

These steps are equivalent to the simulated response computed in a single step using `sim(idnlarx)`:

```
ys = sim(M,estData,x0)
```

Low-Level Nonlinearity Evaluation

This examples performs a low-level computation of the nonlinearity response for the `sigmoidnet` network function:

$$F(x) = (x-r)PL + a_1 f\left((x-r)Qb_1 + c_1\right) + \ldots + a_n f\left((x-r)Qb_n + c_n\right) + d$$

where f is the sigmoid function, given by the following equation:

$$f(z) = \frac{1}{e^{-z}+1}.$$

In `F(x)`, the input to the sigmoid function is `x-r`. `x` is the regressor value and `r` is regressor mean, computed from the estimation data. a_n, b_n, and c_n are the network parameters stored in the model property `M.nl.par`, where `M` is an `idnlarx` object.

Compute the output value at time `t=1`, when the regressor values are `x=[estData.y(1),estData.u(2)]`:

```
% Estimate model from sample data:
  load twotankdata
  estData = iddata(y,u,0.2,'Tstart',0);
  M = nlarx(estData,[1 1 0],'sig');
  NL = M.Nonlinearity
% Assign values to the parameters in the expression for F(x):
  x = [estData.y(1),estData.u(2)]; % regressor values at t=1
  r = NL.Parameters.RegressorMean;
  P = NL.Parameters.LinearSubspace;
  L = NL.Parameters.LinearCoef;
  d = NL.Parameters.OutputOffset;
```

```
  Q = NL.Parameters.NonLinearSubspace;
  aVec = NL.Parameters.OutputCoef;   %[a_1; a_2; ...]
  cVec = NL.Parameters.Translation;  %[c_1; c_2; ...]
  bMat = NL.Parameters.Dilation;     %[b_1; b_2; ...]
% Compute the linear portion of the response (plus offset):
  yLinear = (x-r)*P*L+d
% Compute the nonlinear portion of the response:
  f = @(z)1/(exp(-z)+1); % anonymous function for sigmoid unit
  yNonlinear = 0;
  for k = 1:length(aVec)
     fInput = (x-r)*Q* bMat(:,k)+cVec(k);
     yNonlinear = yNonlinear+aVec(k)*f(fInput);
  end
% Total response y = F(x) = yLinear + yNonlinear
  y = yLinear + yNonlinear; % y is equal to evaluate(NL,x)
```

Identifying Hammerstein-Wiener Models

In this section...

Applications of Hammerstein-Wiener Models

When the output of a system depends nonlinearly on its inputs, sometimes it is possible to decompose the input-output relationship into two or more interconnected elements. In this case, you can represent the dynamics by a linear transfer function and capture the nonlinearities using nonlinear functions of inputs and outputs of the linear system. The Hammerstein-Wiener model achieves this configuration as a series connection of static nonlinear blocks with a dynamic linear block.

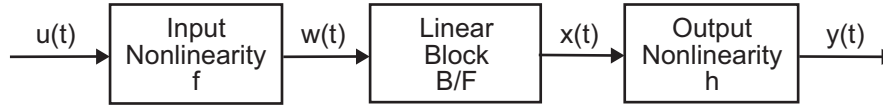
Hammerstein-Wiener model applications span several areas, such as modeling electro-mechanical system and radio frequency components, audio and speech processing and predictive control of chemical processes. These models are popular because they have a convenient block representation, transparent relationship to linear systems, and are easier to implement than heavy-duty nonlinear models (such as neural networks and Volterra models).

You can use the Hammerstein-Wiener model as a black-box model structure because it provides a flexible parameterization for nonlinear models. For example, you might estimate a linear model and try to improve its fidelity by adding an input or output nonlinearity to this model.

You can also use a Hammerstein-Wiener model as a grey-box structure to capture physical knowledge about process characteristics. For example, the input nonlinearity might represent typical physical transformations in actuators and the output nonlinearity might describe common sensor characteristics.

Structure of Hammerstein-Wiener Models

This block diagram represents the structure of a Hammerstein-Wiener model:



where:

- $w(t) = f(u(t))$ is a nonlinear function transforming input data $u(t)$. *w(t)* has the same dimension as *u(t)*.
- $x(t) = (B/F)w(t)$ is a linear transfer function. *x(t)* has the same dimension as *y(t)*.

  where B and F are similar to polynomials in the linear Output-Error model, as described in "What Are Polynomial Models?" on page 3-45.

  For *ny* outputs and *nu* inputs, the linear block is a transfer function matrix containing entries:

  $$\frac{B_{j,i}(q)}{F_{j,i}(q)}$$

  where $j = 1, 2, \ldots, ny$ and $i = 1, 2, \ldots, nu$.
- $y(t) = h(x(t))$ is a nonlinear function that maps the output of the linear block to the system output.

w(t) and *x(t)* are internal variables that define the input and output of the linear block, respectively.

Because *f* acts on the input port of the linear block, this function is called the *input nonlinearity*. Similarly, because *h* acts on the output port of the linear block, this function is called the *output nonlinearity*. If system contains several inputs and outputs, you must define the functions *f* and *h* for each input and output signal.

You do not have to include both the input and the output nonlinearity in the model structure. When a model contains only the input nonlinearity *f*, it is called a *Hammerstein* model. Similarly, when the model contains only the output nonlinearity *h*), it is called a *Wiener* model.

The nonlinearities *f* and *h* are scalar functions, one nonlinear function for each input and output channel.

The Hammerstein-Wiener model computes the output *y* in three stages:

1. Computes $w(t) = f(u(t))$ from the input data.

   $w(t)$ is an input to the linear transfer function B/F.

   The input nonlinearity is a static (*memoryless*) function, where the value of the output a given time *t* depends only on the input value at time *t*.

   You can configure the input nonlinearity as a sigmoid network, wavelet network, saturation, dead zone, piecewise linear function, one-dimensional polynomial, or a custom network. You can also remove the input nonlinearity.

2. Computes the output of the linear block using $w(t)$ and initial conditions: $x(t) = (B/F)w(t)$.

   You can configure the linear block by specifying the numerator *B* and denominator *F* orders.

3. Compute the model output by transforming the output of the linear block $x(t)$ using the nonlinear function *h*: $y(t) = h(x(t))$.

Similar to the input nonlinearity, the output nonlinearity is a static function. Configure the output nonlinearity in the same way as the input nonlinearity. You can also remove the output nonlinearity, such that $y(t) = x(t)$.

Resulting models are `idnlhw` objects that store all model data, including model parameters and nonlinearity estimator. See the `idnlhw` reference page for more information.

Nonlinearity Estimators for Hammerstein-Wiener Models

System Identification Toolbox software provides several scalar nonlinearity estimators $F(x)$ for Hammerstein-Wiener models. The nonlinearity estimators are available for both the input and output nonlinearities f and h, respectively. For more information about $F(x)$, see "Structure of Hammerstein-Wiener Models" on page 4-49.

Each nonlinearity estimator corresponds to an object class in this toolbox. When you estimate Hammerstein-Wiener models in the GUI, System Identification Toolbox creates and configures objects based on these classes. You can also create and configure nonlinearity estimators at the command line. For a detailed description of each estimator, see the references page of the corresponding nonlinearity class.

Nonlinearity	Class	Structure	Comments
Piecewise linear (default)	`pwlinear`	A piecewise linear function parameterized by breakpoint locations.	By default, the number of breakpoints is 10.
One layer sigmoid network	`sigmoidnet`	$g(x) = \sum_{k=1}^{n} \alpha_k \kappa\left(\beta_k (x - \gamma_k)\right)$ $\kappa(s)$ is the sigmoid function $\kappa(s) = \left(e^s + 1\right)^{-1}$. β_k is a row vector such that $\beta_k(x - \gamma_k)$ is a scalar.	Default number of units n is 10.

Nonlinearity	Class	Structure	Comments
Wavelet network	`wavenet`	$g(x) = \sum_{k=1}^{n} \alpha_k \kappa\left(\beta_k (x - \gamma_k)\right)$ where $\kappa(s)$ is the wavelet function.	By default, the estimation algorithm determines the number of units $n$automatically.
Saturation	`saturation`	Parameterize hard limits on the signal value as upper and lower saturation limits.	Use to model known saturation effects on signal amplitudes.
Dead zone	`deadzone`	Parameterize dead zones in signals as the duration of zero response.	Use to model known dead zones in signal amplitudes.
One-dimensional polynomial	`poly1d`	Single-variable polynomial of a degree that you specify.	By default, the polynomial degree is 1.
Unit gain	`unitgain`	Excludes the input or output nonlinearity from the model structure to achieve a Wiener or Hammerstein configuration, respectively. **Note** Excluding both the input and output nonlinearities reduces the Hammerstein-Wiener structure to a linear transfer function.	Useful for configuring multi-input, multi-output (MIMO) models to exclude nonlinearities from specific input and output channels.
Custom network (user-defined)	`customnet`	Similar to sigmoid network but you specify $\kappa(s)$.	(For advanced use) Uses the unit function that you specify.

Ways to Configure Hammerstein-Wiener Estimation

Estimate a Hammerstein-Wiener model with default configuration by:

- Specifying model order and input delay:
  - *nb*—The number of zeros plus one.
  - *nf*—The number of poles.
  - *nk*—The delay from input to the output in terms of the number of samples.

    nb is the order of the transfer function numerator (*B* polynomial), and *nf* is the order of the transfer function denominator (*F* polynomial). As you fit different Hammerstein-Wiener models to your data, you can configure the linear block structure by specifying a different order and delay. For MIMO systems with *ny* outputs and *nu* inputs, *nb*, *nf*, and *nk* are *ny*-by-*nu* matrices.
- Initializing using one of the following discrete-time linear models:
  - An input-output polynomial model of Output-Error (OE) structure (`idpoly`)
  - A linear state-space model with no disturbance component (`idss` object with K=0)

  You can perform this operation only at the command line. The initialization configures the Hammerstein-Wiener model to use orders and delay of the linear model, and the *B* and *F* polynomials as the transfer function numerator and denominator. See "Using Linear Model for Hammerstein-Wiener Estimation" on page 4-63.

By default, the input and output nonlinearity estimators are both piecewise linear functions, parameterized by breakpoint locations (see the `pwlinear` reference page). You can configure the input and output nonlinearity estimators by:

- Configuring the input and output nonlinearity properties.
- Excluding the input or output nonlinear block.

See these topics for detailed steps to change the model structure:

- "How to Estimate Hammerstein-Wiener Models in the GUI" on page 4-54
- "How to Estimate Hammerstein-Wiener Models at the Command Line" on page 4-57

Estimation Algorithm for Hammerstein-Wiener Models

Estimation of Hammerstein-Wiener models uses iterative search to minimize the simulation error between the model output and the measured output.

You can configure the estimation method using the `Algorithm` properties of the `idnlhw` class. The most common of these properties are:

- `MaxIter` — Maximum number of iterations.
- `SearchMethod` — Search method for minimization of prediction or simulation errors, such as Gauss-Newton and Levenberg-Marquardt line search, and Trust-region reflective Newton approach.
- `Tolerance` — Condition for terminating iterative search when the expected improvement of the parameter values is less than a specified value.
- `Display` — Shows progress of iterative minimization in the MATLAB Command Window.

By default, the initial states of the model are zero and not estimated. However, you can choose to estimate initial states during model estimation, which sometimes helps to achieve better results.

How to Estimate Hammerstein-Wiener Models in the GUI

Prerequisites

- Learn about the Hammerstein-Wiener model structure (see "Structure of Hammerstein-Wiener Models" on page 4-49).
- Import data into the System Identification Tool GUI (see "Preparing Data for Nonlinear Identification" on page 4-7).
- (Optional) Choose a nonlinearity estimator in "Nonlinearity Estimators for Hammerstein-Wiener Models" on page 4-51.
- (Optional) Estimate or construct an OE or state-space model to use for initialization. See "Using Linear Model for Hammerstein-Wiener Estimation" on page 4-63.

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

2. In the **Configure** tab, select `Hammerstein-Wiener` from the **Model type** list.

3. (Optional) Edit the **Model name** by clicking the pencil icon. The name of the model should be unique to all Hammerstein-Wiener models in the System Identification Tool GUI.

4. (Optional) If you want to refine a previously estimated model, click **Initialize** to select a previously estimated model from the **Initial Model** list.

   Note Refining a previously estimated model starts with the parameter values of the initial model and uses the same model structure. You can change these settings.

   The **Initial Model** list includes models that:

   - Exist in the System Identification Tool GUI.
   - Have the same number of inputs and outputs as the dimensions of the estimation data (selected as **Working Data** in the System Identification Tool GUI).

5. Keep the default settings in the Nonlinear Models dialog box that specify the model structure and the algorithm, or modify these settings:

   Note For more information about available options, click **Help** in the Nonlinear Models dialog box to open the GUI help.

What to Configure	Options in Nonlinear Models GUI	Comment
Input or output nonlinearity	In the **I/O Nonlinearity** tab, select the **Nonlinearity** and specify the **No. of Units**.	If you do not know which nonlinearity to try, use the (default) piecewise linear nonlinearity. When you estimate from binary input data, you cannot reliably estimate the input nonlinearity. In this case, set **Nonlinearity** for the input channel to `None`. For multiple-input and multiple-output systems, you can assign nonlinearities to specific input and output channels.
Model order and delay	In the **Linear Block** tab, specify **B Order**, **F Order**, and **Input Delay**. For MIMO systems, select the output channel and specify the orders and delays from each input channel.	If you do not know the input delay values, click **Infer Input Delay**. This action opens the Infer Input Delay dialog box which suggests possible delay values.
Estimation algorithm	In the **Estimate** tab, click **Algorithm Options**.	You can specify to estimate initial states.

6 Click **Estimate** to add this model to the System Identification Tool GUI.

The **Estimate** tab displays the estimation progress and results.

7 Validate the model response by selecting the desired plot in the **Model Views** area of the System Identification Tool GUI.

If you get a poor fit, try changing the model structure or algorithm configuration in step 5.

You can export the model to the MATLAB workspace by dragging it to **To Workspace** in the System Identification Tool GUI.

How to Estimate Hammerstein-Wiener Models at the Command Line

Prerequisites

- Learn about the Hammerstein-Wiener model structure described in "Structure of Hammerstein-Wiener Models" on page 4-49.
- Prepare your data, as described in "Preparing Data for Nonlinear Identification" on page 4-7.
- (Optional) Choose a nonlinearity estimator in "Nonlinearity Estimators for Hammerstein-Wiener Models" on page 4-51.
- (Optional) Estimate or construct an input-output polynomial model of Output-Error (OE) structure (`idpoly`) or a state-space model with no disturbance component (`idss` with K=0) for initialization of Hammerstein-Wiener model. See "Using Linear Model for Hammerstein-Wiener Estimation" on page 4-63.

Estimate model using nlhw.

Use `nlhw` to both construct and estimate a Hammerstein-Wiener model. After each estimation, validate the model by comparing it to other models and simulating or predicting the model response.

Basic Estimation

Start with the simplest estimation using *m* = `nlhw`(*data*,[*nb nf nk*]). For example:

```
m = nlhw(data,[2 2 1]) % nb=nf=2 and nk=1
```

The second input argument [*nb nf nk*] sets the order of the linear transfer function, where *nb* is the number of zeros plus 1, *nf* is the number of poles, and *nk* is the input delay. By default, both the input and output nonlinearity estimators are piecewise linear functions (see the `pwlinear` reference page). `m` is an `idnlhw` object.

For MIMO systems, *nb*, *nf*, and *nk* are *ny*-by-*nu* matrices. See the `nlhw` reference page for more information about MIMO estimation.

Configure the nonlinearity estimator.

Specify a different nonlinearity estimator using *m* = `nlhw`(*data*,[*nb nf nk*],*InputNL*,*OutputNL*). *InputNL* and *OutputNL* are nonlinearity estimator objects.

Note If your input signal is binary, set *InputNL* to `unitgain`.

To use nonlinearity estimators with default settings, specify *InputNL* and *OutputNL* using strings (such as `'wave'` for wavelet network or `'sig'` for sigmoid network).

If you need to configure the properties of a nonlinearity estimator, use its object representation. For example, to estimate a Hammerstein-Wiener model that uses saturation as its input nonlinearity and one-dimensional polynomial of degree 3 as its output nonlinearity:

```
m = nlhw(data,[2 2 1],'saturation',poly1d('Degree',3))
```

The third input `'saturation'` is a string representation of the saturation nonlinearity with default property values. `poly1d('Degree',3)` creates a one-dimensional polynomial object of degree 3.

For MIMO models, specify the nonlinearities using objects unless you want to use the same nonlinearity with default configuration for all channels.

This table summarizes values that specify the nonlinearity estimators.

Nonlinearity	Value (Default Nonlinearity Configuration)	Class
Piecewise linear (default)	`'pwlinear'` or `'pwlin'`	`pwlinear`
One layer sigmoid network	`'sigmoidnet'` or `'sigm'`	`sigmoidnet`
Wavelet network	`'wavenet'` or `'wave'`	`wavenet`
Saturation	`'saturation'` or `'sat'`	`saturation`
Dead zone	`'deadzone'` or `'dead'`	`deadzone`
One-dimensional polynomial	`'poly1d'` or `'poly'`	`poly1d`
Unit gain	`'unitgain'` or `[ ]`	`unitgain`

Additional available nonlinearities include custom networks that you create. Specify a custom network by defining a function called `gaussunit.m`, as described in the `customnet` reference page. Define the custom network object `CNetw` as:

```
CNetw = customnet(@gaussunit);
m = nlhw(data,[na nb nk],CNetw)
```

Exclude the input or output nonlinearity.

Exclude a nonlinearity for a specific channel by specifying the `unitgain` value for the `InputNonlinearity` or `OutputNonlinearity` properties.

If the input signal is binary, set *`InputNL`* to `unitgain`.

For more information about model estimation and properties, see the `nlhw` and `idnlhw` reference pages.

For a description of each nonlinearity estimator, see “Nonlinearity Estimators for Hammerstein-Wiener Models” on page 4-51.

Iteratively refine the model.

Use `pem` to refine the original model. For example:

```
m1 = nlhw(data, [2 2 1],'sigmoidnet','wavenet');
m2 = pem(data,m1) % can repeatedly run this command
```

You can also use `nlhw` to refine the original model:

```
m1 = nlhw(data, [2 2 1],'sigmoidnet','wavenet');
m2 = nlhw(data,m1) % can repeatedly run this command
```

Check the search termination criterion in `m.EstimationInfo.WhyStop`. If `WhyStop` indicates that the estimation reached the maximum number of iterations, try repeating the estimation and possibly specifying a larger value for the `MaxIter` `idnlhw` property:

```
m2 = pem(data,m1,'MaxIter',30) % runs 30 more iterations starting at m1
```

When the `m.EstimationInfo.WhyStop` value is `Near (local) minimum, (norm( g) < tol` or `No improvement along the search direction with line search`, validate your model to see if this model adequately fits the data. If not, the solution might be stuck in a local minimum of the cost-function surface. Try adjusting the `Algorithm.Tolerance` property value of the `idnlhw` class or the `Algorithm.SearchMethod` property, and repeat the estimation. You can also try perturbing the parameters of the last model using `init` (called *randomization*) and refining the model using `pem`:

```
M1 = nlhw(data, [2 2 1], `sigm','wave'); % original model
M1p = init(M1); % randomly perturbs parameters about nominal values
M2 = pem(data, M1p);  % estimates parameters of perturbed model
```

You can display the progress of the iterative search in the MATLAB Command Window using the `Display` property of the `idnlhw` class:

```
M2= pem(data,M1p,'Display','On')
```

Improve estimation results using initial states.

If your estimated Hammerstein-Wiener model provides a poor fit to measured data, you can repeat the estimation using the initial state values estimated

from the data. By default, the initial states corresponding to the linear block of the Hammerstein-Wiener model are zero.

To specify estimating initial states during model estimation, use:

```
m = nlhw(data,[nb nf nk],[sigmoidnet;pwlinear],[],...
                         'InitialState','e');
```

What if you cannot get a satisfactory model?

If you do not get a satisfactory model after many trials with various model structures and algorithm settings, it is possible that the data is poor. For example, your data might be missing important input or output variables and does not sufficiently cover all the operating points of the system.

Nonlinear black-box system identification usually requires more data than linear model identification to gain enough information about the system.

Example – Using nlhw to Estimate Hammerstein-Wiener Models

Use `nlhw` to estimate a Hammerstein-Wiener model for measured input/output data.

1 Prepare the data for estimation:

```
load twotankdata
z = iddata(y, u, 0.2);
ze = z(1:1000); zv = z(1001:3000);
```

2 Estimate several models using different model orders, delays, and nonlinearity settings:

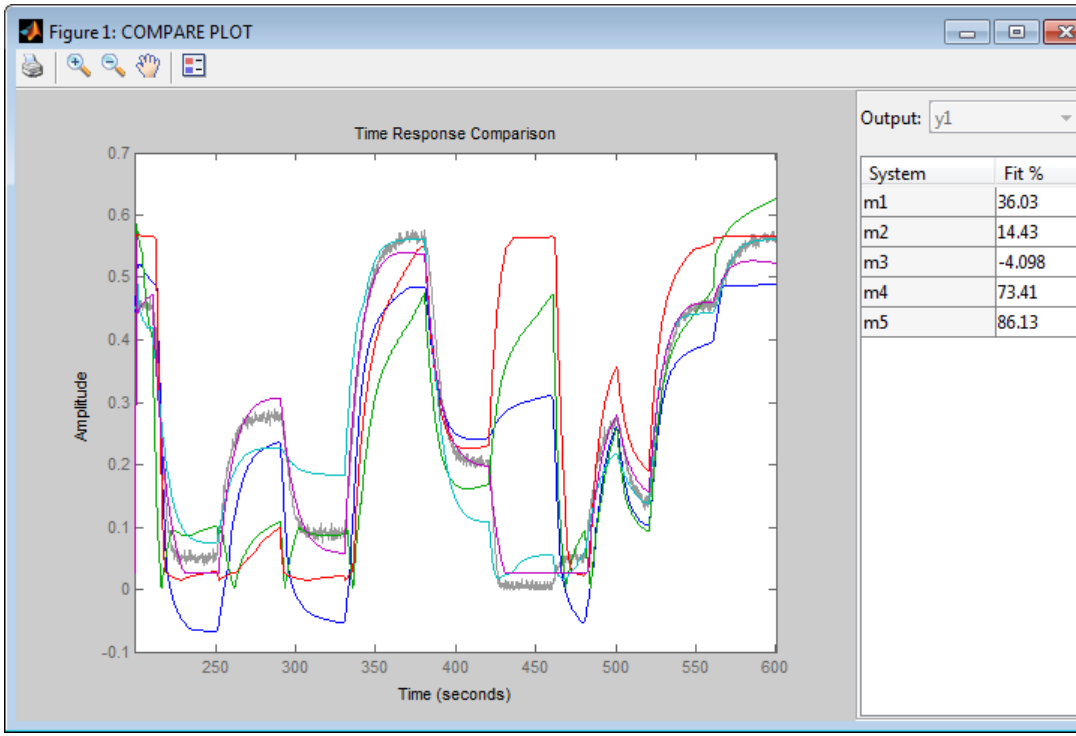
```
m1 = nlhw(ze,[2 3 1]);
m2 = nlhw(ze,[2 2 3]);
m3 = nlhw(ze,[2 2 3], pwlinear('num',13),...
                      pwlinear('num',10));
m4 = nlhw(ze,[2 2 3], sigmoidnet('num',2),...
                      pwlinear('num',10));
```

An alternative way to perform the estimation is to configure the model structure first, and then to estimate this model:

```
m5 = idnlhw([2 2 3], 'dead','sat')
m5 = pem(ze,m5);
```

3 Compare the resulting models by plotting the model outputs on top of the measured output:

```
compare(zv,m1,m2,m3,m4,m5)
```



Example – Improving a Linear Model Using Hammerstein-Wiener Structure

Use the Hammerstein-Wiener model structure to improve a previously estimated linear model. After estimating the linear model, insert it into the Hammerstein-Wiener structure that includes input or output nonlinearities.

1 Estimate a linear model:

```
load iddata1
LM = arx(z1,[2 2 1]);
```

2 Extract the transfer function coefficients from the linear model:

```
[Num, Den] = tfdata(LM);
```

3 Create a Hammerstein-Wiener model, where you initialize the linear block properties `B` and `F` using `Num` and `Den`, respectively:

```
nb = 1;       % In general, nb = ones(ny,nu)
              % ny is number of outputs
              % nu is number of inputs
nf = nb;
nk = 0;       % In general, nk = zeros(ny,nu)
              % ny is number of outputs
              % nu is number of inputs
M = idnlhw([nb nf nk],'poly1d','pwlinear');
M.b = Num;
M.f = Den;
```

4 Estimate the model coefficients, which refines the linear model coefficients in `Num` and `Den`:

```
M = pem(z1, M);
```

5 Compare responses of linear and nonlinear model against measured data:

```
compare(z1,LM,M)
```

Using Linear Model for Hammerstein-Wiener Estimation

- "About Using Linear Models" on page 4-64
- "How to Initialize Hammerstein-Wiener Estimation Using Linear Polynomial Output-Error or State-Space Models" on page 4-64
- "Estimate Hammerstein-Wiener Models Using Linear OE Models" on page 4-65

About Using Linear Models

You can use a polynomial model of Output-Error (OE) structure (`idpoly`) or state-space model with no disturbance component (`idss` model with K = 0) for Hammerstein-Wiener estimation. The linear model must sufficiently represent the linear dynamics of your system.

Tip To learn more about when to use linear models, see "When to Fit Nonlinear Models" on page 4-2.

Typically, you use the `oe` or `n4sid` command to obtain the linear model. You can provide the linear model only at the command line when constructing (see `idnlhw`) or estimating (see `nlhw`) a Hammerstein-Wiener model.

The software uses the linear model for initializing the Hammerstein-Wiener estimation:

- Assigns the linear model orders as initial values of nonlinear model orders (`nb` and `nf` properties of the Hammerstein-Wiener (`idnlhw`) and delays (`nk` property).
- Sets the B and F polynomials of the linear transfer function in the Hammerstein-Wiener model structure.

During estimation, the estimation algorithm uses these values to further adjust the nonlinear model to the data.

How to Initialize Hammerstein-Wiener Estimation Using Linear Polynomial Output-Error or State-Space Models

Estimate a Hammerstein-Wiener model using either a linear input-output polynomial model of OE structure or state-space model by typing

```
m = nlhw(data,LinModel)
```

`LinModel` must be an `idpoly` model of OE structure, a state-space model (`idss` with `K` = 0) or transfer function `idtf` model. *`m`* is an `idnlhw` object. *`data`* is a time-domain `iddata` object.

By default, both the input and output nonlinearity estimators are piecewise linear functions (see `pwlinear`).

Specify different input and output nonlinearity, for example `sigmoid` and `deadzone`:

```
m = nlarx(data,LinModel, 'sigmoid','deadzone')
```

After each estimation, validate the model by comparing the simulated response to the data. To improve the fit of the Hammerstein-Wiener model, adjust various elements of the Hammerstein-Wiener structure. For more information, see "Ways to Configure Hammerstein-Wiener Estimation" on page 4-52.

Estimate Hammerstein-Wiener Models Using Linear OE Models

This example shows how to estimate Hammerstein-Wiener models using linear OE models.

1 Load the estimation data.

```
load throttledata.mat
```

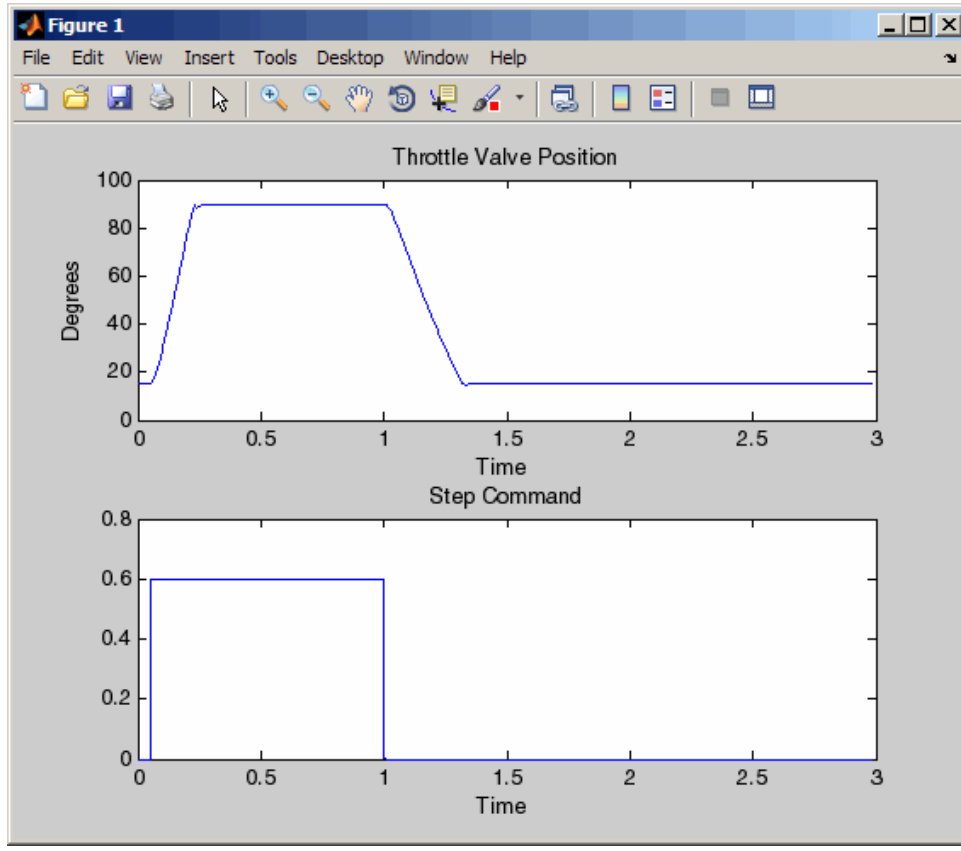
This command loads the data object `ThrottleData` into the MATLAB workspace. The object contains input and output samples collected from an engine throttle system, sampled at a rate of 100 Hz.

A DC motor controls the opening angle of the butterfly valve in the throttle system. A step signal (in volts) drives the DC motor. The output is the angular position (in degrees) of the valve.

2 Plot the data to view and analyze the data characteristics.

```
plot(ThrottleData)
```

In the normal operating range of `15 90` degrees, the input and output variables have a linear relationship, as shown in the following figure. You use a linear model of low order to model this relationship.



In the throttle system, a hard stop limits the valve position to 90 degrees, and a spring brings the valve to 15 degrees when the DC motor is turned off. These physical components introduce nonlinearities that a linear model cannot capture.

3 Estimate a Hammerstein-Wiener model to model the linear behavior of this single-input single-output system in the normal operating range.

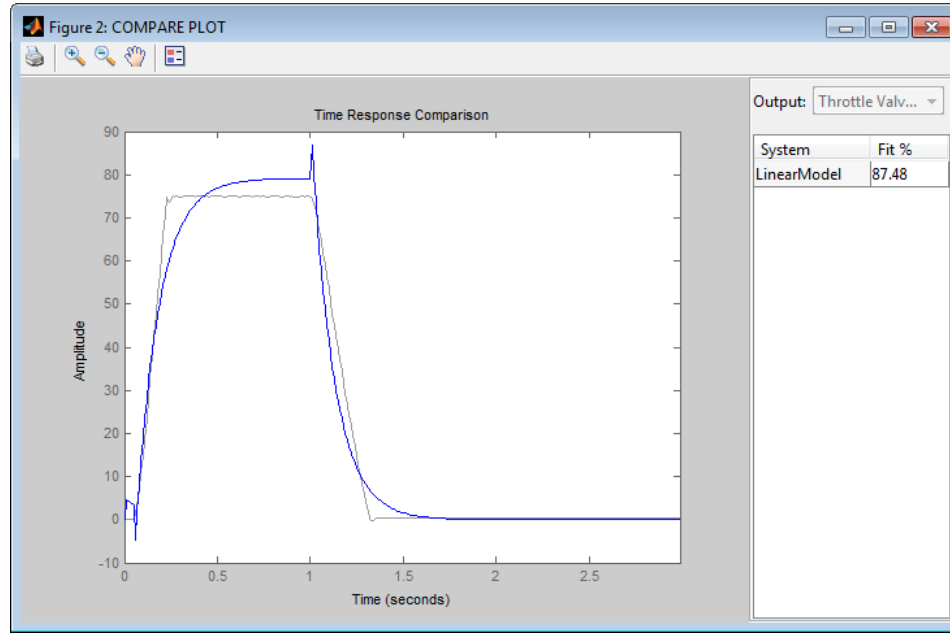
```
% Detrend the data because linear models cannot capture offsets.
Tr = getTrend(ThrottleData);
Tr.OutputOffset = 15;
DetrendedData = detrend(ThrottleData,Tr);
```

```
% Estimate a linear OE model with na=2, nb=1, nk=1.
opt = oeOptions('Focus','simulation');
LinearModel = oe(DetrendedData,[2 1 1],opt);
```

4 Compare the simulated model response with estimation data.

```
compare(DetrendedData, LinearModel)
```

The linear model captures the rising and settling behavior in the linear operating range but does not account for output saturation at 90 degrees, as shown in the next figure.



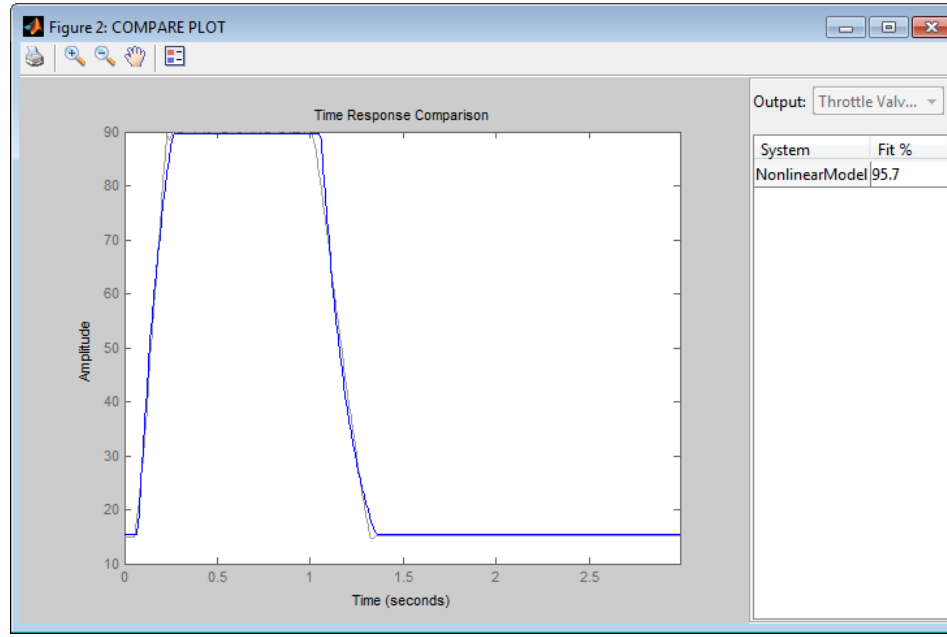
5 Estimate a Hammerstein-Wiener model to model the output saturation.

```
NonlinearModel = nlhw(ThrottleData, LinearModel, [], 'saturation')
```

The software uses the orders and delay of the linear model for the orders of the nonlinear model. In addition, the software uses the B and F polynomials of the linear transfer function.

6 Compare the nonlinear model with data.

```
compare(ThrottleData, NonlinearModel)
```



Validating Hammerstein-Wiener Models

About Hammerstein-Wiener Plots

Hammerstein-Wiener model plot lets you explore the characteristics of the linear block and the static nonlinearities of the Hammerstein-Wiener model.

For more information about estimating nonlinear Hammerstein-Wiener models, see "Identifying Hammerstein-Wiener Models" on page 4-48.

Examining a Hammerstein-Wiener plot can help you determine whether you chose an unnecessarily complicated nonlinearity for modeling your system. For example, if you chose a piece-wise-linear nonlinearity (which is very general), but the plot indicates saturation behavior, then you can estimate a new model using the simpler saturation nonlinearity instead.

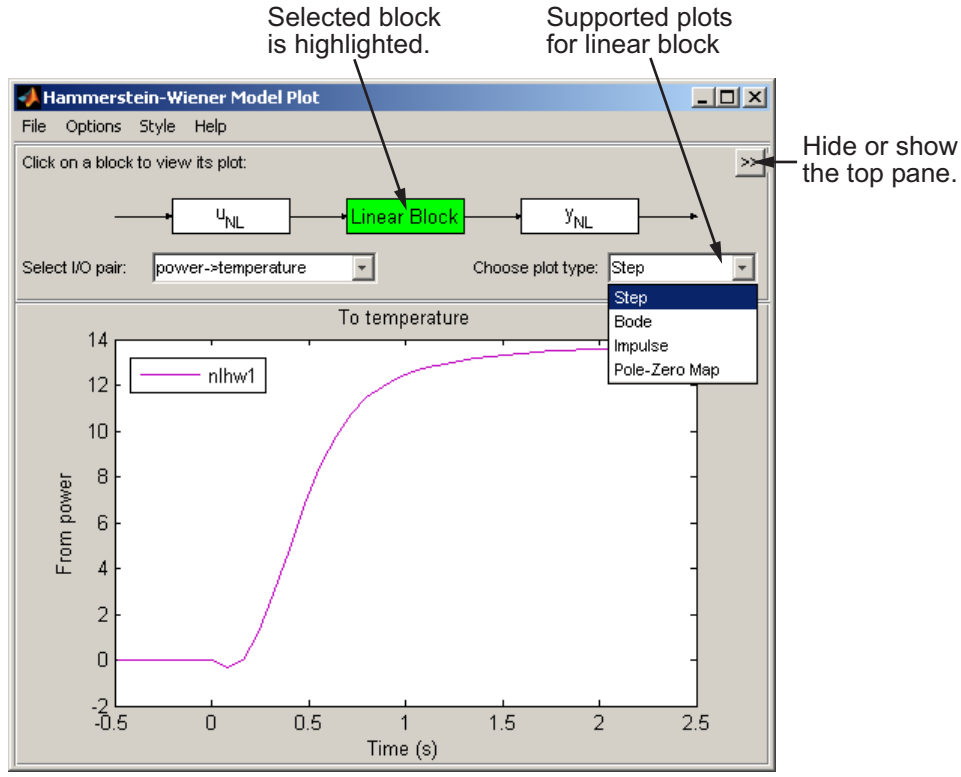
For multivariable systems, you can use the Hammerstein-Wiener plot to determine whether to exclude nonlinearities for specific channels. If the nonlinearity for a specific input or output channel does not exhibit strong nonlinear behavior, you can estimate a new model after setting the nonlinearity at that channel to unit gain.

How to Create Hammerstein-Wiener Plots in the GUI

To create a Hammerstein-Wiener plot in the System Identification Tool GUI, select the **Hamm-Wiener** check box in the **Model Views** area. For general information about creating and working with plots, see "Working with Plots" on page 12-13.

Note The **Hamm-Wiener** check box is unavailable if you do not have a Hammerstein-Wiener model in the Model Board.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool GUI. By default, the input nonlinearity block U_{NL} is selected. You can select the output nonlinearity block Y_{NL} or `Linear Block`, as shown in the next figure.



After you generate a plot, you can learn more about your model by:

- “Plotting Nonlinear Block Characteristics” on page 4-72
- “Plotting Linear Block Characteristics” on page 4-73

How to Validate Hammerstein-Wiener Models at the Command Line

You can use the following approaches to validate Hammerstein-Wiener models at the command line:

Compare Model Output to Measured Output

Compare estimated models using `compare`. Use an independent validation data set whenever possible. For more information about validating models, see "Model Validation".

For example, compare linear and nonlinear ARX models of same order:

```
load iddata1
LM = arx(z1,[2 2 1]) % estimates linear ARX model
M = nlhw(z1,[2 2 1]) % estimates Hammerstein-Wiener model
compare(z1,LM,M) % compares responses of LM and M
                 % against measured data
```

Compare the performance of several models using the properties `M.EstimationInfo.FPE` (final prediction error) and `M.EstimationInfo.LossFcn` (value of loss function at estimation termination). Smaller values typically indicate better performance. However, `m.EstimationInfo.FPE` values might be unreliable when the model contains a large number of parameters relative to the estimation data size. Use these indicators in combination with other validation techniques to draw reliable conclusions.

Simulate and Predict Model Response

Use `sim(idnlhw)` and `predict` to simulate and predict model response, respectively. To compute the step response of the model, use `step`. See the corresponding reference page for more information.

Analyze Residuals

Residuals are differences between the model output and the measured output. Thus, residuals represent the portion of the output not explained by the model. Use `resid` to compute and plot the residuals.

Plot Nonlinearity

Access the object representing the nonlinearity estimator and its parameters using `M.InputNonlinearity` (or `M.unl`) and `M.OutputNonlinearity` (or `M.ynl`), where `M` is the estimated model.

Use `plot` to view the shape of the nonlinearity and properties of the linear block. For example:

```
plot(M)
```

You can use additional `plot` arguments to specify the following information:

- Include several Hammerstein-Wiener models on the plot.
- Configure how to evaluate the nonlinearity at each input and output channel.
- Specify the time or frequency values for computing transient and frequency response plots of the linear block.

The plot command opens the Hammerstein-Wiener Model Plot window. For more information about working with this plot window, see "Plotting Nonlinear Block Characteristics" on page 4-72 and "Plotting Linear Block Characteristics" on page 4-73.

For detailed information about `plot`, type the following command at the prompt:

```
help idnlhw/plot
```

Check Iterative Search Termination Conditions

Use `M.EstimationInfo` to display the estimation termination conditions, where `M` is the estimated `idnlhw` model. For example, check the `WhyStop` field of the `EstimationInfo` property, which describes why the estimation was stopped. For example, the algorithm might have reached the maximum number of iterations or the required tolerance value.

Plotting Nonlinear Block Characteristics

The Hammerstein-Wiener model can contain up to two nonlinear blocks. The nonlinearity at the input to the Linear Block is labeled u_{NL} and is called the *input nonlinearity*. The nonlinearity at the output of the Linear Block is labeled y_{NL} and is called the *output nonlinearity*.

To configure the plot, perform the following steps:

1. If the top pane is not visible, click  to expand the Hammerstein-Wiener Model Plot window.

2. Select the nonlinear block you want to plot:

   - To plot u_{NL} as a command of the input data, click the u_{NL} block.
   - To plot y_{NL} as a command of its inputs, click the y_{NL} block.

   The selected block is highlighted in green.

Note The input to the output nonlinearity block y_{NL} is the output from the Linear Block and not the measured input data.

---

3. If your model contains multiple inputs or outputs, select the channel in the **Select nonlinearity at channel** list. Selecting the channel updates the plot and displays the nonlinearity values versus the corresponding input to this nonlinear block.

4. To change the range of the horizontal axis, select **Options > Set input range** to open the Range for Input to Nonlinearity dialog box. Enter the range using the format `[MinValue MaxValue]`. Click **Apply** and then **Close** to update the plot.

Plotting Linear Block Characteristics

The Hammerstein-Wiener model contains one Linear Block that represents the embedded linear model.

To configure the plot:

1. If the top pane is not visible, click  to expand the Hammerstein-Wiener Model Plot window.

2. Click the Linear Block to select it. The Linear Block is highlighted in green.

3. In the **Select I/O pair** list, select the input and output data pair for which to view the response.

4 In the **Choose plot type** list, select the linear plot from the following options:

- `Step`
- `Impulse`
- `Bode`
- `Pole-Zero Map`

5 If you selected to plot step or impulse response, you can set the time span. Select **Options > Time span** and enter a new time span in units of time you specified for the model.

For a time span *T*, the resulting response is plotted from *-T/4* to *T*. The default time span is 10.

Click **Apply** and then **Close**.

6 If you selected to plot a Bode plot, you can set the frequency range.

The default frequency vector is 128 linearly distributed values, greater than zero and less than or equal to the Nyquist frequency. To change the range, select **Options > Frequency range**, and specify a new frequency vector in units of rad per model time units.

Enter the frequency vector using any one of following methods:

- MATLAB expression, such as `[1:100]*pi/100` or `logspace(-3,-1,200)`. Cannot contain variables in the MATLAB workspace.
- Row vector of values, such as `[1:.1:100]`.

Click **Apply** and then **Close**.

Using Hammerstein-Wiener Models

Simulation and Prediction

Use `sim(idnlhw)` to simulate the model output, and `predict` to predict the model output. To compare models to measured output and to each other, use `compare`.

This toolbox provides a number of options to facilitate how you specify initial states. For example, you can use `findstates(idnlhw)` to automatically search for state values in simulation and prediction applications. You can also specify the states manually.

If you need to specify the states manually, see the `idnlhw` reference page for a definition of the Hammerstein-Wiener model states.

To learn more about how `sim` and `predict` compute the model output, see "How the Software Computes Hammerstein-Wiener Model Output" on page 4-76.

Linearization

Compute linear approximation of Hammerstein-Wiener models using `linearize(idnlhw)` or `linapp`.

`linearize` provides a first-order Taylor series approximation of the system about an operation point (also called *tangent linearization*). `linapp` computes a linear approximation of a nonlinear model for a given input data. For more information, see the "Linear Approximation of Nonlinear Black-Box Models" on page 4-79.

You can compute the operating point for linearization using `findop(idnlhw)`.

After computing a linear approximation of a nonlinear model, you can perform linear analysis and control design on your model using Control System Toolbox commands. For more information, see "Using Identified Models for Control Design Applications" on page 10-2 and "Using System Identification Toolbox Software with Control System Toolbox Software" on page 10-6.

Simulation and Code Generation Using Simulink

You can import the estimated Hammerstein-Wiener Model into the Simulink software using the Hammerstein-Wiener block (IDNLHW Model) from the System Identification Toolbox block library. After you bring the `idnlhw` object from the workspace into Simulink, you can simulate the model output.

The IDNLHW Model block supports code generation with the Simulink Coder software, using both generic and embedded targets. Code generation does not work when the model contains `customnet` as the input or output nonlinearity.

How the Software Computes Hammerstein-Wiener Model Output

In most applications, `sim(idnlhw)` and `predict` are sufficient for computing the simulated and predicted model response, respectively. This advanced topic describes how the software evaluates the output of nonlinearity estimators and uses this output to compute the model response.

Evaluating Nonlinearities (SISO)

Evaluating the predicted output of a nonlinearity for a input u requires that you first extract the input or output nonlinearity F from the model:

```
F = M.InputNonlinearity  % equivalent to F = M.unl
H = M.OutputNonlinearity % equivalent to F = M.ynl
```

Evaluate $F(u)$:

```
w = evaluate(F,u)
```

where `u` is a scalar representing the value of the input signal at a given time.

You can evaluate predicted output values at multiple time instants by evaluating F for several time values simultaneously using a column vector of input values:

```
w = evaluate(F,[u1;u2;u3])
```

Similarly, you can evaluate the value of the nonlinearity H using the output of the linear block $x(t)$ as its input:

```
y = evaluate(H,x)
```

Evaluating Nonlinearities (MIMO)

For MIMO models, F and H are vectors of length nu and ny, respectively. nu is the number of inputs and ny is the number of outputs. In this case, you must evaluate the predicted output of each nonlinearity separately.

For example, suppose that you estimate a two-input model:

```
M = nlhw(data,[nb nf nk],[wavenet;poly1d],'saturation')
```

In the input nonlinearity:

```
F = M.InputNonlinearity
F1 = F(1);
F2 = F(2);
```

`F` is a vector function containing two elements: `F=[F1(u1_value); F2(u2_value)]`, where `F1` is a `wavenet` object and `F2` is a `poly1d` object. `u1_value` is the first input signal and `u2_value` is the second input signal.

Evaluate `F` by evaluating `F1` and `F2` separately:

```
w1 = evaluate(F(1), u1_value);
w2 = evaluate(F(2), u2_value);
```

The total input to the linear block, `w`, is a vector of `w1` and `w2` (`w = [w1 w2]`).

Similarly, you can evaluate the value of the nonlinearity *H*:

```
H = M.OutputNonlinearity %equivalent to H = M.ynl
```

Low-level Simulation of Hammerstein-Wiener Model

This example shows how the software evaluates the simulated output by first computing the output of the input and output nonlinearity estimators. For same initial conditions, the prediction results match the simulation results.

1. Estimate Hammerstein-Wiener model:

   ```
   load twotankdata
   estData = iddata(y,u,0.2)
   M = nlhw(estData,[1 5 3],'pwlinear','poly1d');
   ```

2. Extract the input nonlinearity, linear model, and output nonlinearity as separate variables:

   ```
   uNL = M.InputNonlinearity;
   linModel = M.LinearModel;
   ```

```
yNL = M.OutputNonlinearity;
```

3 Simulate the output of the input nonlinearity estimator:

```
u = estData.u; %input data for simulation
% Compute output of input nonlinearity:
w = evaluate(uNL, u);
% Response of linear model to input w and zero
% initial conditions:
x = sim(linModel, w);
% Compute the output of the Hammerstein-Wiener model M
% as the output of the output nonlinearity estimator to input x:
y = evaluate(yNL, x);
% Previous commands are equivalent to:
ysim = sim(M, u);
% Compare low-level and direct simulation results:
time = estData.SamplingInstants;
plot(time, y, time, ysim, '.')
```

Linear Approximation of Nonlinear Black-Box Models

In this section...

Why Compute a Linear Approximation of a Nonlinear Model?

Control design and linear analysis techniques using Control System Toolbox software require linear models. You can use your estimated nonlinear model in these applications after you linear the model. After you linearize your model, you can use the model for control design and linear analysis.

Choosing Your Linear Approximation Approach

System Identification Toolbox software provides two approaches for computing a linear approximation of nonlinear ARX and Hammerstein-Wiener models.

To compute a linear approximation of a nonlinear model for a given input signal, use the `linapp` command. The resulting model is only valid for the same input that you use to compute the linear approximation. For more information, see "Linear Approximation of Nonlinear Black-Box Models for a Given Input" on page 4-80.

If you want a tangent approximation of the nonlinear dynamics that is accurate near the system operating point, use the `linearize` command. The resulting model is a first-order Taylor series approximation for the system about the operating point, which is defined by a constant input and model state values. For more information, see "Tangent Linearization of Nonlinear Black-Box Models" on page 4-80.

Linear Approximation of Nonlinear Black-Box Models for a Given Input

`linapp` computes the best linear approximation, in a mean-square-error sense, of a nonlinear ARX or Hammerstein-Wiener model for a given input or a randomly generated input. The resulting linear model might only be valid for the same input signal as you the one you used to generate the linear approximation.

`linapp` estimates the best linear model that is structurally similar to the original nonlinear model and provides the best fit between a given input and the corresponding simulated response of the nonlinear model.

To compute a linear approximation of a nonlinear black-box model for a given input, you must have these variables:

- Nonlinear ARX model (`idnlarx` object) or Hammerstein-Wiener model (`idnlhw` object)
- Input signal for which you want to obtain a linear approximation, specified as a real matrix or an `iddata` object

`linapp` uses the specified input signal to compute a linear approximation:

- For nonlinear ARX models, `linapp` estimates a linear ARX model using the same model orders `na`, `nb`, and `nk` as the original model.
- For Hammerstein-Wiener models, `linapp` estimates a linear Output-Error (OE) model using the same model orders `nb`, `nf`, and `nk`.

To compute a linear approximation of a nonlinear black-box model for a randomly generated input, you must specify the minimum and maximum input values for generating white-noise input with a magnitude in this rectangular range, `umin` and `umax`.

For more information, see the `linapp` reference page.

Tangent Linearization of Nonlinear Black-Box Models

`linearize` computes a first-order Taylor series approximation for nonlinear system dynamics about an *operating point*, which is defined by a constant

input and model state values. The resulting linear model is accurate in the local neighborhood of this operating point.

To compute a tangent linear approximation of a nonlinear black-box model, you must have these variables:

- Nonlinear ARX model (`idnlarx` object) or Hammerstein-Wiener model (`idnlhw` object)
- Operating point

To specify the operating point of your system, you must specify the constant input and the states. For more information about state definitions for each type of parametric model, see these reference pages:

- `idnlarx` — Nonlinear ARX model
- `idnlhw` — Nonlinear Hammerstein-Wiener model

If you do not know the operating point values for your system, see "Computing Operating Points for Nonlinear Black-Box Models" on page 4-81.

For more information, see the `linearize(idnlarx)` or `linearize(idnlhw)` reference page.

Computing Operating Points for Nonlinear Black-Box Models

An *operating point* is defined by a constant input and model state values.

If you do not know the operating conditions of your system for linearization, you can use `findop` to compute the operating point from specifications:

- "Computing Operating Point from Steady-State Specifications" on page 4-81
- "Computing Operating Points at a Simulation Snapshot" on page 4-82

Computing Operating Point from Steady-State Specifications

Use `findop` to compute an operating point from steady-state specifications:

- Values of input and output signals.
  If either the steady-state input or output value is unknown, you can specify it as `NaN` to estimate this value. This is especially useful when modeling MIMO systems, where only a subset of the input and output steady-state values are known.
- More complex steady-state specifications.

  Construct an object that stores specifications for computing the operating point, including input and output bounds, known values, and initial guesses. For more information, see `operspec(idnlarx)` or `operspec(idnlhw)`.

For more information, see the `findop(idnlarx)` or `findop(idnlhw)` reference page.

Computing Operating Points at a Simulation Snapshot

Compute an operating point at a specific time during model simulation (snapshot) by specifying the snapshot time and the input value. To use this method for computing the equilibrium operating point, choose an input that leads to a steady-state output value. Use that input and the time value at which the output reaches steady state (*snapshot* time) to compute the operating point.

It is optional to specify the initial conditions for simulation when using this method because initial conditions often do not affect the steady-state values. By default, the initial conditions are zero.

However, for nonlinear ARX models, the steady-state output value might depend on initial conditions. For these models, you should investigate the effect of initial conditions on model response and use the values that produce the desired output. You can use `data2state(idnlarx)` to map the input-output signal values from before the simulation starts to the model's initial states. Because the initial states are a function of the past history of the model's input and output values, `data2state` generates the initial states by transforming the data.

5

ODE Parameter Estimation (Grey-Box Modeling)

Supported Grey-Box Models

If you understand the physics of your system and can represent the system using ordinary differential or difference equations (ODEs) with unknown parameters, then you can use System Identification Toolbox commands to perform linear or nonlinear grey-box modeling. *Grey-box model* ODEs specify the mathematical structure of the model explicitly, including couplings between parameters. Grey-box modeling is useful when you know the relationships between variables, constraints on model behavior, or explicit equations representing system dynamics.

The toolbox supports both continuous-time and discrete-time linear and nonlinear models. However, because most laws of physics are expressed in continuous time, it is easier to construct models with physical insight in continuous time, rather than in discrete time.

In addition to dynamic input-output models, you can also create time-series models that have no inputs and static models that have no states.

If it is too difficult to describe your system using known physical laws, you can use the black-box modeling approach. For more information, see "Linear Model Identification" and "Nonlinear Model Identification".

You can also use the `idss` model object to perform structured model estimation by using its `Structure` property to fix or free specific parameters. However, you cannot use this approach to estimate arbitrary structures (arbitrary parameterization). For more information about structure matrices, see "How to Estimate State-Space Models with Structured Parameterization" on page 3-100.

Data Supported by Grey-Box Models

You can estimate both continuous-time or discrete-time grey-box models for data with the following characteristics:

- Time-domain or frequency-domain data, including time-series data with no inputs.

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

- Single-output or multiple-output data

You must first import your data into the MATLAB workspace. You must represent your data as an `iddata` or `idfrd` object. For more information about preparing data for identification, see “Data Preparation”.

Choosing idgrey or idnlgrey Model Object

Grey-box models require that you specify the structure of the ODE model in a file. You use this file to create the `idgrey` or `idnlgrey` model object. You can use both the `idgrey` and the `idnlgrey` objects to model linear systems. However, you can only represent nonlinear dynamics using the `idnlgrey` model object.

The `idgrey` object requires that you write a function to describe the linear dynamics in the state-space form, such that this file returns the state-space matrices as a function of your parameters. For more information, see "Specifying the Linear Grey-Box Model Structure" on page 5-6.

The `idnlgrey` object requires that you write a function or MEX-file to describe the dynamics as a set of first-order differential equations, such that this file returns the output and state derivatives as a function of time, input, state, and parameter values. For more information, see "Specifying the Nonlinear Grey-Box Model Structure" on page 5-17.

The following table compares `idgrey` and `idnlgrey` model objects.

Comparison of idgrey and idnlgrey Objects

Settings and Operations	Supported by idgrey?	Supported by idnlgrey?
Set bounds on parameter values.	Yes	Yes
Handle initial states individually.	Yes	Yes
Perform linear analysis.	Yes For example, use the `bode` command.	No

Comparison of idgrey and idnlgrey Objects (Continued)

Settings and Operations	Supported by idgrey?	Supported by idnlgrey?
Honor stability constraints.	Yes Specify constraints using the `Advanced.StabilityThreshold` estimation option. For more information, see `greyestOptions`.	No **Note** You can use parameter bounds to ensure stability of an `idnlgrey` model, if these bounds are known.
Estimate a disturbance model.	Yes The disturbance model is represented by `K` in state-space equations.	No
Optimize estimation results for simulation or prediction.	Yes Set the `Focus` estimation option to `'Simulation'` or `'Prediction'`. For more information, see `greyestOptions`.	No Because `idnlgrey` models are Output-Error models, there is no difference between simulation and prediction results.

Estimating Linear Grey-Box Models

In this section...

Specifying the Linear Grey-Box Model Structure

You can estimate linear discrete-time and continuous-time grey-box models for arbitrary ordinary differential or difference equations using single-output and multiple-output time-domain data, or time-series data (output-only).

You must represent your system equations in state-space form. *State-space models* use state variables *x(t)* to describe a system as a set of first-order differential equations, rather than by one or more *n*th-order differential equations.

The first step in grey-box modeling is to write a function that returns state-space matrices as a function of user-defined parameters and information about the model.

Use the following format to implement the linear grey-box model in the file:

```
[A,B,C,D] = myfunc(par1,par2,...,parN,Ts,aux1,aux2,...)
```

where the output arguments are the state-space matrices and `myfunc` is the name of the file. `par1,par2,...,parN` are the *N* parameters of the model. Each entry may be a scalar, vector or matrix.`Ts` is the sample time. `aux1,aux2,...` are the optional input arguments that `myfunc` uses to compute the state-space matrices in addition to the parameters and sample time.

`aux` contains auxiliary variables in your system. You use auxiliary variables to vary system parameters at the input to the function, and avoid editing the file.

You can write the contents of `myfunc` to parameterize either a continuous-time, or a discrete-time state-space model, or both. When you create the `idgrey` model using `myfunc`, you can specify the nature of the output arguments of `myfunc`. The continuous-time state-space model has the form:

In continuous-time, the state-space description has the following form:

$$\dot{x}(t) = Ax(t) + Bu(t) + Ke(t)$$
$$y(t) = Cx(t) + Du(t) + e(t)$$
$$x(0) = x0$$

Where, A,B,C and D are matrices that are parameterized by the parameters `par1,par2,...,parN`. The noise matrix K and initial state vector, $x0$, are not parameterized by `myfunc`. They can still be estimated as additional quantities along with the model parameters. To configure the handling of initial states, $x0$, and the disturbance component, K, during estimation, use the `greyestOptions` option set.

In discrete-time, the state-space description has a similar form:

$$x(k+1) = Ax(k) + Bu(k) + Ke(t)$$
$$y(k) = Cx(k) + Du(k) + e(t)$$
$$x(0) = x0$$

Where, A,B,C and D are now the discrete-time matrices that are parameterized by the parameters `par1,par2,...,parN`. K and $x0$ are not directly parameterized, but can be estimated if required by configuring the corresponding estimation options.

Parameterizing Disturbance Model and Initial States

In some applications, you may want to express K and $x0$ as quantities that are parameterized by chosen parameters, just as the A, B, C and D matrices. To handle such cases, you can write the ODE file, `myfunc`, to return K and $x0$ as additional output arguments:

```
[A,B,C,D,K,x0] = myfunc(par1,par2,...,parN,Ts,aux1,aux2,...)
```

K and *x0* are thus treated in the same way as the *A*, *B*, *C* and *D* matrices. They are all functions of the parameters `par1,par2,...,parN`.

Constructing the idgrey Object

After creating the function or MEX-file with your model structure, you must define as `idgrey` object. For information regarding creating this, see `idgrey`.

Create Function to Represent a Grey-Box Model

This example shows how to represent the structure of the following continuous-time model:

$$\dot{x}(t) = \begin{bmatrix} 0 & 1 \\ 0 & \theta_1 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ \theta_2 \end{bmatrix} u(t)$$

$$y(t) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x(t) + e(t)$$

$$x(0) = \begin{bmatrix} \theta_3 \\ 0 \end{bmatrix}$$

This equation represents an electrical motor, where $y_1(t) = x_1(t)$ is the angular position of the motor shaft, and $y_2(t) = x_2(t)$ is the angular velocity.

The parameter $-\theta_1$ is the inverse time constant of the motor, and $-\theta_2/\theta_1$ is the static gain from the input to the angular velocity.

The motor is at rest at *t*=0, but its angular position θ_3 is unknown. Suppose that the approximate nominal values of the unknown parameters are $\theta_1 = -1$, $\theta_2 = 0.25$ and $\theta_3 = 0$. For more information about this example, see the section on state-space models in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

The continuous-time state-space model structure is defined by the following equation:

$$\dot{x}(t) = Fx(t) + Gu(t) + \tilde{K}w(t)$$
$$y(t) = Hx(t) + Du(t) + w(t)$$
$$x(0) = x0$$

To prepare this model for identification:

1. Create the following file to represent the model structure in this example:

```
function [A,B,C,D,K,x0] = myfunc(par,T)
A = [0 1; 0 par(1)];
B = [0;par(2)];
C = eye(2);
D = zeros(2,1);
K = zeros(2,2);
x0 =[par(3);0];
```

2. Use the following syntax to define an `idgrey` model object based on the `myfunc` file:

```
par = [-1; 0.25; 0];
aux = {};
T = 0;
m = idgrey('myfunc',par,'c',aux,T);
```

   where `par` represents a vector of all the user-defined parameters and contains their nominal (initial) values. In this example, all the scalar-valued parameters are grouped in the `par` vector. The scalar-valued parameters could also have been treated as independent input arguments to the ODE function `myfunc`.`'c'` specifies that the underlying parameterization is in continuous time. `aux` represents optional arguments. As `myfunc` does not have any optional arguments, use `aux = {}`. `T` specifies the sample interval; `T = 0` indicates a continuous-time model.

Use `greyest` to estimate the grey-box parameter values:

```
m_est = greyest(data,m)
```

where `data` is the estimation data and `m` is an estimation initialization `idgrey` model. `m_est` is the estimated `idgrey` model.

Note Compare this example to "Example – Estimating Structured Continuous-Time State-Space Models" on page 3-105, where the same problem is solved using a structured state-space representation.

Estimate Continuous-Time Grey-Box Model for Heat Diffusion

This example shows how to estimate the heat conductivity and the heat-transfer coefficient of a continuous-time grey-box model for a heated-rod system.

This system consists of a well-insulated metal rod of length L and a heat-diffusion coefficient κ. The input to the system is the heating power *u(t)* and the measured output *y(t)* is the temperature at the other end.

Under ideal conditions, this system is described by the heat-diffusion equation—which is a partial differential equation in space and time.

$$\frac{\partial x(t,\xi)}{\partial t} = \kappa \frac{\partial^2 x(t,\xi)}{\partial \xi^2}$$

To get a continuous-time state-space model, you can represent the second-derivative using the following difference approximation:

$$\frac{\partial^2 x(t,\xi)}{\partial \xi^2} = \frac{x(t,\xi+\Delta L) - 2x(t,\xi) + x(t,\xi-\Delta L)}{(\Delta L)^2}$$

where $\xi = k \cdot \Delta L$

This transformation produces a state-space model of order $n = \frac{L}{\Delta L}$, where the state variables $x(t, k \cdot \Delta L)$ are lumped representations for $x(t, \xi)$ for the following range of values:

$$k \cdot \Delta L \le \xi < (k+1)\Delta L$$

The dimension of x depends on the spatial grid size ΔL in the approximation.

The heat-diffusion equation is mapped to the following continuous-time state-space model structure to identify the state-space matrices:

$$\dot{x}(t) = Fx(t) + Gu(t) + \tilde{K}w(t)$$
$$y(t) = Hx(t) + Du(t) + w(t)$$
$$x(0) = x0$$

The state-space matrices are parameterized by the heat diffusion coefficient κ and the heat transfer coefficient at the far end of the rod h_{tf}. The expressions also depend upon the grid size, *Ngrid*, and the length of the rod L. The initial conditions x0 are a function of the initial room temperature, treated as a known quantity in this example.

1 Create a MATLAB file.

The following code describes the state-space equation for this model. The parameters are κ and h_{tf} while the auxiliary variables are *Ngrid*, *L* and initial room temperature `temp`. The grid size is supplied as an auxiliary variable so that the ODE function can be easily adapted for various grid sizes.

```
function [A,B,C,D,K,x0] = heatd(kappa, htf, T, Ngrid, L, temp)
% ODE file parameterizing the heat diffusion model

% kappa (first parameter) - heat diffusion coefficient
% htf (second parameter) - heat transfer coefficient
%                          at the far end of rod

% Auxiliary variables for computing state-space matrices:
% Ngrid: Number of points in the space-discretization
```

```
% L: Length of the rod
% temp: Initial room temperature (uniform)

% Compute space interval
deltaL = L/Ngrid;

% A matrix
A = zeros(Ngrid,Ngrid);
for kk = 2:Ngrid-1
  A(kk,kk-1) = 1;
  A(kk,kk) = -2;
  A(kk,kk+1) = 1;
end

% Boundary condition on insulated end
A(1,1) = -1; A(1,2) = 1;
A(Ngrid,Ngrid-1) = 1;
A(Ngrid,Ngrid) = -1;
A = A*kappa/deltaL/deltaL;

% B matrix
B = zeros(Ngrid,1);
B(Ngrid,1) = htf/deltaL;

% C matrix
C = zeros(1,Ngrid);
C(1,1) = 1;

% D matrix (fixed to zero)
D = 0;

% K matrix: fixed to zero
K = zeros(Ngrid,1);

% Initial states: fixed to room temperature
x0 = temp*ones(Ngrid,1);
```

2 Use the following syntax to define an `idgrey` model object based on the `heatd` code file:

```
m = idgrey('heatd', {0.27 1}, 'c', {10, 1, 22});
```

This command specifies the auxiliary parameters as inputs to the function, include the model order (grid size) `10`, the rod length of 1 meter, and an initial temperature of `22` degrees Celsius. The command also specifies the initial values for heat conductivity as `0.27`, and for the heat transfer coefficient as `1`.

3. For given `data`, you can use `greyest` to estimate the grey-box parameter values:

```
me = greyest(data,m)
```

The following command shows how you can specify to estimate a new model with different auxiliary variables:

```
m.Structure.ExtraArgs = {20,1,22};
me = greyest(data,m);
```

This syntax uses the `ExtraArgs` model structure attribute to specify a finer grid using a larger value for `Ngrid`. For more information about linear grey-box model properties, see the `idgrey` reference page.

Estimate Discrete-Time Grey-Box Model with Parameterized Disturbance

This example shows how to create a single-input and single-output grey-box model structure when you know the variance of the measurement noise. The code in this example uses the Control System Toolbox command `kalman` for computing the Kalman gain from the known and estimated noise variance.

Description of the SISO System

This example is based on a discrete, single-input and single-output (SISO) system represented by the following state-space equations:

$$x(kT+T) = \begin{bmatrix} par1 & par2 \\ 1 & 0 \end{bmatrix} x(kT) + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u(kT) + w(kT)$$

$$y(kT) = \begin{bmatrix} par3 & par4 \end{bmatrix} x(kT) + e(kT)$$

$$x(0) = x0$$

where w and e are independent white-noise terms with covariance matrices *R1* and *R2*, respectively. *R1*=E{ww'} is a 2–by-2 matrix and *R2*=E{ee'} is a scalar. *par1*, *par2*, *par3*, and *par4* represent the unknown parameter values to be estimated.

Assume that you know the variance of the measurement noise *R2* to be 1. *R1*(1,1) is unknown and is treated as an additional parameter *par5*. The remaining elements of *R1* are known to be zero.

Estimating the Parameters of an idgrey Model

You can represent the system described in "Description of the SISO System" on page 5-14 as an `idgrey` (grey-box) model using a function. Then, you can use this file and the `greyest` command to estimate the model parameters based on initial parameter guesses.

To run this example, you must load an input-output data set and represent it as an `iddata` or `idfrd` object called `data`. For more information about this operation, see "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-55 or "Representing Frequency-Response Data Using idfrd Objects" on page 2-76.

To estimate the parameters of a grey-box model:

1. Create the file `mynoise` that computes the state-space matrices as a function of the five unknown parameters and the auxiliary variable

that represents the known variance `R2`. The initial conditions are not parameterized; they are assumed to be zero during this estimation.

Note `R2` is treated as an auxiliary variable rather than assigned a value in the file to let you change this value directly at the command line and avoid editing the file.

```
function [A,B,C,D,K] = 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] = kalman(ss(A,eye(2),C,0,T),R1,R2);
      % Uses Control System Toolbox product
      % u=[]
```

2 Specify initial guesses for the unknown parameter values and the auxiliary parameter value `R2`:

```
par1 = 0.1; % Initial guess for A(1,1)
par2 = -2;  % Initial guess for A(1,2)
par3 = 1;   % Initial guess for C(1,1)
par4 = 3;   % Initial guess for C(1,2)
par5 = 0.2; % Initial guess for R1(1,1)
Pvec = [par1; par2; par3; par4; par5]
auxVal = 1; % R2=1
```

3 Construct an `idgrey` model using the `mynoise` file:

```
Minit = idgrey('mynoise',Pvec,'d',auxVal);
```

The third input argument `'d'` specifies a discrete-time system.

4 Estimate the model parameter values from data:

```
opt = greyestOptions;
opt.InitialState = 'zero';
```

```
opt.Display = 'full';
Model = greyest(data,Minit,opt)
```

Estimating Nonlinear Grey-Box Models

In this section...

Specifying the Nonlinear Grey-Box Model Structure

You must represent your system as a set of first-order nonlinear difference or differential equations:

$$x^{\dagger}(t) = F(t, x(t), u(t), par1, par2, ..., parN)$$
$$y(t) = H(t, x(t), u(t), par1, par2, ..., parN) + e(t)$$
$$x(0) = x0$$

where $x^{\dagger}(t) = {dx(t)}/{dt}$ for continuous-time representation and $x^{\dagger}(t) = x(t + T_s)$ for discrete-time representation with *Ts* as the sampling interval. *F* and *H* are arbitrary linear or nonlinear functions with *Nx* and *Ny* components, respectively. *Nx* is the number of states and *Ny* is the number of outputs.

After you establish the equations for your system, create a function or MEX-file. MEX-files, which can be created in C or Fortran, are dynamically linked subroutines that can be loaded and executed by the MATLAB interpreter. For more information about MEX-files, see "Create MEX-Files".

The purpose of the model file is to return the state derivatives and model outputs as a function of time, states, inputs, and model parameters, as follows:

```
[dx,y] = MODFILENAME(t,x,u,p1,p2, ...,pN,FileArgument)
```

Tip The template file for writing the C MEX-file, `IDNLGREY_MODEL_TEMPLATE.c`, is located in `matlab/toolbox/ident/nlident`.

The output variables are:

- `dx` — Represents the right side(s) of the state-space equation(s). A column vector with *Nx* entries. For static models, `dx=[]`.

  For discrete-time models. `dx` is the value of the states at the next time step `x(t+Ts)`.

  For continuous-time models. `dx` is the state derivatives at time *t*, or $\frac{dx}{dt}$.
- `y` — Represents the right side(s) of the output equation(s). A column vector with *Ny* entries.

The file inputs are:

- `t` — Current time.
- `x` — State vector at time `t`. For static models, equals `[]`.
- `u` — Input vector at time `t`. For time-series models, equals `[]`.
- `p1,p2, ...,pN` — Parameters, which can be real scalars, column vectors or two-dimensional matrices. `N` is the number of parameter objects. For scalar parameters, `N` is the total number of parameter elements.
- `FileArgument` — Contains auxiliary variables that might be required for updating the constants in the state equations.

Tip After creating a model file, call it directly from the MATLAB software with reasonable inputs and verify the output values.

For an example of creating grey-box model files and `idnlgrey` model object, see Creating idnlgrey Model Files.

For examples of code files and MEX-files that specify model structure, see the `toolbox/ident/iddemos/examples` folder. For example, the model of a DC motor is described in files `dcmotor_m` and `dcmotor_c`.

Constructing the idnlgrey Object

After you create the function or MEX-file with your model structure, you must define an `idnlgrey` object. This object shares many of the properties of the linear `idgrey` model object.

Use the following syntax to define the `idnlgrey` model object:

```
m = idnlgrey('filename',Order,Parameters,InitialStates)
```

The `idnlgrey` arguments are defined as follows:

- `'filename'` — Name of the function or MEX-file storing the model structure. This file must be on the MATLAB path when you use this model object for model estimation, prediction, or simulation.
- `Order` — Vector with three entries `[Ny Nu Nx]`, specifying the number of model outputs `Ny`, the number of inputs `Nu`, and the number of states `Nx`.
- `Parameters` — Parameters, specified as `struct` arrays, cell arrays, or double arrays.
- `InitialStates` — Specified in the same way as parameters. Must be the fourth input to the `idnlgrey` constructor.

For detailed information about this object and its properties, see the `idnlgrey` reference page.

Use `pem` to estimate your grey-box model.

Using pem to Estimate Nonlinear Grey-Box Models

You can use the `pem` command to estimate the unknown `idnlgrey` model parameters and initial states using measured data.

The input-output dimensions of the data must be compatible with the input and output orders you specified for the `idnlgrey` model.

Use the following general estimation syntax:

```
m = pem(data,m)
```

where `data` is the estimation data and `m` is the `idnlgrey` model object you constructed.

You can pass additional property-value pairs to `pem` to specify the properties of the model or the estimation algorithm. Assignable properties include the ones returned by the `get(idnlgrey)` command and the algorithm properties returned by the `get(idnlgrey, 'Algorithm')`, such as `MaxIter` and `Tolerance`. For detailed information about these model properties, see the `idnlgrey` reference page.

For more information about validating your models, see “Model Validation”.

Nonlinear Grey-Box Model Estimation Algorithm Options

The `Algorithm` property of the model specifies the estimation algorithm, which simulates the model several times by trying various parameter values to reduce the prediction error.

The following algorithm properties can affect the quality of the results:

For detailed information about these and other model properties, see the `idnlgrey` reference page.

Simulation Method

You can specify the simulation method using the `SimulationOptions` (`struct`) fields of the model `Algorithm` property.

System Identification Toolbox software provides several variable-step and fixed-step solvers for simulating `idnlgrey` models. To view a list of available solvers and their properties, type the following command at the prompt:

```
idprops idnlgrey algorithm.simulationoptions
```

For discrete-time systems, the default solver is `'FixedStepDiscrete'`. For continuous-time systems, the default solver is `'ode45'`.

By default, `SimulationOptions.Solver` is set to `'Auto'`, which automatically selects either `'ode45'` or `'FixedStepDiscrete'` during estimation and simulation—depending on whether the system is continuous or discrete in time.

Search Method

You can specify the search method for estimating model parameters using the `SearchMethod` field of the `Algorithm` property. Two categories of methods are available for nonlinear grey-box modeling.

One category of methods consists of the minimization schemes that are based on line-search methods, including Gauss-Newton type methods, steepest-descent methods, and Levenberg-Marquardt methods.

The Trust-Region Reflective Newton method of nonlinear least-squares (`lsqnonlin`), where the cost is the sum of squares of errors between the measured and simulated outputs, requires Optimization Toolbox™ software. When the parameter bounds differ from the default +/- Inf, this search method handles the bounds better than the schemes based on a line search. However, unlike the line-search-based methods, `lsqnonlin` only works with `Criterion='Trace'`.

By default, `SearchMethod` is set to `Auto`, which automatically selects a method from the available minimizers. If the Optimization Toolbox product is installed, `SearchMethod` is set to `'lsqnonlin'`. Otherwise, `SearchMethod` is a combination of line-search based schemes.

Gradient Options

You can specify the method for calculating gradients using the `GradientOptions` field of the `Algorithm` property. *Gradients* are the derivatives of errors with respect to unknown parameters and initial states.

Gradients are calculated by numerically perturbing unknown quantities and measuring their effects on the simulation error.

Option for gradient computation include the choice of the differencing scheme (forward, backward or central), the size of minimum perturbation of the unknown quantities, and whether the gradients are calculated simultaneously or individually.

Example – Specifying Algorithm Properties

You can specify the `Algorithm` fields directly in the estimation syntax, as property-value pairs.

For example, you can specify the following properties as part of the `pem` syntax:

```
m = pem(data,init_model,'Search','gn',...
                        'MaxIter',5,...
                        'Display','On')
```

Represent Nonlinear Dynamics Using MATLAB File for Grey-Box Estimation

This example shows how to construct, estimate and analyze nonlinear grey-box models.

Nonlinear grey-box (`idnlgrey`) models are suitable for estimating parameters of systems that are described by nonlinear state-space structures in continuous or discrete time. You can use both `idgrey` (linear grey-box model) and `idnlgrey` objects to model linear systems. However, you can only use `idnlgrey` to represent nonlinear dynamics. To learn about linear grey-box modeling using `idgrey`, see "Building Structured and User-Defined Models Using System Identification Toolbox™".

About the Model

In this example, you model the dynamics of a linear DC motor using the `idnlgrey` object.

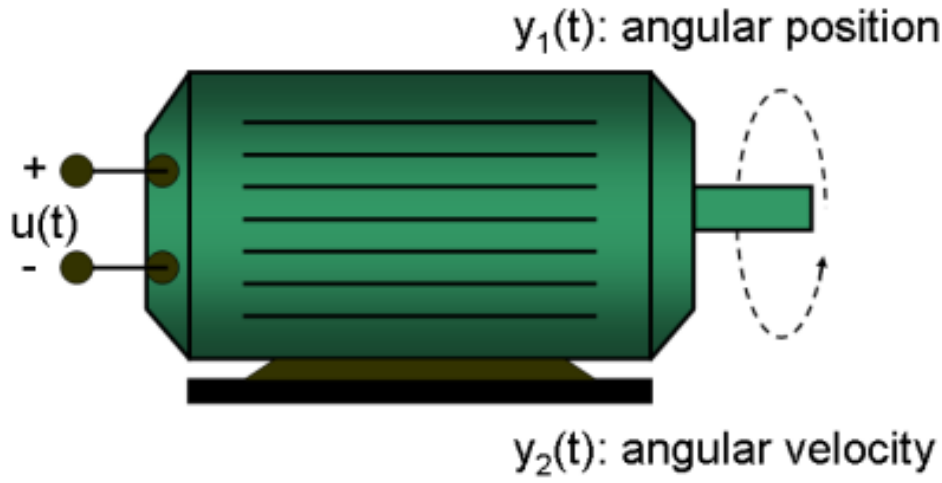


Figure 1: Schematic diagram of a DC-motor.

If you ignore the disturbances and choose `y(1)` as the angular position [rad] and `y(2)` as the angular velocity [rad/s] of the motor, you can set up a linear state-space structure of the following form (see Ljung, L. System Identification: Theory for the User, Upper Saddle River, NJ, Prentice-Hall PTR, 1999, 2nd ed., p. 95-97 for the derivation):

```
d          | 0      1   |         |   0   |
-- x(t) = |             | x(t) + |       | u(t)
dt         | 0   -1/tau |         | k/tau |

           | 1   0 |
   y(t) = |       | x(t)
           | 0   1 |
```

`tau` is the time-constant of the motor in [s] and `k` is the static gain from the input to the angular velocity in [rad/(V*s)] . See Ljung (1999) for how `tau` and `k` relate to the physical parameters of the motor.

About the Input-Output Data

1. Load the DC motor data.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'dcmotorda
```

2. Represent the estimation data as an `iddata` object.

```
z = iddata(y, u, 0.1, 'Name', 'DC-motor');
```

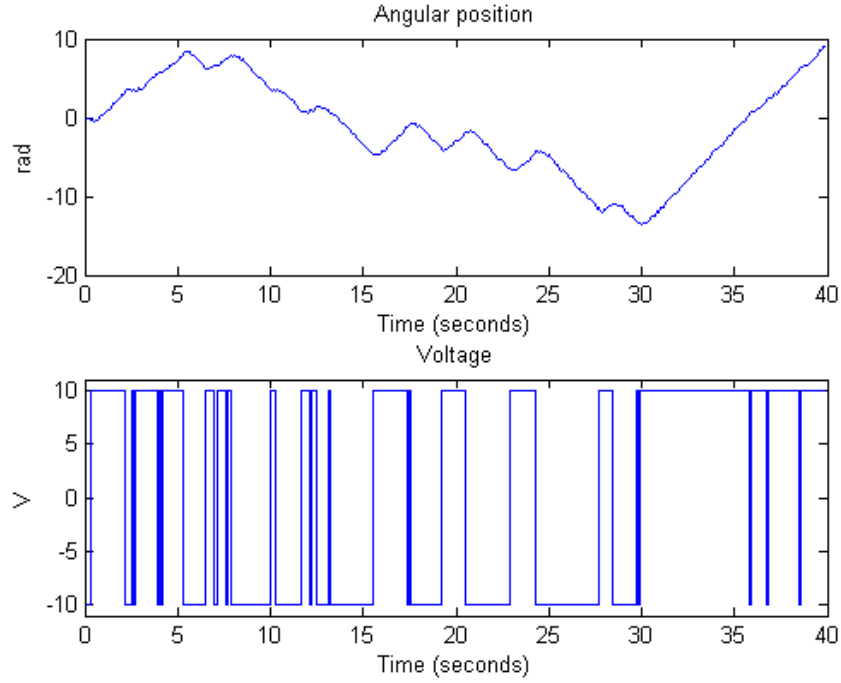
3. Specify input and output signal names, start time and time units.

```
set(z, 'InputName', 'Voltage', 'InputUnit', 'V');
set(z, 'OutputName', {'Angular position', 'Angular velocity'});
set(z, 'OutputUnit', {'rad', 'rad/s'});
set(z, 'Tstart', 0, 'TimeUnit', 's');
```

4. Plot the data.

The data is shown in two plot windows.

```
figure('Name', [z.Name ': Voltage input -> Angular position output']);
plot(z(:, 1, 1));   % Plot first input-output pair (Voltage -> Angular posi
figure('Name', [z.Name ': Voltage input -> Angular velocity output']);
plot(z(:, 2, 1));   % Plot second input-output pair (Voltage -> Angular vel
```


Angular position
rad
10
0
-10
-20
0
5
10
15
20
25
30
35
40
Time (seconds)
Voltage
V
10
5
0
-5
-10
0
5
10
15
20
25
30
35
40
Time (seconds)

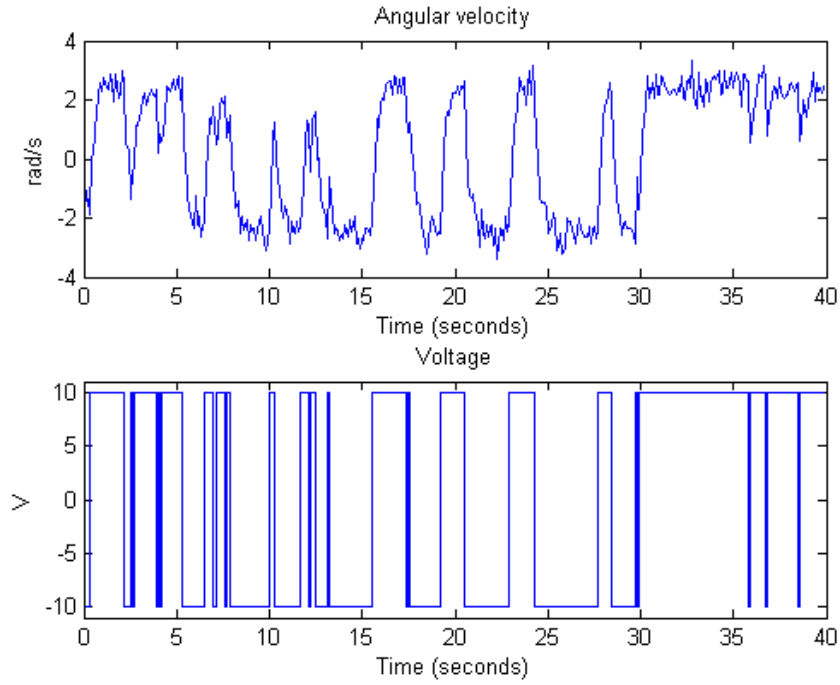


Figure 2: Input-output data from a DC-motor.

Linear Modeling of the DC-Motor

1. Represent the DC motor structure in a function.

In this example, you use a MATLAB file, but you can also use C MEX-files (to gain computational speed), P-files or function handles. For more information, see "Creating IDNLGREY Model Files".

The DC-motor function is called `dcmotor_m.m` and is shown below.

```
function [dx, y] = dcmotor_m(t, x, u, tau, k, varargin)

% Output equations.
y = [x(1);                          ... % Angular position.
     x(2)                           ... % Angular velocity.
```

```
    ];

% State equations.
dx = [x(2);                          ... % Angular position.
      -(1/tau)*x(2)+(k/tau)*u(1)     ... % Angular velocity.
     ];
```

The file must always be structured to return the following:

Output arguments:

- `dx` is the right-hand side(s) of the state-space equation(s)
- `y` is the output equation(s)

Input arguments:

- The first three input arguments must be: `t` (time), `x` (state vector, [] for static systems), `u` (input vector, [] for time-series).
- Ordered list of parameters follow. The parameters can be scalars, column vectors, or 2-dimensional matrices.
- `varargin` for the auxiliary input arguments

2. Represent the DC motor dynamics using an `idnlgrey` object.

The model describes how the inputs generate the outputs using the state equation(s).

```
FileName      = 'dcmotor_m';       % File describing the model structure.
Order         = [2 1 2];           % Model orders [ny nu nx].
Parameters    = [1; 0.28];         % Initial parameters. Np = 2.
InitialStates = [0; 0];            % Initial initial states.
Ts            = 0;                 % Time-continuous system.
nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts, ...
                'Name', 'DC-motor');
```

In practice, there are disturbances that affect the outputs. An `idnlgrey` model does not explicitly model the disturbances, but assumes that these

are just added to the output(s). Thus, `idnlgrey` models are equivalent to Output-Error (OE) models. Without a noise model, past outputs do not influence prediction of future outputs, which means that predicted output for any prediction horizon k coincide with simulated outputs.

3. Specify input and output names, and units.

```
set(nlgr, 'InputName', 'Voltage', 'InputUnit', 'V',                ...
          'OutputName', {'Angular position', 'Angular velocity'}, ...
          'OutputUnit', {'rad', 'rad/s'},                          ...
          'TimeUnit', 's');
```

4. Specify names and units of the initial states and parameters.

```
setinit(nlgr, 'Name', {'Angular position' 'Angular velocity'});
setinit(nlgr, 'Unit', {'rad' 'rad/s'});
setpar(nlgr, 'Name', {'Time-constant' 'Static gain'});
setpar(nlgr, 'Unit', {'s' 'rad/(V*s)'});
```

You can also use `setinit` and `setpar` to assign values, minima, maxima, and estimation status for all initial states or parameters simultaneously.

5. View the initial model.

a. Get basic information about the model.

The DC-motor has 2 (initial) states and 2 model parameters.

```
size(nlgr)
```

```
Nolinear grey-box model with 2 outputs, 1 inputs, 2 states and 2 parameters
```

b. View the initial states and parameters.

Both the initial states and parameters are structure arrays. The fields specify the properties of an individual initial state or parameter. Type `idprops idnlgrey InitialStates` and `idprops idnlgrey Parameters` for more information.

```
nlgr.InitialStates(1)
nlgr.Parameters(2)
```

```
ans =

       Name: 'Angular position'
       Unit: 'rad'
      Value: 0
    Minimum: -Inf
    Maximum: Inf
      Fixed: 1


ans =

       Name: 'Static gain'
       Unit: 'rad/(V*s)'
      Value: 0.2800
    Minimum: -Inf
    Maximum: Inf
      Fixed: 0
```

c. Retrieve information for all initial states or model parameters in one call.

For example, obtain information on initial states that are fixed (not estimated) and the minima of all model parameters.

```
getinit(nlgr, 'Fixed')
getpar(nlgr, 'Min')
```

```
ans =

    [1]
    [1]


ans =

    [-Inf]
```

```
    [-Inf]
```

d. Obtain basic information about the object:

```
nlgr
```

```
Continuous-time nonlinear grey-box model defined by 'dcmotor_m' (MATLAB fil

   dx/dt = F(t, u(t), x(t), p1, p2)
    y(t) = H(t, u(t), x(t), p1, p2) + e(t)

with 1 input, 2 states, 2 outputs, and 2 free parameters (out of 2).
```

Use `get` to obtain more information about the model properties. The `idnlgrey` object shares many properties of parametric linear model objects.

```
get(nlgr)
```

```
           FileName: 'dcmotor_m'
              Order: [1x1 struct]
         Parameters: [2x1 struct]
      InitialStates: [2x1 struct]
       FileArgument: {}
    CovarianceMatrix: 'estimate'
      EstimationInfo: [1x1 struct]
        TimeVariable: 't'
       NoiseVariance: [2x2 double]
           Algorithm: [1x1 struct]
                  Ts: 0
            TimeUnit: 'seconds'
           InputName: {'Voltage'}
           InputUnit: {'V'}
          InputGroup: [1x1 struct]
          OutputName: {2x1 cell}
          OutputUnit: {2x1 cell}
         OutputGroup: [1x1 struct]
                Name: 'DC-motor'
               Notes: {}
            UserData: []
```

Performance Evaluation of the Initial DC-Motor Model

Before estimating the parameters `tau` and `k`, simulate the output of the system with the parameter guesses using the default differential equation solver (a Runge-Kutta 45 solver with adaptive step length adjustment).

1. Set the absolute and relative error tolerances to small values (`1e-6` and `1e-5`, respectively).

```
nlgr.Algorithm.SimulationOptions.AbsTol = 1e-6;
nlgr.Algorithm.SimulationOptions.RelTol = 1e-5;
```

2. Compare the simulated output with the measured data.

`compare` displays both measured and simulated outputs of one or more models, whereas `predict`, called with the same input arguments, displays the simulated outputs.

The simulated and measured outputs are shown in a plot window.

```
figure;
compare(z, nlgr);
```

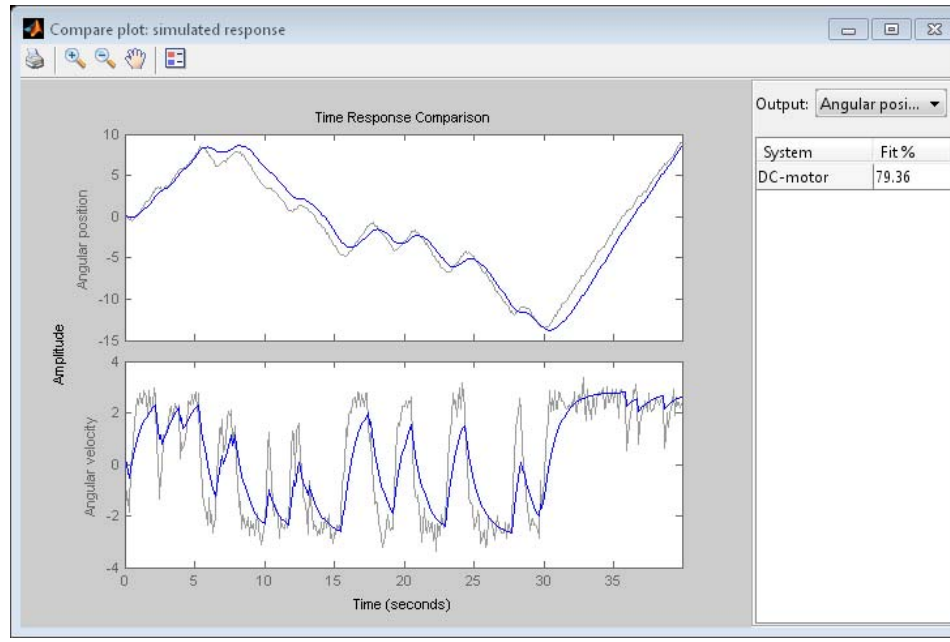


Figure 3: Comparison between measured outputs and the simulated outputs of the initial DC-motor model.

Parameter Estimation

Estimate the parameters and initial states using `pem` (Prediction-Error identification Method).

```
setinit(nlgr, 'Fixed', {false false});   % Estimate the initial state.
nlgr = pem(z, nlgr, 'Display', 'Full');
```

Performance Evaluation of the Estimated DC-Motor Model

1. Review the information about the estimation process.

This information is stored in the `EstimationInfo` property of the `idnlgrey` object. The property also contains information about how the model was estimated, such as solver and search method, data set, and why the estimation was terminated.

```
nlgr.EstimationInfo
```

```
ans =

             Status: 'Estimated model (PEM)'
             Method: 'Solver: ode45; Search: lsqnonlin'
            LossFcn: 0.0011
                FPE: 0.0011
           DataName: 'DC-motor'
         DataLength: 400
             DataTs: {[0.1000]}
         DataDomain: 'Time'
    DataInterSample: {'zoh'}
            WhyStop: 'Change in cost was less than the specified tolerance'
         UpdateNorm: []
    LastImprovement: []
         Iterations: 4
       InitialGuess: [1x1 struct]
            Warning: ''
     EstimationTime: 4.2432
```

2. Evaluate the model quality by comparing simulated and measured outputs.

The fits are 98% and 84%, which indicate that the estimated model captures the dynamics of the DC motor well.

```
figure;
compare(z, nlgr);
```

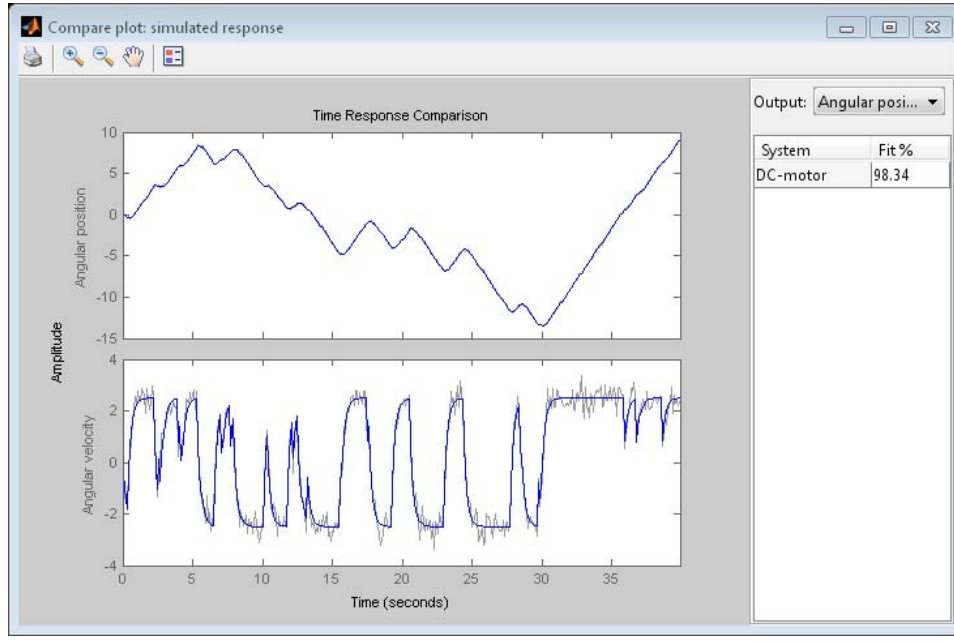


Figure 4: Comparison between measured outputs and the simulated outputs of the estimated IDNLGREY DC-motor model.

3. Compare the performance of the `idnlgrey` model with a second-order ARX model.

```
na = [2 2; 2 2];
nb = [2; 2];
nk = [1; 1];
dcarx = arx(z, [na nb nk]);
figure;
compare(z, nlgr, dcarx);
```

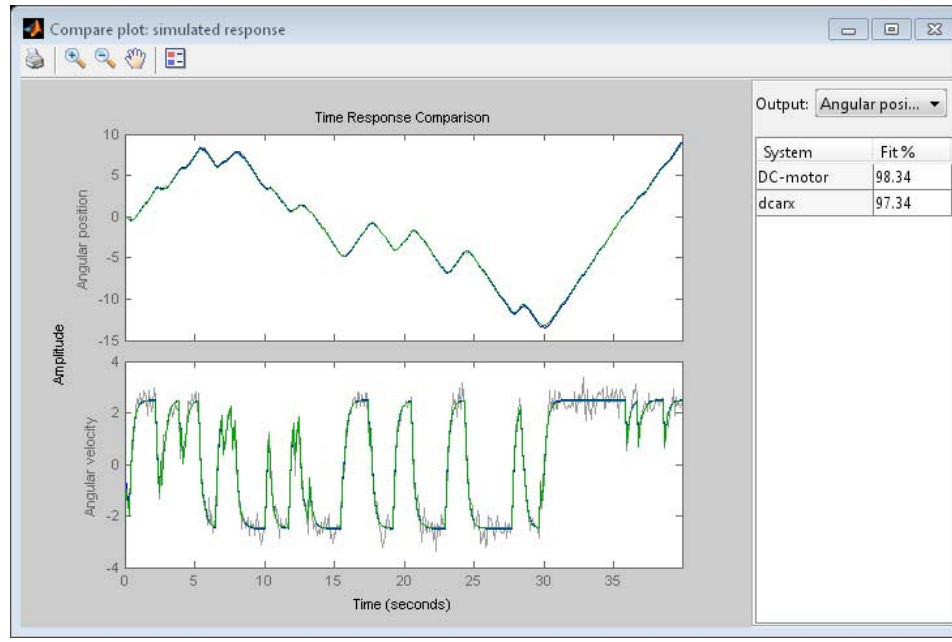


Figure 5: Comparison between measured outputs and the simulated outputs of the estimated IDNLGREY and ARX DC-motor models.

4. Check the prediction errors.

The prediction errors obtained are small and are centered around zero (non-biased).

```
figure;
pe(z, nlgr);
```

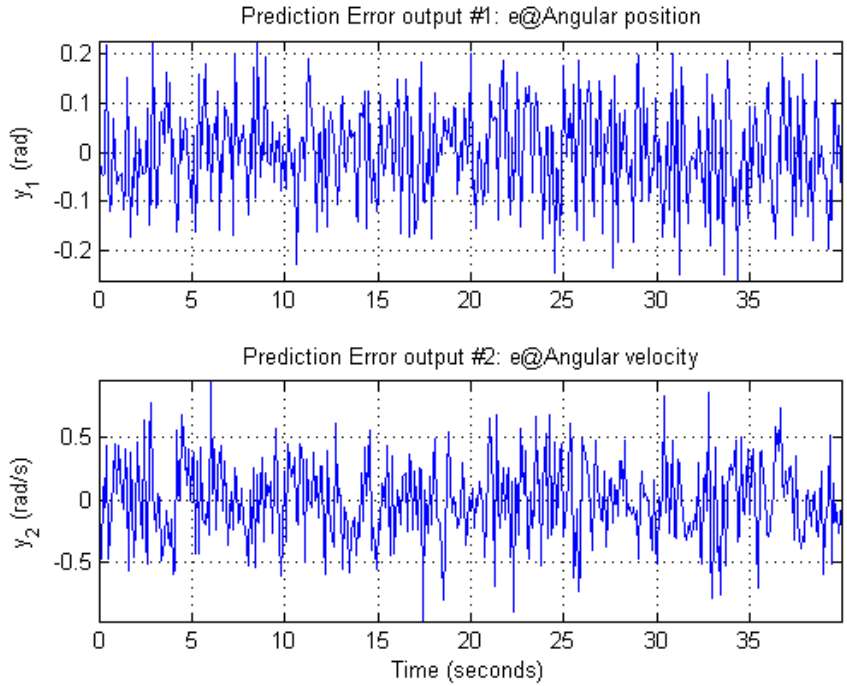
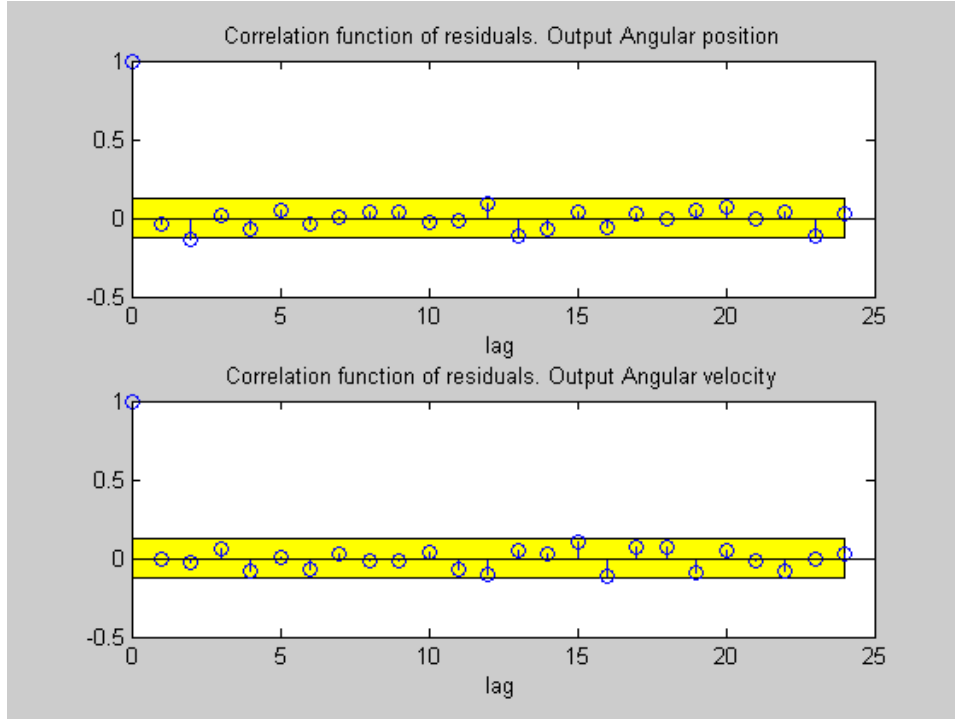


Figure 6: Prediction errors obtained with the estimated IDNLGREY DC-motor model.

5. Check the residuals ("leftovers").

Residuals indicate what is left unexplained by the model and are small for good model quality. Execute the following two lines of code to generate the residual plot. Press any key to advance from one plot to another.

```
figure('Name', [nlgr.Name ': residuals of estimated model']);
resid(z, nlgr);
```


Correlation function of residuals. Output Angular position
1
0.5
0
-0.5
0
5
10
15
20
25
lag
Correlation function of residuals. Output Angular velocity
1
0.5
0
-0.5
0
5
10
15
20
25
lag

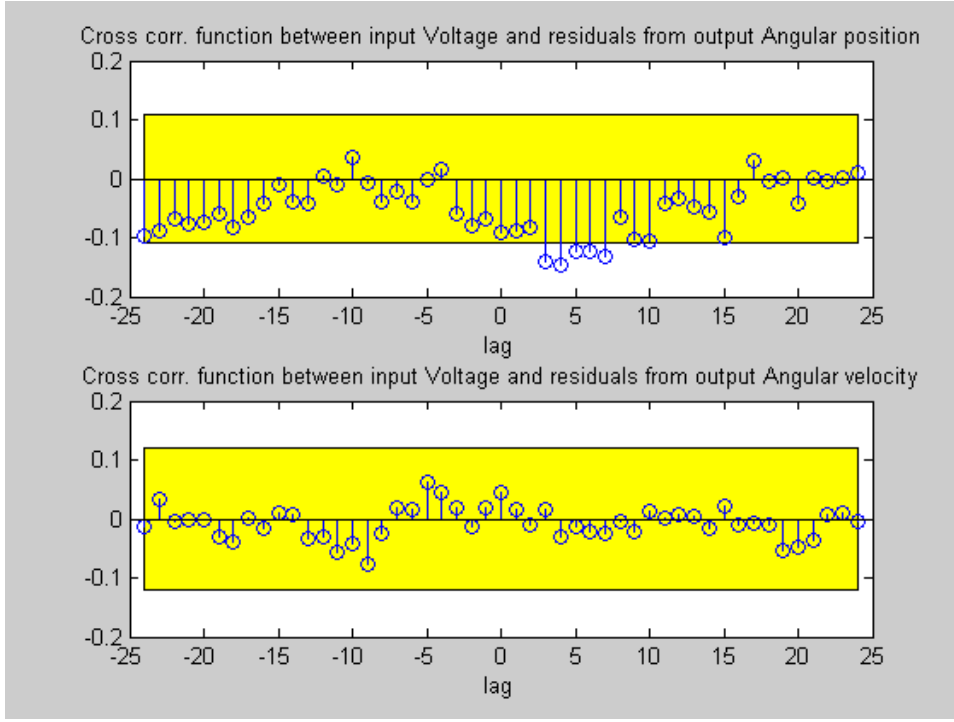


Figure 7: Residuals obtained with the estimated IDNLGREY DC-motor model.

6. Plot the step response.

A unit input step results in an angular position showing a ramp-type behavior and to an angular velocity that stabilizes at a constant level.

```
figure('Name', [nlgr.Name ': step response of estimated model']);
step(nlgr);
```

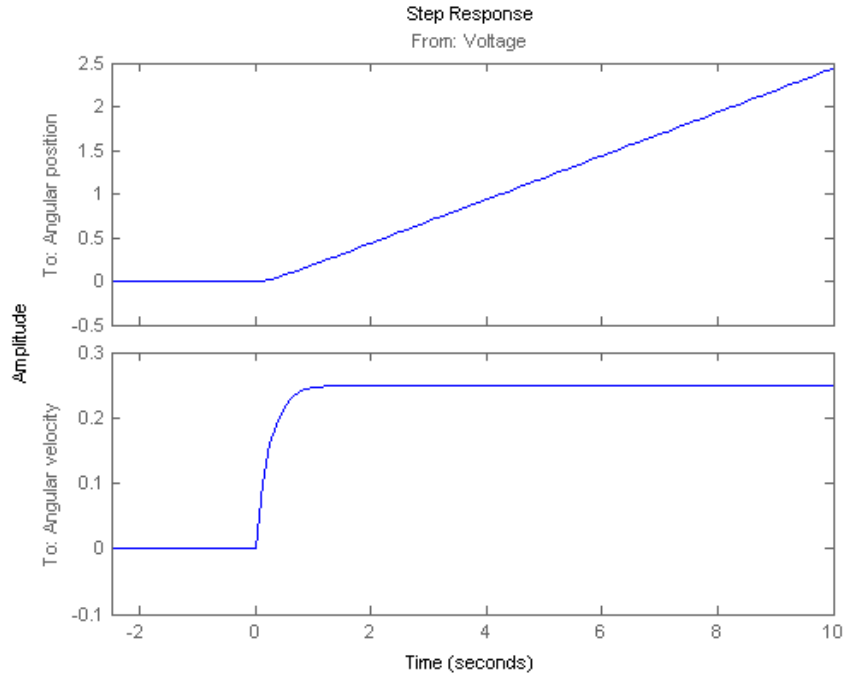


Figure 8: Step response with the estimated IDNLGREY DC-motor model.

7. Examine the model covariance.

You can assess the quality of the estimated model to some extent by looking at the estimated covariance matrix and the estimated noise variance. A "small" value of the `(i, i)` diagonal element of the covariance matrix indicates that the `i`:th model parameter is important for explaining the system dynamics when using the chosen model structure. Small noise variance (covariance for multi-output systems) elements are also a good indication that the model captures the estimation data in a good way.

```
nlgr.CovarianceMatrix
nlgr.NoiseVariance
```

```
ans =

   1.0e-04 *

    0.1521    0.0015
    0.0015    0.0007


ans =

    0.0099   -0.0004
   -0.0004    0.1094
```

For more information about the estimated model, use `present` to display the initial states and estimated parameter values, and estimated uncertainty (standard deviation) for the parameters.

```
present(nlgr);
```

```
Continuous-time nonlinear grey-box model defined by 'dcmotor_m' (MATLAB fil

   dx/dt = F(t, u(t), x(t), p1, p2)
    y(t) = H(t, u(t), x(t), p1, p2) + e(t)

with 1 input, 2 states, 2 outputs, and 2 free parameters (out of 2).

Input:
   u(1)  Voltage(t) [V]
States:                                  initial value
   x(1)  Angular position(t) [rad]     xinit@exp1   0.0302986   (est) in [-
   x(2)  Angular velocity(t) [rad/s]   xinit@exp1   -0.133728   (est) in [-
Outputs:
   y(1)  Angular position(t) [rad]
   y(2)  Angular velocity(t) [rad/s]
Parameters:                          value      standard dev
   p1   Time-constant [s]         0.243646   0.00390033   (est) in [-Inf, I
   p2   Static gain [rad/(V*s)]   0.249645   0.00027217   (est) in [-Inf, I

The model was estimated from the data set 'DC-motor', which
```

```
contains 400 data samples.
Loss function 0.00107462 and Akaike's FPE 0.00108536

Created:       17-Jan-2013 14:13:55
Last modified: 17-Jan-2013 14:14:05
```

Conclusions

This example illustrates the basic tools for performing nonlinear grey-box modeling. See the other nonlinear grey-box examples to learn about:

- Using nonlinear grey-box models in more advanced modeling situations, such as building nonlinear continuous- and discrete-time, time-series and static models.
- Writing and using C MEX model-files.
- Handling nonscalar parameters.
- Impact of certain algorithm choices.

For more information on identification of dynamic systems with System Identification Toolbox, visit the System Identification Toolbox product information page.

After Estimating Grey-Box Models

After estimating linear and nonlinear grey-box models, you can simulate the model output using the `sim` command. For more information, see "Validating Models After Estimation" on page 8-3.

The toolbox represents linear grey-box models using the `idgrey` model object. To convert grey-box models to state-space form, use the `idss` command, as described in "Transforming Between Linear Model Representations" on page 3-156. You must convert your model to an `idss` object to perform input-output concatenation or to use sample time conversion functions (`c2d`, `d2c`, `d2d`).

Note Sample-time conversion functions require that you convert `idgrey` models with `FcnType ='cd'` to `idss` models.

The toolbox represents nonlinear grey-box models as `idnlgrey` model objects. These model objects store the parameter values resulting from the estimation. You can access these parameters from the model objects to use these variables in computation in the MATLAB workspace.

Note Linearization of nonlinear grey-box models is not supported.

You can import nonlinear and linear grey box models into a Simulink model using the System Identification Toolbox Block Library. For more information, see "Simulating Identified Model Output in Simulink" on page 11-5.

Estimating Coefficients of ODEs to Fit Given Solution

This example shows how to estimate a model parameter using linear and nonlinear grey-box modeling.

Use grey-box identification to estimate coefficients of ODEs that describe the model dynamics to fit a given response trajectory.

- For linear dynamics, represent the model using a linear grey-box model (`idgrey`). Estimate the model coefficients using the command `greyest`.
- For nonlinear dynamics, represent the model using a nonlinear grey-box model (`idnlgrey`). Estimate the model coefficients using the command `pem`.

For this example, estimate the value of the friction coefficient of a simple pendulum using its oscillation data.

The motion of a simple pendulum can be described by:

$$ml^2\ddot{\theta} + b\dot{\theta} + mglsin\theta = 0$$

Θ is the angular displacement of the pendulum relative to its state of rest. g is the gravitational acceleration constant. m is the mass of the pendulum and l is the length of the pendulum. b is the viscous friction coefficient whose value will be estimated to fit the given angular displacement data. There is no external driving force that is contributing to the pendulum's motion.

Load measured data.

```
load(fullfile(matlabroot, 'toolbox', 'ident', ...
     'iddemos', 'data', 'pendulumdata'));

data = iddata(y, [], 0.1, 'Name', 'Pendulum');

set(data, 'OutputName', 'Pendulum position');
set(data, 'OutputUnit', 'rad');
set(data, 'Tstart', 0, 'TimeUnit', 's');
```

The measured angular displacement data is loaded and saved as `data`, an `iddata` object with a sampling time of 0.1 seconds. The `set` command is used

to specify data attributes such as the output name, output unit and the start time and units of the time vector.

Perform linear grey-box estimation.

Assuming that the pendulum undergoes only small angular displacements, the equation describing the pendulum motion can be simplified:

$$ml^2\ddot{\theta} + b\dot{\theta} + mgl\theta = 0$$

Using the angular displacement (θ) and the angular velocity ($\dot{\theta}$) as state variables, the simplified equation can be rewritten in the form:

$$\dot{X}(t) = AX(t) + Bu(t)$$
$$y(t) = CX(t) + Du(t)$$

Here,

$$X(t) = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}$$
$$A = \begin{bmatrix} 0 & 1 \\ \frac{-g}{l} & \frac{-b}{ml^2} \end{bmatrix}$$
$$B = 0$$
$$C = \begin{bmatrix} 1 & 0 \end{bmatrix}$$
$$D = 0$$

The B and D matrices are zero because there is no external driving force for the simple pendulum.

1 Create an ODE file that relates the model coefficients to its state space representation.

```
function [A,B,C,D] = LinearPendulum(m,g,l,b,Ts)
% Function mapping ODE coefficients to state-space matrices.
% Save this function as a file on your computer.
A = [0 1; -g/l, -b/m/l^2];
```

```
B = zeros(2,0);
C = [1 0];
D = zeros(1,0);
end
```

The function, `LinearPendulum`, returns the state space representation of the linear motion model of the simple pendulum using the model coefficients m, g, l and b. The sample time used is specified by Ts.

Save this function as `LinearPendulum.m`.

2 Create a linear grey-box model associated with the `LinearPendulum` function.

```
m = 1;
g = 9.81;
l = 1;
b = 0.2;
linear_model = idgrey('LinearPendulum', {m,g,l,b},'c');
```

`m`, `g` and `l` specify the values of the known model coefficients. `b` specifies the initial guess for the viscous friction coefficient.

The function `LinearPendulum` must be on the MATLAB path. Alternatively, you can specify the full path name for this function.

The `'c'` input argument in the call to `idgrey` specifies `linear_model` as a continuous-time system.

3 Specify m, g and l as known parameters.

```
linear_model.Structure.Parameters(1).Free = false;
linear_model.Structure.Parameters(2).Free = false;
linear_model.Structure.Parameters(3).Free = false;
```

As defined in the previous step, m, g, and l are the first three parameters of `linear_model`. Using the `Structure.Parameters.Free` field for each of the parameters, m, g, and l are specified as fixed values.

4 Create an estimation option set that specifies the initial state to be estimated and turns on the estimation progress display.

```
opt = greyestOptions('InitialState',...
      'estimate','Display','on');
opt.Focus = 'stability';
```

5 Estimate the viscous friction coefficient.

```
linear_model = greyest(data,linear_model,opt);
```

The `greyest` command updates the parameter of `linear_model`.

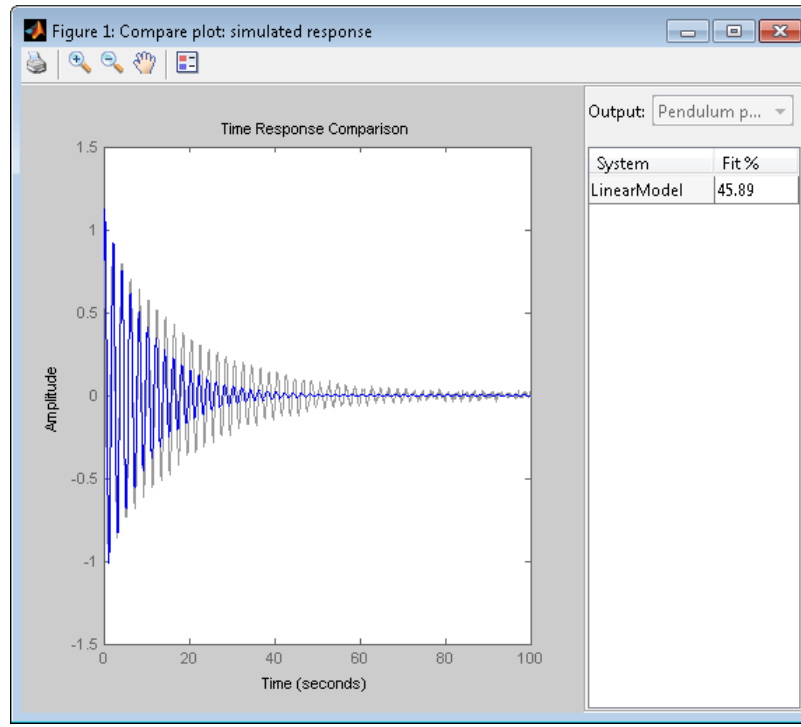
```
b_est = linear_model.Structure.Parameters(4).Value;
[linear_b_est, dlinear_b_est] = getpvec(linear_model, 'free')
```

`getpvec` returns, as `dlinear_b_est`, the 1 standard deviation uncertainty associated with `b`, the free estimation parameter of `linear_model`.

The estimated value of b, the viscous friction coefficient, using linear grey-box estimation is 0.1178 with a standard deviation of 0.0088.

6 Compare the response of the linear grey-box model to the measured data.

```
compare(data,LinearModel)
```



The linear grey-box estimation model provides a 45.9% fit to measured data. This is not surprising given that the underlying assumption of the linear pendulum motion model is that the pendulum undergoes small angular displacements, whereas the measured data shows large oscillations.

Perform nonlinear grey-box estimation.

Nonlinear grey-box estimation requires that you express the differential equation as a set of first order equations.

Using the angular displacement (θ) and the angular velocity ($\dot{\theta}$) as state variables, the equation of motion can be rewritten as a set of first order nonlinear differential equations:

$$x_1(t) = \theta(t)$$
$$x_2(t) = \dot{\theta}(t)$$
$$\dot{x}_1(t) = x_2(t)$$
$$\dot{x}_2(t) = \frac{-g}{l} sin(x_1(t)) - \frac{b}{ml^2} x_2(t)$$
$$y(t) = x_1(t)$$

1. Create an ODE file that relates the model coefficients to its nonlinear representation.

```
function [dx,y] = NonlinearPendulum(t,x,u,m,g,l,b,varargin)
% Function that maps the ODE coefficients to state
% variable derivatives and output.
% Save this function as a file on your computer.

% Output equation.
y = x(1); % Angular position.

% State equations.
dx = [x(2);                                ... % Angular position
      -(g/l)*sin(x(1))-b/(m*l^2)*x(2)      ... % Angular velocity
     ];
end
```

   The function, `NonlinearPendulum`, returns the state derivatives and output of the nonlinear motion model of the pendulum using the model coefficients m, g, l and b.

   Save this function as `NonlinearPendulum.m`.

2. Create a nonlinear grey-box model associated with the `NonlinearPendulum` function.

```
m = 1;
g = 9.81;
l = 1;
b = 0.2;
order         = [1 0 2];
parameters    = {m,g,l,b};
```

```
initial_states = [1; 0];
Ts             = 0;
nonlinear_model = idnlgrey('NonlinearPendulum', order, ...
                  parameters, initial_states, Ts);
```

3 Specify m, g and l as known parameters.

```
setpar(nonlinear_model, 'Fixed', {true true true false});
```

As defined in the previous step, m, g, and l are the first three parameters of `nonlinear_model`. Using the `setpar` command, m, g, and l are specified as fixed values and b is specified as a free estimation parameter.

4 Estimate the viscous friction coefficient.

```
nonlinear_model = pem(data,nonlinear_model,'Display','Full');
```

The `pem` command updates the parameter of `nonlinear_model`.

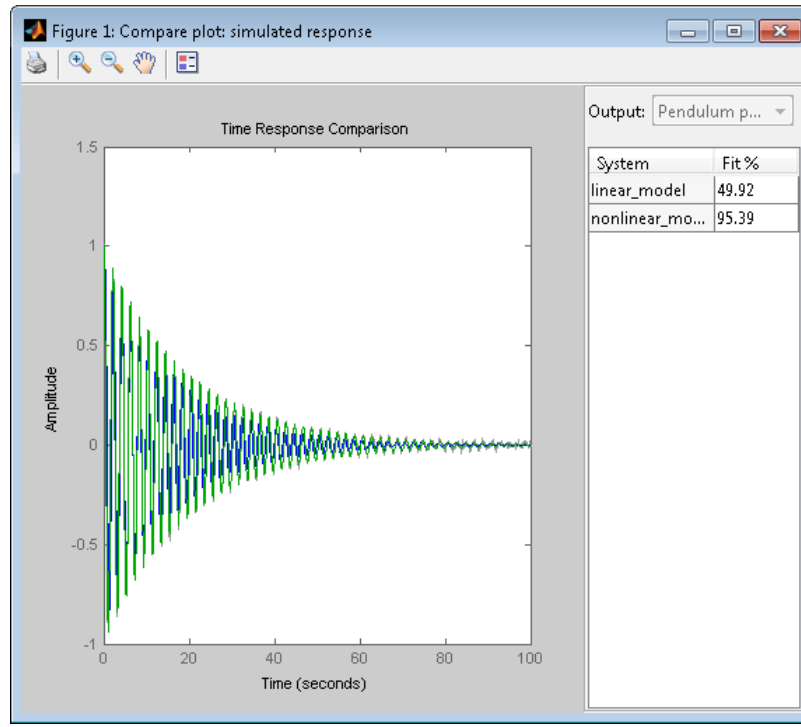
```
b_est = nonlinear_model.Parameters(4).Value;
[nonlinear_b_est, dnonlinear_b_est] = ...
                              getpvec(nonlinear_model, 'free')
```

`getpvec` returns, as `dnonlinear_b_est`, the 1 standard deviation uncertainty associated with `b`, the free estimation parameter of `nonlinear_model`.

The estimated value of b, the viscous friction coefficient, using nonlinear grey-box estimation is 0.1 with a standard deviation of 0.0149.

5 Compare the response of the linear and nonlinear grey-box models to the measured data.

```
compare(data,linear_model,nonlinear_model)
```



The nonlinear grey-box model estimation provides a closer fit (95%) to the measured data.

Estimate Model Using Zero/Pole/Gain Parameters

This example shows how to estimate a model that is parameterized by poles, zeros, and gains.

For this example, parameterize the model using complex-conjugate pole/zero pairs. When you parameterize a real, grey-box model using complex-conjugate pairs of parameters, the software updates parameter values such that the estimated values are also complex conjugate pairs

---

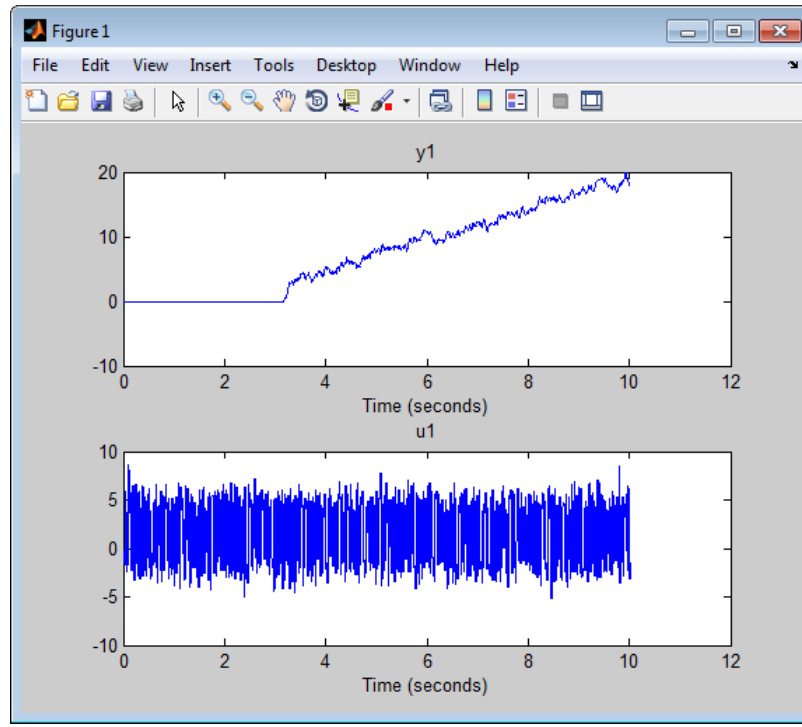
Note This example requires Control System Toolbox license.

Load the measured data.

```
load zpkestdata zd;
```

The variable `zd`, which contains measured data, is loaded into the MATLAB workspace.

```
plot(zd);
```



The output shows an input delay of approximately 3.14 seconds.

Create an ODE file that returns the state-space matrices of the estimated model.

For this example, estimate the model using the zero-pole-gain (`zpk`) form and use the shipped function `zpkestODE.m`.

```
function [a,b,c,d] = zpkestODE(z,p,k,Ts,varargin)
%zpkestODE ODE file that parameterizes a state-space model
% using poles and zeros as its parameters.
%
% Requires Control System Toolbox.

sysc = zpk(z,p,k);
if Ts==0
```

```
   [a,b,c,d] = ssdata(sysc);
else
   [a,b,c,d] = ssdata(c2d(sysc,Ts,'foh'));
end
```

To view this function, enter `edit zpkestODE.m`.

Create a linear grey-box model associated with the ODE function.

For this example, assume that the model has five poles and four zeros. Assume that two of the poles and two of the zeros are complex conjugate pairs.

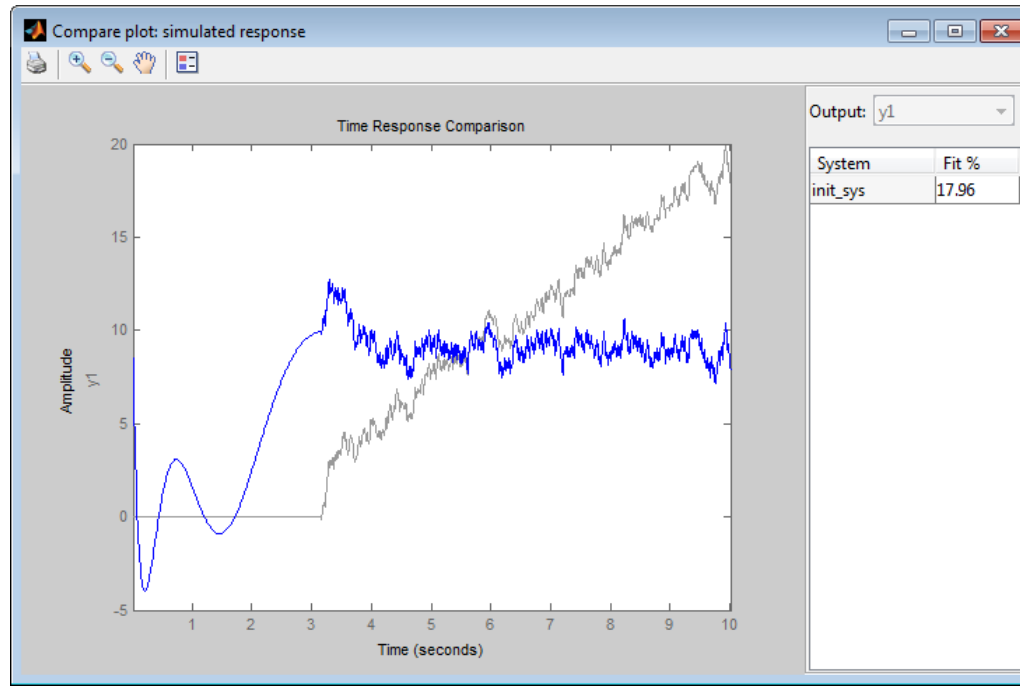
```
z = [-0.5+1i, -0.5-1i, -0.5, -1];
p = [-1.11+2i, -1.11-2i, -3.01, -4.01, -0.02];
k = 10.1;

parameters = {z,p,k};

Ts = 0;

odefun = @zpkestODE;

init_sys = idgrey(odefun,parameters,...
                  'cd',{},Ts,'InputDelay',3.14);
```

`z`, `p`, and `k` are the initial guesses for the model parameters.

`init_sys` is an `idgrey` model that is associated with the `zpkestODE.m` function. The `'cd'` flag indicates that the ODE function, `zpkestODE`, returns continuous or discrete models, depending on the sampling period.

Evaluate the quality of the fit provided by the initial model.

```
compare(zd,init_sys);
```



The initial model provides a poor fit (18%).

Specify estimation options.

```
opt = greyestOptions('InitialState','zero',...
                     'DisturbanceModel','none',...
                     'SearchMethod','gna');
```

Estimate the model.

```
sys = greyest(zd,init_sys,opt);
```

`sys`, an `idgrey` model, contains the estimated zero-pole-gain model parameters.

Compare the estimated and initial parameter values.

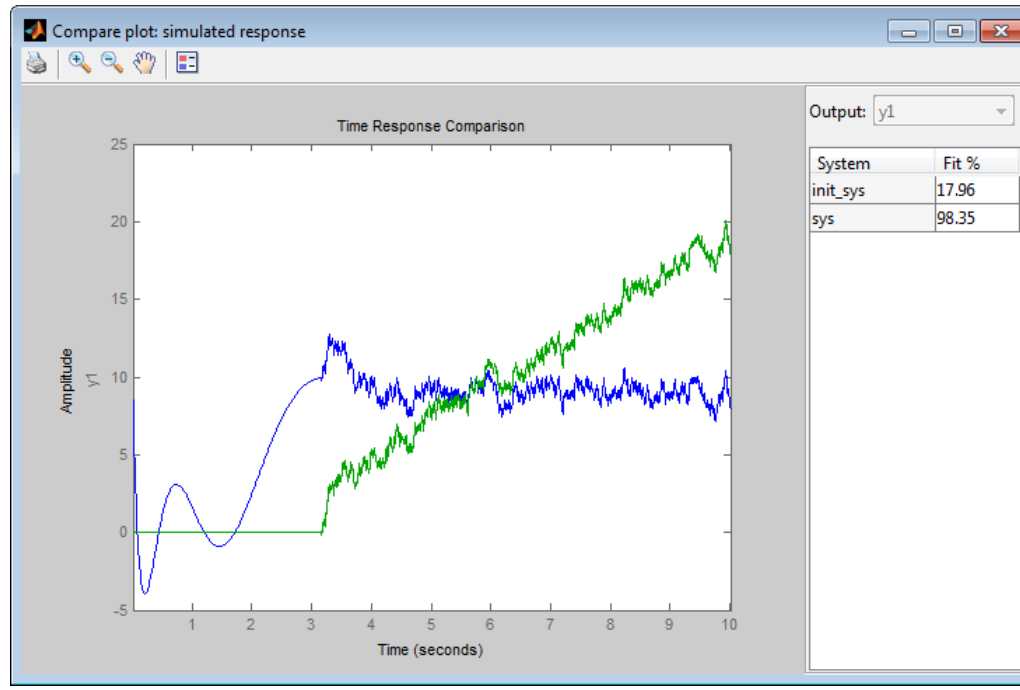
```
[getpvec(init_sys) getpvec(sys)]
```

```
  -0.5000 + 1.0000i  -1.6158 + 1.6173i
  -0.5000 - 1.0000i  -1.6158 - 1.6173i
  -0.5000            -0.9416
  -1.0000            -1.4101
  -1.1100 + 2.0000i  -2.4049 + 1.4341i
  -1.1100 - 2.0000i  -2.4049 - 1.4341i
  -3.0100            -2.3389
  -4.0100            -2.3394
  -0.0200            -0.0082
  10.1000             9.7881
```

The `getpvec` command returns the parameter values for a model. In the output above, each row displays corresponding initial and estimated parameter values. All parameters that were initially specified as complex conjugate pairs remain so after estimation.

Evaluate the quality of the fit provided by the estimated model.

```
compare(zd,init_sys,sys);
```



`sys` provides a closer fit (98.35%) to the measured data.

See Also

`idgrey` | `greyest` | `getpvec` | `ssdata` | `c2d`

Related Examples

- "Estimating Coefficients of ODEs to Fit Given Solution" on page 5-43
- "Creating IDNLGREY Model Files" on page 5-57

Creating IDNLGREY Model Files

This example shows how to write ODE files for nonlinear grey-box models as MATLAB and C MEX files.

Grey box modeling is conceptually different to black box modeling in that it involves a more comprehensive modeling step. For IDNLGREY (the nonlinear grey-box model object; the nonlinear counterpart of IDGREY), this step consists of creating an ODE file, also called a "model file". The ODE file specifes the right-hand sides of the state and the output equations typically arrived at through physical first principle modeling. In this example we will concentrate on general aspects of implementing it as a MATLAB file or a C MEX file.

IDNLGREY Model Files

IDNLGREY supports estimation of parameters and initial states in nonlinear model structures written on the following explicit state-space form (so-called output-error, OE, form, named so as the noise e(t) only affects the output of the model structure in an additive manner):

```
xn(t) = F(t, x(t), u(t), p1, ..., pNpo);         x(0) = X0;
 y(t) = H(t, x(t), u(t), p1, ..., pNpo) + e(t)
```

For discrete-time structures, xn(t) = x(T+Ts) with Ts being the sampling time, and for continuous-time structures xn(t) = d/dt x(t). In addition, F(.) and H(.) are arbitrary linear or nonlinear functions with Nx (number of states) and Ny (number of outputs) components, respectively. Any of the model parameters p1, ..., pNpo as well as the initial state vector X(0) can be estimated. Worth stressing is that

1. time-series modeling, i.e., modeling without an exogenous input signal u(t), and
2. static modeling, i.e., modeling without any states x(t)

are two special cases that are supported by IDNLGREY. (See the tutorials idnlgreydemo3 and idnlgreydemo5 for examples of these two modeling categories).

The first IDNLGREY modeling step to perform is always to implement a MATLAB or C MEX model file specifying how to update the states and

compute the outputs. More to the point, the user must write a model file, MODFILENAME.m or MODFILENAME.c, defined with the following input and output arguments (notice that this form is required for both MATLAB and C MEX type of model files)

```
[dx, y] = MODFILENAME(t, x, u, p1, p2, ..., pNpo, FileArgument)
```

MODFILENAME can here be any user chosen file name of a MATLAB or C MEX-file, e.g., see twotanks_m.m, pendulum_c.c etc. This file should be defined to return two outputs

```
dx: the right-hand side(s) of the state-space equation(s) (a column
    vector with Nx real entries; [] for static models)
 y: the right-hand side(s) of the output equation(s) (a column vector
    with Ny real entries)
```

and it should take 3+Npo(+1) input arguments specified as follows:

```
t: the current time
x: the state vector at time t ([] for static models)
u: the input vector at time t ([] for time-series models)
p1, p2, ..., pNpo: the individual parameters (which can be real
   scalars, column vectors or 2-dimensional matrices); Npo is here
   the number of parameter objects, which for models with scalar
   parameters coincide with the number of parameters Np
FileArgument: optional inputs to the model file
```

In the onward discussion we will focus on writing model using either MATLAB language or using C-MEX files. However, IDNLGREY also supports P-files (protected MATLAB files obtained using the MATLAB command "pcode") and function handles. In fact, it is not only possible to use C MEX model files but also Fortran MEX files. Consult the MATLAB documentation on External Interfaces for more information about the latter.

What kind of model file should be implemented? The answer to this question really depends on the use of the model.

Implementation using MATLAB language (resulting in a *.m file) has some distinct advantages. Firstly, one can avoid time-consuming, low-level programming and concentrate more on the modeling aspects. Secondly, any function available within MATLAB and its toolboxes can be used directly

in the model files. Thirdly, such files will be smaller and, without any modifications, all built-in MATLAB error checking will automatically be enforced. In addition, this is obtained without any code compilation.

C MEX modeling is much more involved and requires basic knowledge about the C programming language. The main advantage with C MEX model files is the improved execution speed. Our general advice is to pursue C MEX modeling when the model is going to be used many times, when large data sets are employed, and/or when the model structure contains a lot of computations. It is often worthwhile to start with using a MATLAB file and later on turn to the C MEX counterpart.

IDNLGREY Model Files Written Using MATLAB Language

With this said, let us next move on to MATLAB file modeling and use a nonlinear second order model structure, describing a two tank system, as an example. See idnlgreydemo2 for the modeling details. The contents of twotanks_m.m are as follows.

```
type twotanks_m.m
```

```
function [dx, y] = twotanks_m(t, x, u, A1, k, a1, g, A2, a2, varargin)
%TWOTANKS_M  A two tank system.

%   Copyright 2005-2006 The MathWorks, Inc.
%   $Revision: 1.1.8.1 $ $Date: 2006/11/17 13:29:14 $

% Output equation.
y = x(2);                                                % Water level, lower

% State equations.
dx = [1/A1*(k*u(1)-a1*sqrt(2*g*x(1)));                ... % Water level, upper
      1/A2*(a1*sqrt(2*g*x(1))-a2*sqrt(2*g*x(2)))      ... % Water level, lower
     ];
```

In the function header, we here find the required t, x, and u input arguments followed by the six scalar model parameters, A1, k, a1, g, A2 and a2. In the MATLAB file case, the last input argument should always be varargin to support the passing of an optional model file input argument, FileArgument.

In an IDNLGREY model object, FileArgument is stored as a cell array that might hold any kind of data. The first element of FileArgument is here accessed through varargin{1}{1}.

The variables and parameters are referred in the standard MATLAB way. The first state is x(1) and the second x(2), the input is u(1) (or just u in case it is scalar), and the scalar parameters are simply accessed through their names (A1, k, a1, g, A2 and a2). Individual elements of vector and matrix parameters are accessed as P(i) (element i of a vector parameter named P) and as P(i, j) (element at row i and column j of a matrix parameter named P), respectively.

IDNLGREY C MEX Model Files

Writing a C MEX model file is more involved than writing a MATLAB model file. To simplify this step, it is recommended that the available IDNLGREY C MEX model template is copied to MODFILENAME.c. This template contains skeleton source code as well as detailed instructions on how to customize the code for a particular application. The location of the template file is found by typing the following at the MATLAB command prompt.

```
fullfile(matlabroot, 'toolbox', 'ident', 'nlident', 'IDNLGREY_MODEL_TEMPL
```

For the two tank example, this template was copied to twotanks_c.c. After some initial modifications and configurations (described below) the state and output equations were entered, thereby resulting in the following C MEX source code.

```
type twotanks_c.c
```

```
/*   Copyright 2005-2008 The MathWorks, Inc. */
/*   $Revision: 1.1.8.4 $ $Date: 2008/04/28 03:17:22 $ */
/*   Written by Peter Lindskog. */

/* Include libraries. */
#include "mex.h"
#include <math.h>

/* Specify the number of outputs here. */
#define NY 1
```

```
/* State equations. */
void compute_dx(double *dx, double t, double *x, double *u, double **p,
                const mxArray *auxvar)
{
    /* Retrieve model parameters. */
    double *A1, *k, *a1, *g, *A2, *a2;
    A1 = p[0];   /* Upper tank area.        */
    k  = p[1];   /* Pump constant.          */
    a1 = p[2];   /* Upper tank outlet area. */
    g  = p[3];   /* Gravity constant.       */
    A2 = p[4];   /* Lower tank area.        */
    a2 = p[5];   /* Lower tank outlet area. */

    /* x[0]: Water level, upper tank. */
    /* x[1]: Water level, lower tank. */
    dx[0] = 1/A1[0]*(k[0]*u[0]-a1[0]*sqrt(2*g[0]*x[0]));
    dx[1] = 1/A2[0]*(a1[0]*sqrt(2*g[0]*x[0])-a2[0]*sqrt(2*g[0]*x[1]));
}

/* Output equation. */
void compute_y(double *y, double t, double *x, double *u, double **p,
               const mxArray *auxvar)
{
    /* y[0]: Water level, lower tank. */
    y[0] = x[1];
}



/*-------------------------------------------------------------------------- *
   DO NOT MODIFY THE CODE BELOW UNLESS YOU NEED TO PASS ADDITIONAL
   INFORMATION TO COMPUTE_DX AND COMPUTE_Y

   To add extra arguments to compute_dx and compute_y (e.g., size
   information), modify the definitions above and calls below.
 *-------------------------------------------------------------------------*/

void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
```

```
/* Declaration of input and output arguments. */
double *x, *u, **p, *dx, *y, *t;
int     i, np, nu, nx;
const mxArray *auxvar = NULL; /* Cell array of additional data. */

if (nrhs < 3) {
    mexErrMsgIdAndTxt("IDNLGREY:ODE_FILE:InvalidSyntax",
    "At least 3 inputs expected (t, u, x).");
}

/* Determine if auxiliary variables were passed as last input.  */
if ((nrhs > 3) && (mxIsCell(prhs[nrhs-1]))) {
    /* Auxiliary variables were passed as input. */
    auxvar = prhs[nrhs-1];
    np = nrhs - 4; /* Number of parameters (could be 0). */
} else {
    /* Auxiliary variables were not passed. */
    np = nrhs - 3; /* Number of parameters. */
}

/* Determine number of inputs and states. */
nx = mxGetNumberOfElements(prhs[1]); /* Number of states. */
nu = mxGetNumberOfElements(prhs[2]); /* Number of inputs. */

/* Obtain double data pointers from mxArrays. */
t = mxGetPr(prhs[0]);  /* Current time value (scalar). */
x = mxGetPr(prhs[1]);  /* States at time t. */
u = mxGetPr(prhs[2]);  /* Inputs at time t. */

p = mxCalloc(np, sizeof(double*));
for (i = 0; i < np; i++) {
    p[i] = mxGetPr(prhs[3+i]); /* Parameter arrays. */
}

/* Create matrix for the return arguments. */
plhs[0] = mxCreateDoubleMatrix(nx, 1, mxREAL);
plhs[1] = mxCreateDoubleMatrix(NY, 1, mxREAL);
dx      = mxGetPr(plhs[0]); /* State derivative values. */
y       = mxGetPr(plhs[1]); /* Output values. */
```

```
    /*
      Call the state and output update functions.

      Note: You may also pass other inputs that you might need,
      such as number of states (nx) and number of parameters (np).
      You may also omit unused inputs (such as auxvar).

      For example, you may want to use orders nx and nu, but not time (t)
      or auxiliary data (auxvar). You may write these functions as:
          compute_dx(dx, nx, nu, x, u, p);
          compute_y(y, nx, nu, x, u, p);
    */

    /* Call function for state derivative update. */
    compute_dx(dx, t[0], x, u, p, auxvar);

    /* Call function for output update. */
    compute_y(y, t[0], x, u, p, auxvar);

    /* Clean up. */
    mxFree(p);
}
```

Let us go through the contents of this file. As a first observation, we can divide the work of writing a C MEX model file into four separate sub-steps, the last one being optional:

1. Inclusion of C-libraries and definitions of the number of outputs.
2. Writing the function computing the right-hand side(s) of the state equation(s), compute_dx.
3. Writing the function computing the right-hand side(s) of the output equation(s), compute_y.
4. Optionally updating the main interface function which includes basic error checking functionality, code for creating and handling input and output arguments, and calls to compute_dx and compute_y.

Before we address these sub-steps in more detail, let us briefly comment upon a couple of general features of the C programming language.

```
A. High-precision variables (all inputs, states, outputs and
   parameters of an IDNLGREY object) should be defined to be of the
   data type "double".
B. The unary * operator placed just in front of the variable or
   parameter names is a so-called dereferencing operator. The
   C-declaration "double *A1;" specifies that A1 is a pointer to a
   double variable. The pointer construct is a concept within C that
   is not always that easy to comprehend. Fortunately, if the
   declarations of the output/input variables of compute_y and
   compute_x are not changed and all unpacked model parameters are
   internally declared with a *, then there is no need to know more
   about pointers from an IDNLGREY modeling point of view.
C. Both compute_y and compute_dx are first declared and implemented,
   where after they are called in the main interface function. In the
   declaration, the keyword "void" states explicitly that no value is
   to be returned.
```

For further details of the C programming language we refer to the book

```
B.W. Kernighan and D. Ritchie. The C Programming Language, 2nd
edition, Prentice Hall, 1988.
```

1. In the first sub-step we first include the C-libraries "mex.h" (required) and "math.h" (required for more advanced mathematics). The number of outputs is also declared per modeling file using a standard C-define:

```
/* Include libraries. */
#include "mex.h"
#include "math.h"

/* Specify the number of outputs here. */
#define NY 1
```

If desired, one may also include more C-libraries than the ones above.

The "math.h" library must be included whenever any state or output equation contains more advanced mathematics, like trigonometric and square root functions. Below is a selected list of functions included in "math.h" and the counterpart found within MATLAB:

```
C-function              MATLAB function
```

```
=======================================
sin, cos, tan           sin, cos, tan
asin, acos, atan        asin, acos, atan
sinh, cosh, tanh        sinh, cosh, tanh
exp, log, log10         exp, log, log10
pow(x, y)               x^y
sqrt                    sqrt
fabs                    abs
```

Notice that the MATLAB functions are more versatile than the corresponding C-functions, e.g., the former handle complex numbers, while the latter do not.

2-3. Next in the file we find the functions for updating the states, compute_dx, and the output, compute_y. Both these functions hold argument lists, with the output to be computed (dx or y) at position 1, after which follows all variables and parameters required to compute the right-hand side(s) of the state and the output equations, respectively.

All parameters are contained in the parameter array p. The first step in compute_dx and compute_y is to unpack and name the parameters to be used in the subsequent equations. In twotanks_c.c, compute_dx declares six parameter variables whose values are determined accordingly:

```
/* Retrieve model parameters. */
double *A1, *k, *a1, *g, *A2, *a2;
A1 = p[0];   /* Upper tank area.        */
k  = p[1];   /* Pump constant.          */
a1 = p[2];   /* Upper tank outlet area. */
g  = p[3];   /* Gravity constant.       */
A2 = p[4];   /* Lower tank area.        */
a2 = p[5];   /* Lower tank outlet area. */
```

compute_y on the other hand does not require any parameter for computing the output, and hence no model parameter is retrieved.

As is the case in C, the first element of an array is stored at position 0. Hence, dx[0] in C corresponds to dx(1) in MATLAB (or just dx in case it is a scalar), the input u[0] corresponds to u (or u(1)), the parameter A1[0] corresponds to A1, and so on.

In the example above, we are only using scalar parameters, in which case the overall number of parameters Np equals the number of parameter objects Npo. If any vector or matrix parameter is included in the model, then Npo < Np.

The scalar parameters are referenced as P[0] (P(1) or just P in a MATLAB file) and the i:th vector element as P[i-1] (P(i) in a MATLAB file). The matrices passed to a C MEX model file are different in the sense that the columns are stacked upon each other in the obvious order. Hence, if P is a 2-by-2 matrix, then P(1, 1) is referred as P[0], P(2, 1) as P[1], P(1, 2) as P[2] and P(2, 2) as P[3]. See "Tutorials on Nonlinear Grey Box Identification: An Industrial Three Degrees of Freedom Robot : C MEX-File Modeling of MIMO System Using Vector/Matrix Parameters", idnlgreydemo8, for an example where scalar, vector and matrix parameters are used.

The state and output update functions may also include other computations than just retrieving parameters and computing right-hand side expressions. For execution speed, one might, e.g., declare and use intermediate variables, whose values are used several times in the coming expressions. The robot tutorial mentioned above, idnlgreydemo8, is a good example in this respect.

compute_dx and compute_y are also able to handle an optional FileArgument. The FileArgument data is passed to these functions in the auxvar variable, so that the first component of FileArgument (a cell array) can be obtained through

```
mxArray* auxvar1 = mxGetCell(auxvar, 0);
```

Here, mxArray is a MATLAB-defined data type that enables interchange of data between the C MEX-file and MATLAB. In turn, auxvar1 may contain any data. The parsing, checking and use of auxvar1 must be handled solely within these functions, where it is up to the model file designer to implement this functionality. Let us here just refer to the MATLAB documentation on External Interfaces for more information about functions that operate on mxArrays. An example of how to use optional C MEX model file arguments is provided in idnlgreydemo6, "Tutorials on Nonlinear Grey Box Identification: A Signal Transmission System : C MEX-File Modeling Using Optional Input Arguments".

4. The main interface function should almost always have the same content and for most applications no modification whatsoever is needed. In principle,

the only part that might be considered for changes is where the calls to compute_dx and compute_y are made. For static systems, one can leave out the call to compute_dx. In other situations, it might be desired to only pass the variables and parameters referred in the state and output equations. For example, in the output equation of the two tank system, where only one state is used, one could very well shorten the input argument list to

```
void compute_y(double *y, double *x)
```

and call compute_y in the main interface function as

```
compute_y(y, x);
```

The input argument lists of compute_dx and compute_y might also be extended to include further variables inferred in the interface function. The following integer variables are computed and might therefore be passed on: nu (the number of inputs), nx (the number of states), and np (here the number of parameter objects). As an example, nx is passed to compute_y in the model investigated in the tutorial idnlgreydemo6.

The completed C MEX model file must be compiled before it can be used for IDNLGREY modeling. The compilation can readily be done from the MATLAB command line as

```
mex MODFILENAME.c
```

Notice that the mex-command must be configured before it is used for the very first time. This is also achieved from the MATLAB command line via

```
mex -setup
```

IDNLGREY Model Object

With an execution ready model file, it is straightforward to create IDNLGREY model objects for which simulations, parameter estimations, and so forth can be carried out. We exemplify this by creating two different IDNLGREY model objects for describing the two tank system, one using the model file written in MATLAB and one using the C MEX file detailed above (notice here that the C MEX model file has already been compiled).

```
Order         = [1 1 2];              % Model orders [ny nu nx].
Parameters    = [0.5; 0.003; 0.019; ...
```

```
                 9.81; 0.25; 0.016];   % Initial parameter vector.
InitialStates = [0; 0.1];              % Initial initial states.
nlgr_m    = idnlgrey('twotanks_m', Order, Parameters, InitialStates, 0)
nlgr_cmex = idnlgrey('twotanks_c', Order, Parameters, InitialStates, 0)

Continuous-time nonlinear grey-box model defined by 'twotanks_m' (MATLAB fi

   dx/dt = F(t, u(t), x(t), p1, ..., p6)
    y(t) = H(t, u(t), x(t), p1, ..., p6) + e(t)

with 1 input, 2 states, 1 output, and 6 free parameters (out of 6).
Continuous-time nonlinear grey-box model defined by 'twotanks_c' (MEX-file)

   dx/dt = F(t, u(t), x(t), p1, ..., p6)
    y(t) = H(t, u(t), x(t), p1, ..., p6) + e(t)

with 1 input, 2 states, 1 output, and 6 free parameters (out of 6).
```

Conclusions

In this tutorial we have discussed how to write IDNLGREY MATLAB and C MEX model files. We finally conclude the presentation by listing the currently available IDNLGREY model files and the tutorial/case study where they are being used. To simplify further comparisons, we list both the MATLAB (naming convention FILENAME_m.m) and the C MEX model files (naming convention FILENAME_c.c), and indicate in the tutorial column which type of modeling approach that is being employed in the tutorial or case study.

Tutorial/Case study		MATLAB file	C MEX-file
idnlgreydemo1	(MATLAB)	dcmotor_m.m	dcmotor_c.c
idnlgreydemo2	(C MEX)	twotanks_m.m	twotanks_c.c
idnlgreydemo3	(MATLAB)	preys_m.m	preys_c.c
	(C MEX)	predprey1_m.m	predprey1_c.c
	(C MEX)	predprey2_m.m	predprey2_c.c
idnlgreydemo4	(MATLAB)	narendrali_m.m	narendrali_c.c
idnlgreydemo5	(MATLAB)	friction_m.m	friction_c.c
idnlgreydemo6	(C MEX)	signaltransmission_m.m	signaltransmission_c.
idnlgreydemo7	(C MEX)	twobodies_m.m	twobodies_c.c
idnlgreydemo8	(C MEX)	robot_m.m	robot_c.c

```
idnlgreydemo9   (MATLAB)   cstr_m.m                cstr_c.c
idnlgreydemo10  (MATLAB)   pendulum_m.m            pendulum_c.c
idnlgreydemo11  (C MEX)    vehicle_m.m             vehicle_c.c
idnlgreydemo12  (C MEX)    aero_m.m                aero_c.c
idnlgreydemo13  (C MEX)    robotarm_m.m            robotarm_c.c
```

The contents of these model files can be displayed in the MATLAB command window through the command "type FILENAME_m.m" or "type FILENAME_c.c". All model files are found in the directory returned by the following MATLAB command.

```
fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'examples')
```

Additional Information

For more information on identification of dynamic systems with System Identification Toolbox™ visit the System Identification Toolbox product information page.

6

Time Series Identification

What Are Time-Series Models?

A *time series* is one or more measured output channels with no measured input. A time-series model, also called a signal model, is a dynamic system that is identified to fit a given signal or time series data. The time series can be multivariate, which leads to multivariate models.

A time series is modeled by assuming it to be the output of a system that takes a white noise signal *e(t)* of variance *NV* as its virtual input. The true measured input size of such models is zero, and their governing equation takes the form *y(t) = He(t)*, where *y(t)* is the signal being modeled and *H* is the transfer function that represents the relationship between *y(t)* and *e(t)*. The power spectrum of the time series is given by *H*(NV*Ts)*H'*, where *NV* is the noise variance and *Ts* is the model sample time.

System Identification Toolbox software provides tools for modeling and forecasting time-series data. You can estimate both linear and nonlinear black-box and grey-box models for time-series data. A linear time-series model can be a polynomial (`idpoly`) or state-space (`idss`, `idgrey`) model. Some particular types of models are parametric autoregressive (AR), autoregressive and moving average (ARMA), and autoregressive models with integrated moving average (ARIMA).

You can estimate time-series spectra using both time- and frequency-domain data. Time-series spectra describe time-series variations using cyclic components at different frequencies.

The following example illustrates a 4th order autoregressive model estimation for time series data:

```
load iddata9
sys = ar(z9,4);
```

Because the model has no measured inputs, `size(sys,2)` returns zero. The governing equation of `sys` is *A(q)y(t) = e(t)*. You can access the *A* polynomial using `sys.a` and the estimated variance of the noise *e(t)* using `sys.NoiseVariance`.

Preparing Time-Series Data

Before you can estimate models for time-series data, you must import your data into the MATLAB software. You can only use time domain data. For information about which variables you need to represent time-series data, see "Time-Series Data Representation" on page 2-10.

For more information about preparing data for modeling, see "Ways to Prepare Data for System Identification" on page 2-6.

If your data is already in the MATLAB workspace, you can import it directly into the System Identification Tool GUI. If you prefer to work at the command line, you must represent the data as a System Identification Toolbox data object instead.

In the System Identification Tool GUI. When you import scalar or multiple-output time series data into the GUI, leave the **Input** field empty. For more information about importing data, see "Importing Data into the GUI" on page 2-17.

At the command line. To represent a time series vector or a matrix `s` as an `iddata` object, use the following syntax:

```
y = iddata(s,[],Ts);
```

`s` contains as many columns as there are measured outputs and `Ts` is the sample time.

Estimating Time-Series Power Spectra

In this section...

How to Estimate Time-Series Power Spectra Using the GUI

You must have already imported your data into the GUI, as described in "Preparing Time-Series Data" on page 6-3.

To estimate time-series spectral models in the System Identification Tool GUI:

1. In the System Identification Tool GUI, select **Estimate > Spectral models** to open the Spectral Model dialog box.
2. In the **Method** list, select the spectral analysis method you want to use. For information about each method, see "Selecting the Method for Computing Spectral Models" on page 3-11.
3. Specify the frequencies at which to compute the spectral model in either of the following ways:
   - In the **Frequencies** field, enter either a vector of values, a MATLAB expression that evaluates to a vector, or a variable name of a vector in the MATLAB workspace. For example, `logspace(-1,2,500)`.
   - Use the combination of **Frequency Spacing** and **Frequencies** to construct the frequency vector of values:
     - In the **Frequency Spacing** list, select `Linear` or `Logarithmic` frequency spacing.

       Note For `etfe`, only the `Linear` option is available.

     - In the **Frequencies** field, enter the number of frequency points.

For time-domain data, the frequency ranges from 0 to the Nyquist frequency. For frequency-domain data, the frequency ranges from the smallest to the largest frequency in the data set.

4. In the **Frequency Resolution** field, enter the frequency resolution, as described in "Controlling Frequency Resolution of Spectral Models " on page 3-12. To use the default value, enter `default` or leave the field empty.

5. In the **Model Name** field, enter the name of the correlation analysis model. The model name should be unique in the Model Board.

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

7. In the Spectral Model dialog box, click **Close**.

8. To view the estimated disturbance spectrum, select the **Noise spectrum** check box in the System Identification Tool GUI. For more information about working with this plot, see "Noise Spectrum Plots" on page 8-51.

To export the model to the MATLAB workspace, drag it to the **To Workspace** rectangle in the System Identification Tool GUI. You can view the power spectrum and the confidence intervals of the resulting `idfrd` model object using the `bode` command.

How to Estimate Time-Series Power Spectra at the Command Line

You can use the `etfe`, `spa`, and `spafdr` commands to estimate power spectra of time series for both time-domain and frequency-domain data. The following table provides a brief description of each command.

You must have already prepared your data, as described in "Preparing Time-Series Data" on page 6-3.

The resulting models are stored as an `idfrd` model object, which contains `SpectrumData` and its variance. For multiple-output data, `SpectrumData` contains power spectra of each output and the cross-spectra between each output pair.

Estimating Frequency Response of Time Series

Command	Description
`etfe`	Estimates a periodogram using Fourier analysis.
`spa`	Estimates the power spectrum with its standard deviation using spectral analysis.
`spafdr`	Estimates the power spectrum with its standard deviation using a variable frequency resolution.

For example, suppose `y` is time-series data. The following commands estimate the power spectrum `g` and the periodogram `p`, and plot both models with three standard deviation confidence intervals:

```
g = spa(y);
p = etfe(y);
spectrum(g,p);
```

For detailed information about these commands, see the corresponding reference pages.

Estimating AR and ARMA Models

In this section...

Definition of AR and ARMA Models

For a single-output signal *y(t)*, the AR model is given by the following equation:

$$A(q)y(t) = e(t)$$

The AR model is a special case of the ARX model with no input.

The ARMA model for a single-output time-series is given by the following equation:

$$A(q)y(t) = C(q)e(t)$$

The ARMA structure reduces to the AR structure for *C(q)*=1. The ARMA model is a special case of the ARMAX model with no input.

For more information about polynomial models, see "What Are Polynomial Models?" on page 3-45.

For information on models containing noise integration see "Estimating ARIMA Models" on page 6-17

Estimating Polynomial Time-Series Models in the GUI

Before you begin, you must have accomplished the following:

- Prepared the data, as described in "Preparing Time-Series Data" on page 6-3

- Estimated model order, as described in "Preliminary Step – Estimating Model Orders and Input Delays" on page 3-53
- (Multiple-output AR models only) Specified the model-order matrix in the MATLAB workspace before estimation, as described in "Polynomial Sizes and Orders of Multi-Output Polynomial Models" on page 3-68

To estimate AR and ARMA models using the System Identification Tool GUI:

1. In the System Identification Tool GUI, select **Estimate > Polynomial models** to open the Polynomial and State-Space Models dialog box.
2. In the **Structure** list, select the polynomial model structure you want to estimate from the following options:
   - `AR:[na]`
   - `ARMA:[na nc]`

   This action updates the options in the Linear Parametric Models dialog box to correspond with this model structure. For information about each model structure, see "Definition of AR and ARMA Models" on page 6-7.

Note OE and BJ structures are not available for time-series models.

---

3. In the **Orders** field, specify the model orders, as follows:
   - **For single-output models.** Enter the model orders according to the sequence displayed in the **Structure** field.
   - **For multiple-output ARX models.** Enter the model orders directly, as described in "Polynomial Sizes and Orders of Multi-Output Polynomial Models" on page 3-68. Alternatively, enter the name of the matrix `NA` in the MATLAB Workspace browser that stores model orders, which is `Ny`-by-`Ny`.

Tip To enter model orders and delays using the Order Editor dialog box, click **Order Editor**.

---

4 (AR models only) Select the estimation **Method** as **ARX** or **IV** (instrumental variable method). For more information about these methods, see "Polynomial Model Estimation Algorithms" on page 3-73.

Note **IV** is not available for multiple-output data.

---

5 Select the **Add noise integration** check box if you want to include an integrator in noise source $e(t)$. This selection changes an AR model into

an ARI model ($Ay = \frac{e}{1-q^{-1}}$) and an ARMA model into an ARIMA model

($Ay = \frac{C}{1-q^{-1}}e(t)$).

6 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.

7 In the **Initial state** list, specify how you want the algorithm to treat initial states. For more information about the available options, see "Specifying Initial States for Iterative Estimation Algorithms" on page 3-73.

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 might reduce computation time for complex models and large data sets.

9 To view the estimation progress at the command line, select the **Display progress** check box. During estimation, the following information is displayed for each iteration:

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

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

11 (Prediction-error method only) To stop the search and save the results after the current iteration has been completed, click **Stop Iterations**. To continue iterations from the current model, click the **Continue iter** button to assign current parameter values as initial guesses for the next search and start a new search. For the multi-output case, you can stop iterations for each output separately. Note that the software runs independent searches for each output.

12 To plot the model, select the appropriate check box in the **Model Views** area of the System Identification Tool GUI.

You can export the model to the MATLAB workspace for further analysis by dragging it to the **To Workspace** rectangle in the System Identification Tool GUI.

Estimating AR and ARMA Models at the Command Line

You can estimate AR and ARMA models at the command line. The estimated models are represented by `idpoly` model objects. For more information about models objects, see “What Are Model Objects?” on page 1-3.

The following table summarizes the commands and specifies whether single-output or multiple-output models are supported.

Commands for Estimating Polynomial Time-Series Models

Method Name	Description
`ar`	Noniterative, least-squares method to estimate linear, discrete-time single-output AR models.
`armax`	Iterative prediction-error method to estimate linear ARMAX models.
`arx`	Noniterative, least-squares method for estimating linear AR models.
`ivar`	Noniterative, instrumental variable method for estimating single-output AR models.

The following code shows usage examples for estimating AR models:

```
% For scalar signals
m = ar(y,na)
% For multiple-output vector signals
m = arx(y,na)
% Instrumental variable method
m = ivar(y,na)
% For ARMA, do not need to specify nb and nk
th = armax(y,[na nc])
```

The `ar` command provides additional options to let you choose the algorithm for computing the least-squares from a group of several popular techniques from the following methods:

- Burg (geometric lattice)
- Yule-Walker
- Covariance

Estimating State-Space Time-Series Models

In this section...

Definition of State-Space Time-Series Model

The discrete-time state-space model for a time series is given by the following equations:

$$x(kT + T) = Ax(kT) + Ke(kT)$$
$$y(kT) = Cx(kT) + e(kT)$$

where T is the sampling interval and $y(kT)$ is the output at time instant kT.

The time-series structure corresponds to the general structure with empty B and D matrices.

For information about general discrete-time and continuous-time structures for state-space models, see "What Are State-Space Models?" on page 3-79.

Estimating State-Space Models at the Command Line

You can estimate single-output and multiple-output state-space models at the command line for time-domain data (`iddata` object).

The following table provides a brief description of each command. The resulting models are `idss` model objects. You can estimate either continuous-time, or discrete-time models using these commands.

Commands for Estimating State-Space Time-Series Models

Command	Description
`n4sid`	Noniterative subspace method for estimating linear state-space models.
`ssest`	Estimates linear time-series models using an iterative estimation method that minimizes the prediction error.

Identify Time-Series Models at Command Line

This example shows how to simulate a time-series model, compare the spectral estimates, estimate covariance, and predict output of the model.

Generate time-series data.

```
ts0 = idpoly([1 -1.5 0.7],[]);
e = idinput(200,'rgs');
% Define y vector
y = sim(ts0,e);
% iddata object with sampling time 1
y = iddata(y);
plot(y);
```

Compare the spectral estimates.

```
% Estimate periodogram and spectrum
per = etfe(y);
speh = spa(y);
spectrum(per,speh,ts0);
% Estimate a second-order AR model
ts2 = ar(y,2);
spectrum(speh,ts2,ts0);
```

Define the true covariance function.

```
ir = sim(ts0,[1;zeros(24,1)]);
Ry0 = conv(ir,ir(25:-1:1));
ir2 = sim(ts2,[1;zeros(24,1)]);
Ry2 = conv(ir2,ir2(25:-1:1));
```

Estimate covariance.

```
z = [y.y ; zeros(25,1)];
j=1:200;
Ryh = zeros(25,1);
for k=1:25,
a = z(j,:)'*z(j+k-1,:);
Ryh(k) = Ryh(k)+conj(a(:));
end
```

```
Ryh = Ryh/200; % biased estimate
Ryh = [Ryh(end:-1:2); Ryh];
```

Alternatively, you can use the Signal Processing Toolbox command `xcorr`.

```
Ryh = xcorr(y.y, 24 ,'biased');
```

Plot and compare the covariance.

```
plot([-24:24]'*ones(1,3),[Ryh,Ry2,Ry0]);
```

Predict model output.

```
compare(y,ts2,5);
```

Estimating Nonlinear Models for Time-Series Data

When a linear model provides an insufficient description of the dynamics, you can try estimating a nonlinear models. To learn more about when to estimate nonlinear models, see "Building Models from Data" in the Getting Started Guide.

Before you can estimate models for time-series data, you must have already prepared the data as described in "Preparing Time-Series Data" on page 6-3.

For black-box modeling of time-series data, the toolbox supports nonlinear ARX models. To learn how to estimate this type of model, see "Identifying Nonlinear ARX Models" on page 4-8.

If you understand the underlying physics of the system, you can specify an ordinary differential or difference equation and estimate the coefficients. To learn how to estimate this type of model, see "Estimating Nonlinear Grey-Box Models" on page 5-17. See also "Estimating Coefficients of ODEs to Fit Given Solution" on page 5-43 for an example of time series modeling using the grey-box approach.

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

Estimating ARIMA Models

Models of time series containing non-stationary trends (seasonality) are sometimes required. One category of such models are the "Autoregressive Integrated Moving Average" or ARIMA models. These models contain a fixed integrator in the noise source. Thus if the governing equation of an ARMA model is expressed as *A(q)y(t)=Ce(t)*, where *A(q)* represents the auto-regressive term and *C(q)* the moving average term, the corresponding model of an ARIMA model would be expressed as

$$A(q)y(t) = \frac{C(q)}{(1-q^{-1})}e(t)$$

where the term $\frac{1}{1-q^{-1}}$ represents the discrete-time integrator. Similarly, you can formulate the equations for ARI and ARIX models.

Using time series model estimation commands `ar`, `arx` and `armax` you can now introduce integrators into the noise source *e(t)*. You do this by using the `IntegrateNoise` parameter in the estimation command.

Note The estimation approach does not account any constant offsets in the time series data. The ability to introduce noise integrator is not limited to time series data alone. You can do so also for input-output models where the disturbances might be subject to seasonality. One example is the polynomial models of ARIMAX structure:

$$A(q)y(t) = B(q)u(t) + \frac{C(q)}{(1-q^{-1})}e(t)$$

See the `armax` reference page examples.

Estimate an ARI model for a scalar time series with linear trend.

```
load iddata9 z9
Ts = z9.Ts;
y = cumsum(z9.y);
model = ar(y,4,'ls','Ts',Ts,'IntegrateNoise', true)
```

```
compare(y,model,5) % 5 step ahead prediction
```

Estimate a multivariate time series model such that the noise integration is present in only one of the two time series.

```
load iddata9 z9
Ts = z9.Ts;
y = z9.y;
y2 = cumsum(y);
% artificially construct a bivariate time series
data = iddata([y, y2],[],Ts); na = [4 0; 0 4];
nc = [2;1];
model1 = armax(data, [na nc], 'IntegrateNoise',[false; true])
% Forecast the time series 100 steps into future
yf = forecast(model1,data(1:100), 100);
plot(data(1:100),yf)
```

If the outputs were coupled (`na` was not a diagonal matrix), the situation will be more complex and simply adding an integrator to the second noise channel will not work.

Analyzing of Time-Series Models

A time-series model has no inputs. However, you can use many response computation commands on such models. The software treats (implicitly) the noise source *e(t)* as a measured input. Thus, `step(sys)` plots the step response assuming that the step input was applied to the noise channel *e(t)*.

To avoid ambiguity in how the software treats a time-series model, you can transform it explicitly into an input-output model using `noise2meas`. This command causes the noise input *e(t)* to be treated as a measured input and transforms the linear time series model with *Ny* outputs into an input-output model with *Ny* outputs and *Ny* inputs. You can use the resulting model with commands, such as, `bode`, `nyquist`, and `iopzmap` to study the characteristics of the *H* transfer function. For example:

```
iosys = noise2meas(sys);
% step response of H if the step command was applied
% to the noise source e(t)
step(iosys)
% poles and zeros of H
iopzmap(iosys)
```

You can calculate and plot the time-series spectrum directly (without conversion using `noise2meas`) using `spectrum`. For example:

```
spectrum(sys)
```

plots the time-series spectrum amplitude:

$$\Phi(\omega) = \left\| H(\omega) \right\|^2$$

7

Recursive Model Identification

What Is Recursive Estimation?

Many real-world applications, such as adaptive control, adaptive filtering, and adaptive prediction, require a model of the system to be available online while the system is in operation. Estimating models for batches of input-output data is useful for addressing the following types of questions regarding system operation:

- Which input should be applied at the next sampling instant?
- How should the parameters of a matched filter be tuned?
- What are the predictions of the next few outputs?
- Has a failure occurred? If so, what type of failure?

You might also use online models to investigate time variations in system and signal properties.

The methods for computing online models are called *recursive identification methods*. Recursive algorithms are also called *recursive parameter estimation*, *adaptive parameter estimation*, *sequential estimation*, and *online* algorithms.

For more information, see "Recursive Estimation and Data Segmentation Techniques in System Identification Toolbox™" on page 7-12. For detailed information about recursive parameter estimation algorithms, see the corresponding chapter in *System Identification: Theory for the User* by Lennart Ljung (Prentice Hall PTR, Upper Saddle River, NJ, 1999).

You can recursively estimate linear polynomial models, such as ARX, ARMAX, Box-Jenkins, and Output-Error models. For time-series data containing no inputs and a single output, you can estimate AR (autoregressive) and ARMA (autoregressive and moving average) single-output models.

Data Supported for Recursive Estimation

To recursively estimate linear models, data must be in one of the following formats:

- Matrix of the form `[y u]`, where `y` represents the output data using one or more column vectors. `u` represents the input data using one or more column vectors.
- `iddata` object.

Related Examples

- "Recursive Estimation and Data Segmentation Techniques in System Identification Toolbox™" on page 7-12

Concepts

- "What Is Recursive Estimation?" on page 7-2
- "Algorithms for Recursive Estimation" on page 7-4
- "Preliminary Step – Estimating Model Orders and Input Delays" on page 3-53
- "Polynomial Sizes and Orders of Multi-Output Polynomial Models" on page 3-68

Algorithms for Recursive Estimation

In this section...

Types of Recursive Estimation Algorithms

You can choose from the following four recursive estimation algorithms:

- "General Form of Recursive Estimation Algorithm" on page 7-4
- "Kalman Filter Algorithm" on page 7-6
- "Forgetting Factor Algorithm" on page 7-8
- "Unnormalized and Normalized Gradient Algorithms" on page 7-9

You specify the type of recursive estimation algorithms as arguments in the recursive estimation commands.

For detailed information about these algorithms, see the corresponding chapter in *System Identification: Theory for the User* by Lennart Ljung (Prentice Hall PTR, Upper Saddle River, NJ, 1999).

General Form of Recursive Estimation Algorithm

The general recursive identification algorithm is given by the following equation:

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\left(y(t) - \hat{y}(t)\right)$$

$\hat{\theta}(t)$ is the parameter estimate at time *t*. *y(t)* is the observed output at time *t* and $\hat{y}(t)$ is the prediction of *y(t)* based on observations up to time *t-1*. The

gain, *K(t)*, determines how much the current prediction error $y(t)-\hat{y}(t)$ affects the update of the parameter estimate. The estimation algorithms minimize the prediction-error term $y(t)-\hat{y}(t)$.

The gain has the following general form:

$$K(t)=Q(t)\psi(t)$$

The recursive algorithms supported by the System Identification Toolbox product differ based on different approaches for choosing the form of *Q(t)* and computing $\psi(t)$, where $\psi(t)$ represents the gradient of the predicted model output $\hat{y}(t\,|\,\theta)$ with respect to the parameters θ.

The simplest way to visualize the role of the gradient $\psi(t)$ of the parameters, is to consider models with a linear-regression form:

$$y(t)=\psi^T(t)\theta_0(t)+e(t)$$

In this equation, $\psi(t)$ is the *regression vector* that is computed based on previous values of measured inputs and outputs. $\theta_0(t)$ represents the true parameters. *e(t)* is the noise source (*innovations*), which is assumed to be white noise. The specific form of $\psi(t)$ depends on the structure of the polynomial model.

For linear regression equations, the predicted output is given by the following equation:

$$\hat{y}(t)=\psi^T(t)\hat{\theta}(t-1)$$

For models that do not have the linear regression form, it is not possible to compute exactly the predicted output and the gradient $\psi(t)$ for the current parameter estimate $\hat{\theta}(t-1)$. To learn how you can compute approximation for

$\psi(t)$ and $\hat{\theta}(t-1)$ for general model structures, see the section on recursive prediction-error methods in *System Identification: Theory for the User* by Lennart Ljung (Prentice Hall PTR, Upper Saddle River, NJ, 1999).

Kalman Filter Algorithm

Mathematics of the Kalman Filter Algorithm

The following set of equations summarizes the *Kalman filter* adaptation algorithm:

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\left(y(t) - \hat{y}(t)\right)$$

$$\hat{y}(t) = \psi^T(t)\hat{\theta}(t-1)$$

$$K(t) = Q(t)\psi(t)$$

$$Q(t) = \frac{P(t-1)}{R_2 + \psi(t)^T P(t-1)\psi(t)}$$

$$P(t) = P(t-1) + R_1 - \frac{P(t-1)\psi(t)\psi(t)^T P(t-1)}{R_2 + \psi(t)^T P(t-1)\psi(t)}$$

This formulation assumes the linear-regression form of the model:

$$y(t) = \psi^T(t)\theta_0(t) + e(t)$$

The Kalman filter is used to obtain *Q(t)*.

This formulation also assumes that the true parameters $\theta_0(t)$ are described by a random walk:

$$\theta_0(t) = \theta_0(t-1) + w(t)$$

w(t) is Gaussian white noise with the following covariance matrix, or *drift matrix* R_1:

$$Ew(t)w(t)^T = R_1$$

R_2 is the variance of the innovations *e(t)* in the following equation:

$$y(t) = \psi^T(t)\theta_0(t) + e(t)$$

The Kalman filter algorithm is entirely specified by the sequence of data *y(t)*, the gradient $\psi(t)$, R_1, R_2, and the initial conditions $\theta(t=0)$ (initial guess of the parameters) and $P(t=0)$ (covariance matrix that indicates parameters errors).

Note To simplify the inputs, you can scale R_1, R_2, and $P(t=0)$ of the original problem by the same value such that R_2 is equal to 1. This scaling does not affect the parameters estimates.

Using the Kalman Filter Algorithm

The general syntax for the command described in "Algorithms for Recursive Estimation" on page 7-4 is the following:

```
[params,y_hat]=command(data,nn,adm,adg)
```

To specify the Kalman filter algorithm, set `adm` to `'kf'` and `adg` to the value of the drift matrix R_1 (described in "Mathematics of the Kalman Filter Algorithm" on page 7-6).

Forgetting Factor Algorithm

Mathematics of the Forgetting Factor Algorithm

The following set of equations summarizes the *forgetting factor* adaptation algorithm:

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\left(y(t) - \hat{y}(t)\right)$$

$$\hat{y}(t) = \psi^T(t)\hat{\theta}(t-1)$$

$$K(t) = Q(t)\psi(t)$$

$$Q(t) = P(t) = \frac{P(t-1)}{\lambda + \psi(t)^T P(t-1)\psi(t)}$$

$$P(t) = \frac{1}{\lambda}\left(P(t-1) - \frac{P(t-1)\psi(t)\psi(t)^T P(t-1)}{\lambda + \psi(t)^T P(t-1)\psi(t)} \right)$$

To obtain *Q(t)*, the following function is minimized at time *t*:

$$\sum_{k=1}^{t} \lambda^{t-k} e^2(k)$$

This approach discounts old measurements exponentially such that an observation that is τ samples old carries a weight that is equal to λ^{τ} times the weight of the most recent observation. $\tau = 1/(1-\lambda)$ represents the *memory horizon* of this algorithm. Measurements older than $\tau = 1/(1-\lambda)$ typically carry a weight that is less than about 0.3.

λ is called the forgetting factor and typically has a positive value between `0.97` and `0.995`.

Note In the linear regression case, the forgetting factor algorithm is known as the *recursive least-squares* (RLS) algorithm. The forgetting factor algorithm for $\lambda = 1$ is equivalent to the Kalman filter algorithm with R_1=0 and R_2=1. For more information about the Kalman filter algorithm, see "Kalman Filter Algorithm" on page 7-6.

Using the Forgetting Factor Algorithm

The general syntax for the command described in "Algorithms for Recursive Estimation" on page 7-4 is the following:

```
[params,y_hat]=command(data,nn,adm,adg)
```

To specify the forgetting factor algorithm, set `adm` to `'ff'` and `adg` to the value of the forgetting factor λ (described in "Mathematics of the Forgetting Factor Algorithm" on page 7-8).

Tip λ typically has a positive value from `0.97` to `0.995`.

Unnormalized and Normalized Gradient Algorithms

Mathematics of the Unnormalized and Normalized Gradient Algorithm

In the linear regression case, the gradient methods are also known as the *least mean squares* (LMS) methods.

The following set of equations summarizes the *unnormalized gradient* and *normalized gradient* adaptation algorithm:

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\left(y(t) - \hat{y}(t)\right)$$

$$\hat{y}(t) = \psi^T(t)\hat{\theta}(t-1)$$

$$K(t) = Q(t)\psi(t)$$

In the unnormalized gradient approach, *Q(t)* is the product of the gain γ and the identity matrix:

$$Q(t) = \gamma I$$

In the normalized gradient approach, *Q(t)* is the product of the gain γ, and the identity matrix is normalized by the magnitude of the gradient $\psi(t)$:

$$Q(t) = \frac{\gamma}{\left|\psi(t)\right|^2} I$$

These choices of *Q(t)* update the parameters in the negative gradient direction, where the gradient is computed with respect to the parameters.

Using the Unnormalized and Normalized Gradient Algorithms

The general syntax for the command described in "Algorithms for Recursive Estimation" on page 7-4 is the following:

```
[params,y_hat]=command(data,nn,adm,adg)
```

To specify the unnormalized gain algorithm, set `adm` to `'ug'` and `adg` to the value of the gain γ (described in "Mathematics of the Unnormalized and Normalized Gradient Algorithm" on page 7-9).

To specify the normalized gain algorithm, set `adm` to `'ng'` and `adg` to the value of the gain γ.

Data Segmentation

For systems that exhibit abrupt changes while the data is being collected, you might want to develop models for separate data segments such that the system does not change during a particular data segment. Such modeling requires identification of the time instants when the changes occur in the system, breaking up the data into segments according to these time instants, and identification of models for the different data segments.

The following cases are typical applications for *data segmentation*:

- Segmentation of speech signals, where each data segment corresponds to a phonem.
- Detection of trend breaks in time series.
- Failure detection, where the data segments correspond to operation with and without failure.
- Estimating different working modes of a system.

Use `segment` to build polynomial models, such as ARX, ARMAX, AR, and ARMA, so that the model parameters are piece-wise constant over time. For detailed information about this command, see the corresponding reference page.

To see an example of using data segmentation, run the Recursive Estimation and Data Segmentation demonstration by typing to the following command at the prompt:

```
iddemo5
```

Recursive Estimation and Data Segmentation Techniques in System Identification Toolbox™

This example shows the use of recursive ("online") algorithms available in the System Identification Toolbox™. It also describes the data segmentation scheme SEGMENT which is an alternative to recursive or adaptive estimation schemes for capturing time varying behavior. This utility is useful for capturing abrupt changes in the system because of a failure or change of operating conditions.

Introduction

The recursive estimation functions include RPEM, RPLR, RARMAX, RARX, ROE, and RBJ. These algorithms implement all the recursive algorithms described in Chapter 11 of Ljung(1987).

RPEM is the general Recursive Prediction Error algorithm for arbitrary multiple-input-single-output models (the same models as PEM works for).

PRLR is the general Recursive PseudoLinear Regression method for the same family of models.

RARX is a more efficient version of RPEM (and RPLR) for the ARX-case.

ROE, RARMAX and RBJ are more efficient versions of RPEM for the OE, ARMAX, and BJ cases (compare these functions to the off-line methods).

Adaptation Schemes

Each one of the algorithms implement the four most common adaptation principles:

KALMAN FILTER approach: The true parameters are supposed to vary like a random walk with incremental covariance matrix R1.

FORGETTING FACTOR approach: Old measurements are discounted exponentially. The base of the decay is the forgetting factor lambda.

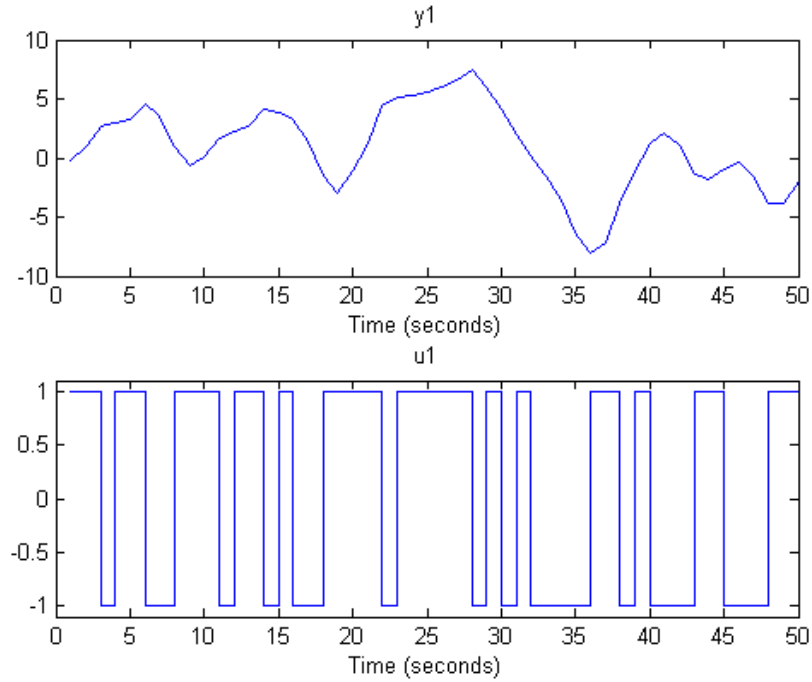
GRADIENT method: The update step is taken as a gradient step of length gamma (`th_new = th_old + gamma*psi*epsilon`).

NORMALIZED GRADIENT method: As above, but `gamma` is replaced by `gamma/(psi'*psi)`. The Gradient methods are also known as LMS (least mean squares) for the ARX case.

Analysis Data

In order to illustrate some of these schemes, let us pick a model and generate some input-output data:

```
u = sign(randn(50,1)); % input
e = 0.2*randn(50,1);   % noise
th0 = idpoly([1 -1.5 0.7],[0 1 0.5],[1 -1 0.2]); % a low order idpoly model
opt = simOptions('AddNoise',true,'NoiseData',e);
y = sim(th0,u,opt);
z = iddata(y,u);
plot(z) % analysis data object
```



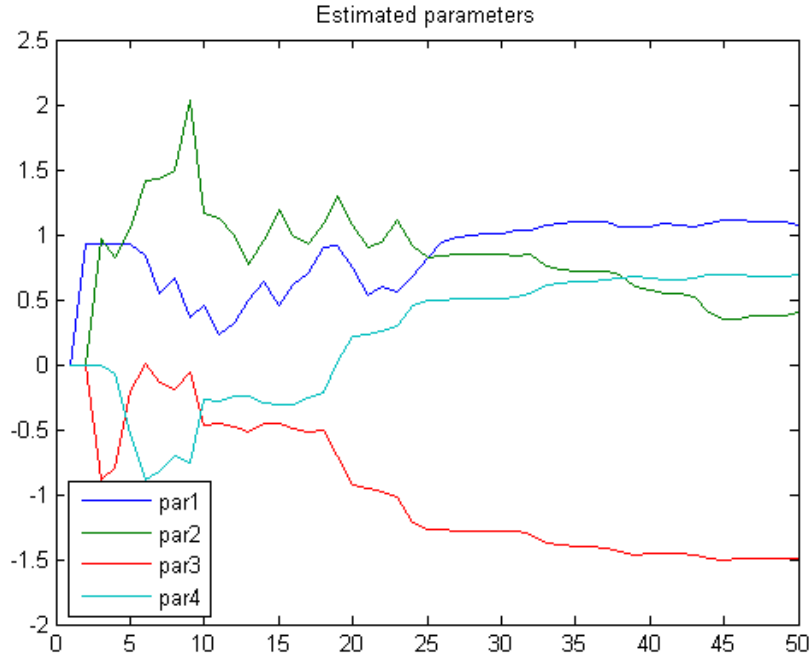
Output Error Model Estimation Using ROE

First we build an Output-Error model of the data we just plotted. Use a second order model with one delay, and apply the forgetting factor algorithm with lambda = 0.98:

```
thm1 = roe(z,[2 2 1],'ff',0.98);
```

The four parameters can now be plotted as functions of time.

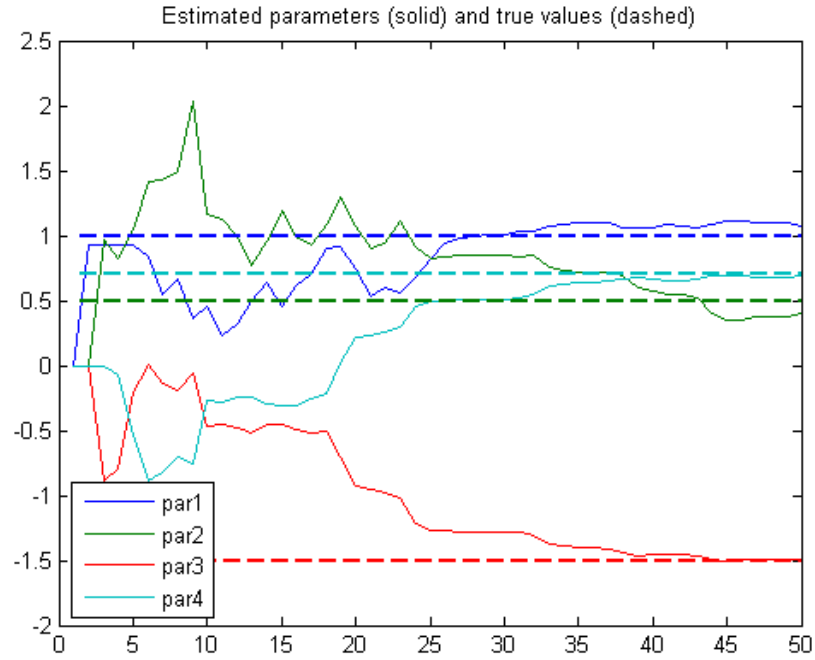
```
plot(thm1), title('Estimated parameters')
legend('par1','par2','par3','par4','location','southwest')
```



The true values are as follows:

```
hold on, plot(ones(50,1)*[1 0.5 -1.5 0.7],'--','linewidth',2),
title('Estimated parameters (solid) and true values (dashed)')
```

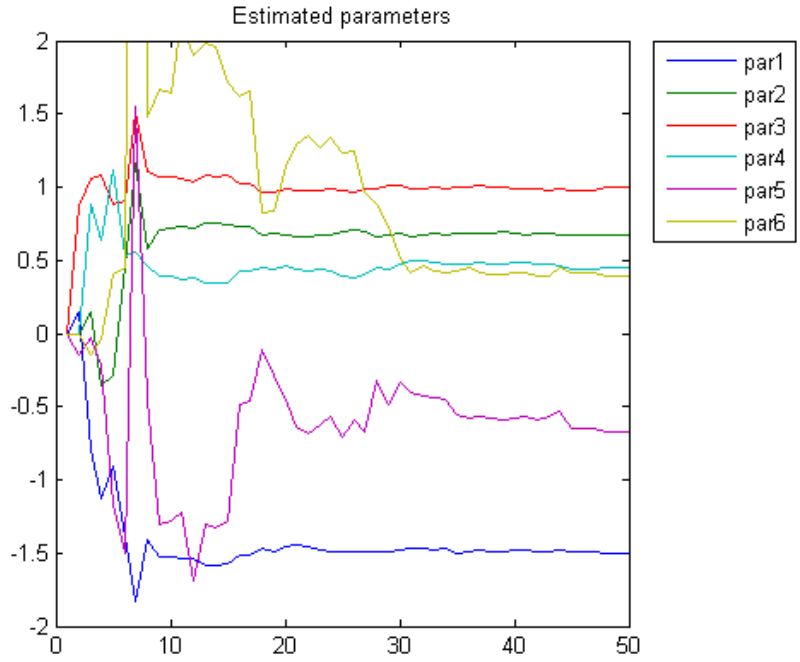
```
hold off
```



ARMAX Model Estimation USING RPLR

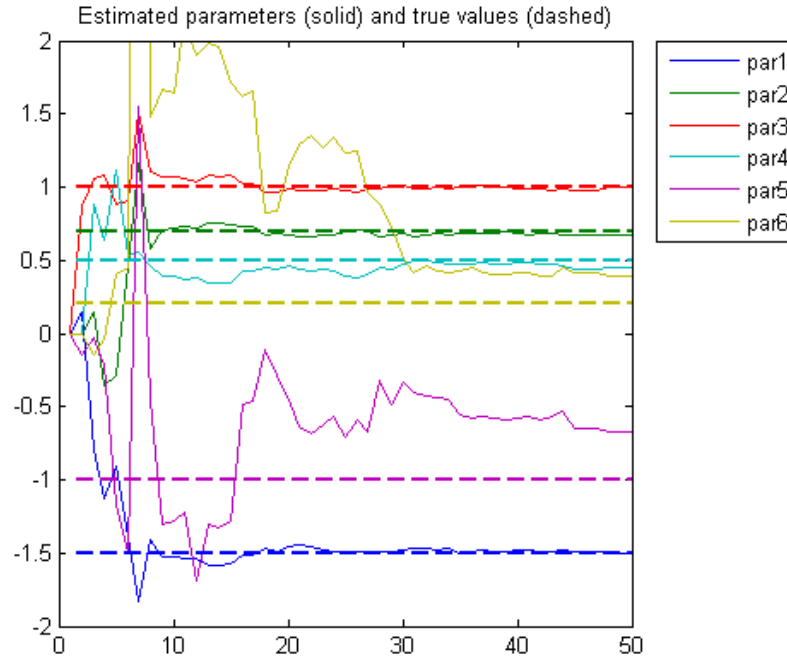
Now let us try a second order ARMAX model, using the RPLR approach (i.e. ELS) with Kalman filter adaptation, assuming a parameter variance of 0.001:

```
thm2 = rplr(z,[2 2 2 0 0 1],'kf',0.001*eye(6));
plot(thm2), title('Estimated parameters')
legend('par1','par2','par3','par4','par5','par6','location','bestoutside')
axis([0 50 -2 2])
```



The true values are as follows:

```
hold on, plot(ones(50,1)*[-1.5 0.7 1 0.5 -1 0.2],'--','linewidth',2)
title('Estimated parameters and true values')
title('Estimated parameters (solid) and true values (dashed)')
hold off
```



So far we have assumed that all data are available at once. We are thus studying the variability of the system rather than doing real on-line calculations. The algorithms are also prepared for such applications, but they must then store more update information. The conceptual update then becomes:

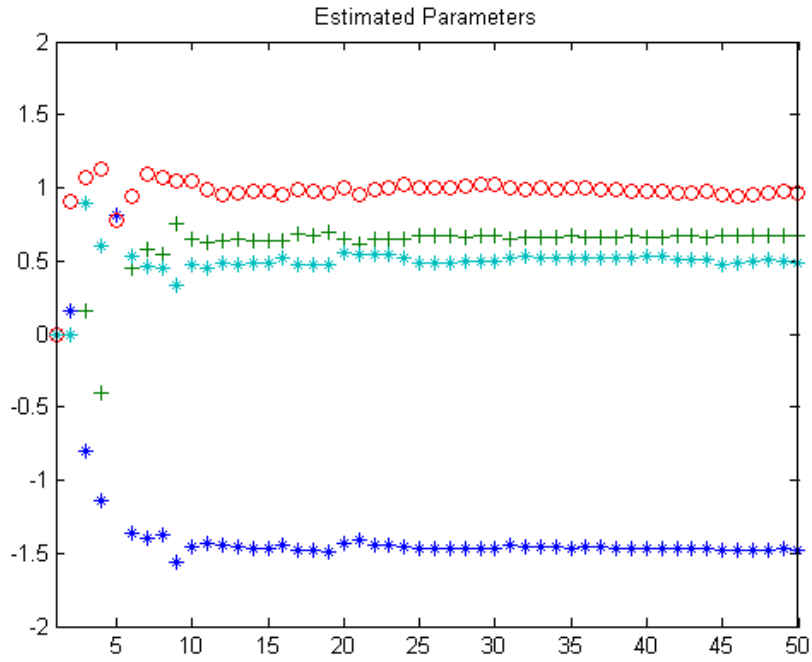
1. Wait for measurements y and u. 2. Update: `[th,yh,p,phi] = rarx([y u],[na nb nk],'ff',0.98,th',p,phi);` 3. Use th for whatever on-line application required. 4. Go to 1.

Thus the previous estimate `th` is fed back into the algorithm along with the previous value of the "P-matrix" and the data vector `phi`.

We now do an example of this where we plot just the current value of `th`.

```
[th,yh,p,phi] = rarx(z(1,:),[2 2 1],'ff',0.98);
```

```
plot(1,th(1),'*',1,th(2),'+',1,th(3),'o',1,th(4),'*'),
axis([1 50 -2 2]),title('Estimated Parameters'),drawnow
hold on;
for kkk = 2:50
   [th,yh,p,phi] = rarx(z(kkk,:),[2 2 1],'ff',0.98,th',p,phi);
   plot(kkk,th(1),'*',kkk,th(2),'+',kkk,th(3),'o',kkk,th(4),'*')
end
hold off
```



Data Segmentation as an Alternative To Recursive Estimation Schemes

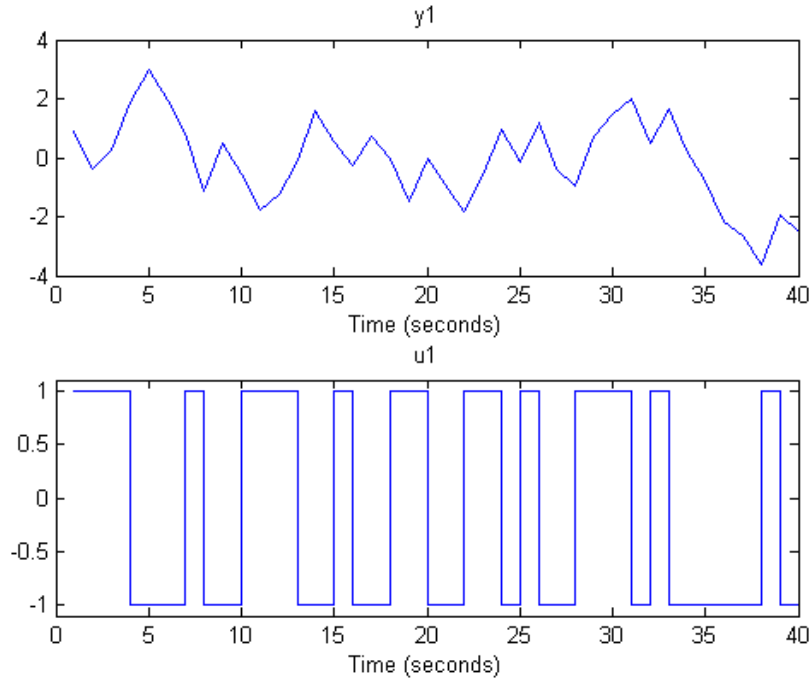
The command SEGMENT segments data that are generated from systems that may undergo abrupt changes. Typical applications for data segmentation are segmentation of speech signals (each segment corresponds to a phonem), failure detection (the segments correspond to operation with and without

failures) and estimating different working modes of a system. We shall study a system whose time delay changes from two to one.

```
load iddemo6m.mat
z = iddata(z(:,1),z(:,2));
```

First, take a look at the data:

```
plot(z)
```



The change takes place at sample number 20, but this is not so easy to see. We would like to estimate the system as an ARX-structure model with one a-parameter, two b-parameters and one delay:

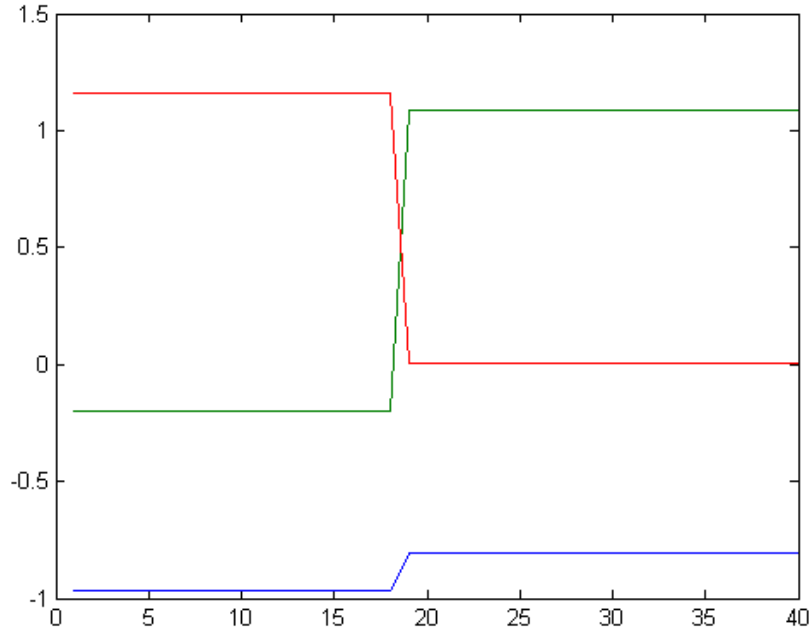
```
y(t) + a*y(t-1) = b1*u(t-1) + b2*u(t-2)
```

The three pieces of information to be given are: the data, the model orders, and a guess of the variance (r2) of the noise that affects the system. If the variance is entirely unknown, it can be estimated automatically. Here we set it to 0.1:

```
nn = [1 2 1];
[seg,v,tvmod] = segment(z,nn,0.1);
```

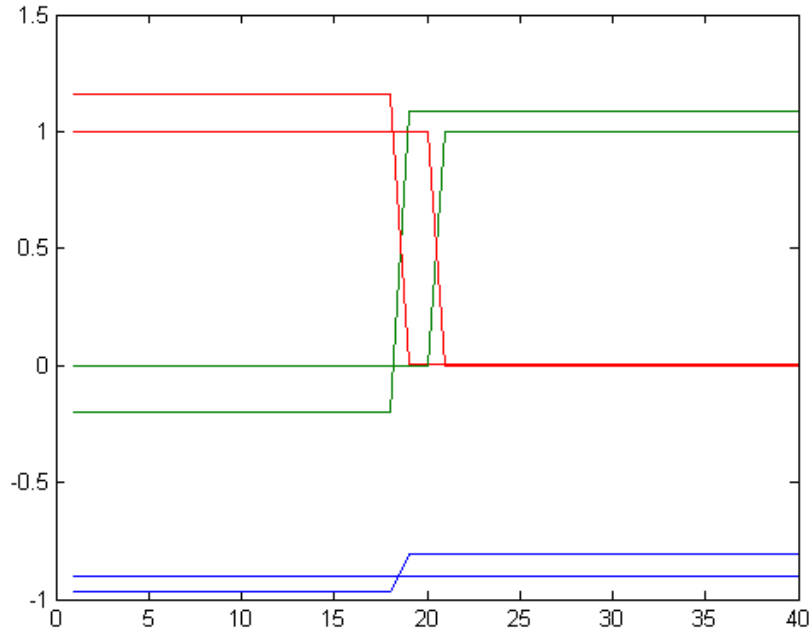
Let's take a look at the segmented model. On light-colored axes, the lines for the parameters `a`, `b1` and `b2` appear in blue, green, and red colors, respectively. On dark colored axes, these lines appear in yellow, magenta, and cyan colors, respectively.

```
plot(seg)
hold on
```



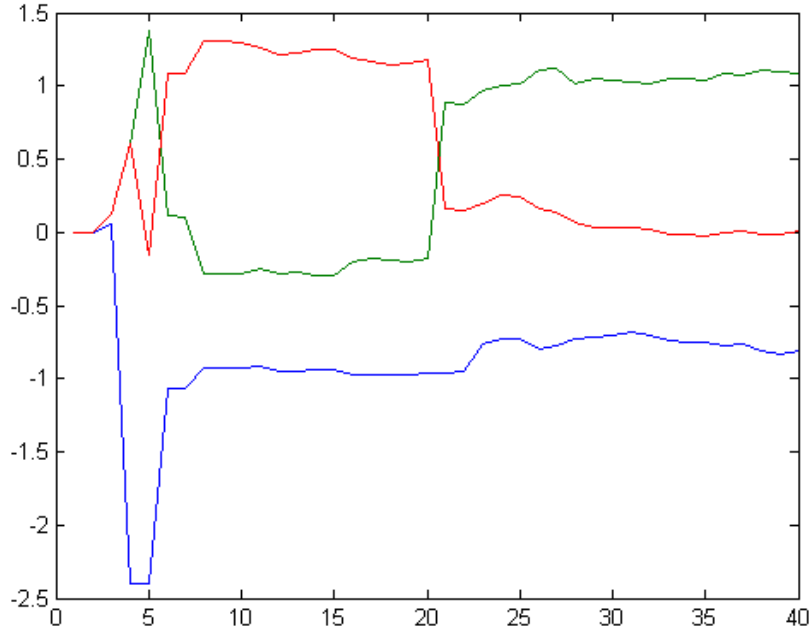
We see clearly the jump around sample number 19. `b1` goes from 0 to 1 and `b2` vice versa, which shows the change of the delay. The true values can also be shown:

```
plot(pars)
hold off
```



The method for segmentation is based on AFMM (adaptive forgetting through multiple models), Andersson, Int. J. Control Nov 1985. A multi-model approach is used in a first step to track the time varying system. The resulting tracking model could be of interest in its own right, and are given by the third output argument of SEGMENT (tvmod in our case). They look as follows:

```
plot(tvmod)
```



The SEGMENT alternative is thus an alternative to the recursive algorithms RPEM, RARX etc for tracking time varying systems. It is particularly suited for systems that may change rapidly.

From the tracking model, SEGMENT estimates the time points when jumps have occurred, and constructs the segmented model by a smoothing procedure over the tracking model.

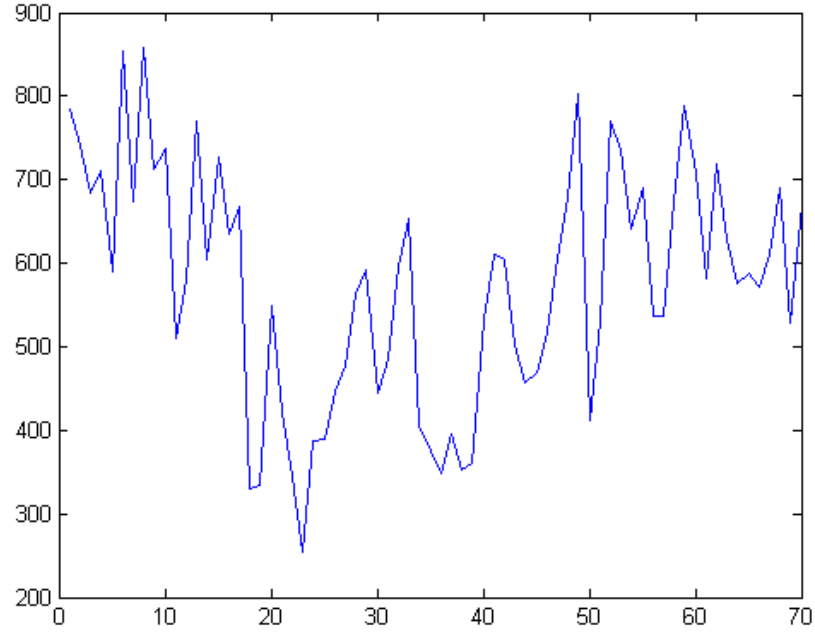
The two most important "knobs" for the algorithm are `r2`, as mentioned before, and the guessed probability of jumps, `q`, the fourth input argument to SEGMENT. The smaller `r2` and the larger `q`, the more willing SEGMENT will be to indicate segmentation points. In an off line situation, the user will have to try a couple of choices (`r2` is usually more sensitive than `q`). The second output argument to SEGMENT, `v`, is the loss function for the segmented model (i.e. the estimated prediction error variance for the segmented model). A goal will be to minimize this value.

Application of SEGMENT: Object Detection in Laser Range Data

The reflected signal from a laser (or radar) beam contains information about the distance to the reflecting object. The signals can be quite noisy. The presence of objects affects both the distance information and the correlation between neighboring points. (A smooth object increases the correlation between nearby points.)

In the following we study some quite noisy laser range data. They are obtained by one horizontal sweep, like one line on a TV-screen. The value is the distance to the reflecting object. We happen to know that an object of interest hides between sample numbers 17 and 48.

```
plot(hline)
```



The eye is not good at detecting the object. We shall use "segment". First we detrend and normalize the data to a variance about one. (This is not necessary, but it means that the default choices in the algorithm are better tuned).

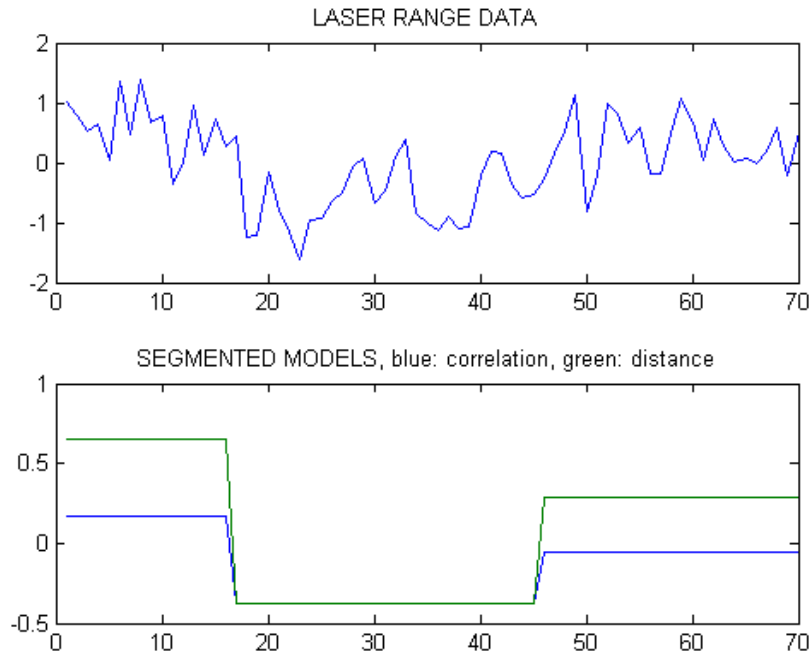
```
hline = detrend(hline)/200;
```

We shall now build a model of the kind:

```
y(t) + a y(t-1) = b
```

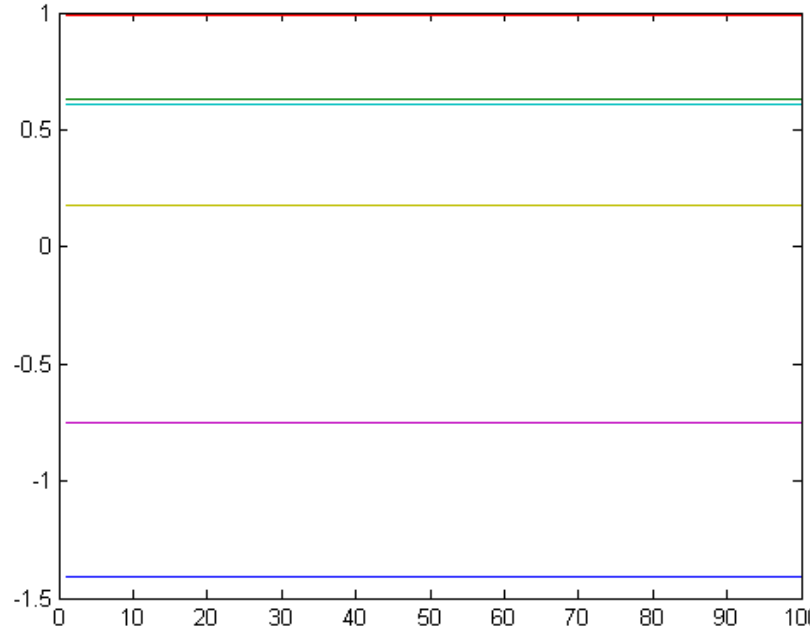
The coefficient 'a' will pick up correlation information. The value 'b' takes up the possible changes in level. We thus introduce a fake input of all ones:

```
[m,n] = size(hline);
zline = [hline ones(m,n)];
s = segment(zline,[1 1 1],0.2);
subplot(211), plot(hline),title('LASER RANGE DATA')
subplot(212), plot(s)
title('SEGMENTED MODELS, blue: correlation, green: distance')
```



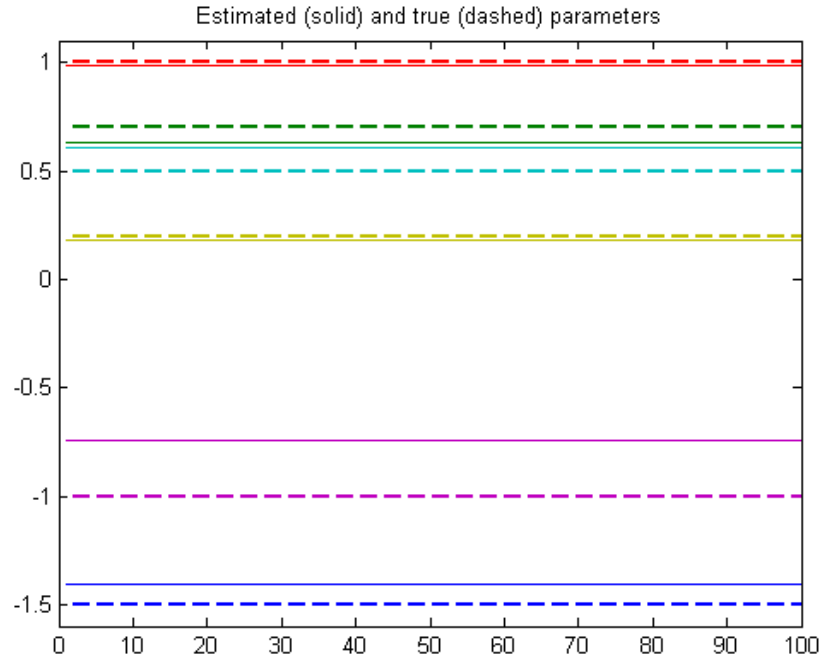
The segmentation has thus been quite successful. SEGMENT is capable of handling multi-input systems, and of using ARMAX models for the added noise. We can try this on the test data iddata1.mat (which contains no jumps):

```
load iddata1.mat
s = segment(z1(1:100),[2 2 2 1],1);
clf
plot(s),hold on
```



Compare this with the true values:

```
plot([ones(100,1)*[-1.5 0.7],ones(100,1)*[1 0.5],ones(100,1)*[-1 0.2]],...
   '--','linewidth',2)
axis([0 100 -1.6 1.1])
title('Estimated (solid) and true (dashed) parameters')
hold off
```



SEGMENT thus correctly finds that no jumps have occurred, and also gives good estimates of the parameters.

Additional Information

For more information on identification of dynamic systems with System Identification Toolbox visit the System Identification Toolbox product information page.

8

Model Analysis

Validating Models After Estimation

In this section...

Ways to Validate Models

You can use the following approaches to validate models:

- Comparing simulated or predicted model output to measured output.

  See "Simulating and Predicting Model Output" on page 8-9.

  To simulate identified models in the Simulink environment, see "Simulating Identified Model Output in Simulink" on page 11-5.

- Analyzing autocorrelation and cross-correlation of the residuals with input.

  See "Residual Analysis" on page 8-24.

- Analyzing model response. For more information, see the following:

  - "Impulse and Step Response Plots" on page 8-33
  - "Frequency Response Plots" on page 8-42

  For information about the response of the noise model, see "Noise Spectrum Plots" on page 8-51.

- Plotting the poles and zeros of the linear parametric model.

  For more information, see "Pole and Zero Plots" on page 8-59.

- Comparing the response of nonparametric models, such as impulse-, step-, and frequency-response models, to parametric models, such as linear polynomial models, state-space model, and nonlinear parametric models.

Note Do not use this comparison when feedback is present in the system because feedback makes nonparametric models unreliable. To test if feedback is present in the system, use the `advice` command on the data.

- Compare models using Akaike Information Criterion or Akaike Final Prediction Error.

  For more information, see the `aic` and `fpe` reference page.

- Plotting linear and nonlinear blocks of Hammerstein-Wiener and nonlinear ARX models.

Displaying confidence intervals on supported plots helps you assess the uncertainty of model parameters. For more information, see "Computing Model Uncertainty" on page 8-89.

Data for Model Validation

For plots that compare model response to measured response and perform residual analysis, you designate two types of data sets: one for estimating the models (*estimation data*), and the other for validating the models (*validation data*). Although you can designate the same data set to be used for estimating and validating the model, you risk over-fitting your data. When you validate a model using an independent data set, this process is called *cross-validation*.

Note Validation data should be the same in frequency content as the estimation data. If you detrended the estimation data, you must remove the same trend from the validation data. For more information about detrending, see "Handling Offsets and Trends in Data" on page 2-104.

Supported Model Plots

The following table summarizes the types of supported model plots.

Plot Type	Supported Models	Learn More
Model Output	All linear and nonlinear models	"Simulating and Predicting Model Output" on page 8-9
Residual Analysis	All linear and nonlinear models	"Residual Analysis" on page 8-24
Transient Response	• All linear parametric models • Correlation analysis (nonparametric) models • For nonlinear models, only step response.	"Impulse and Step Response Plots" on page 8-33
Frequency Response	• All linear parametric models • Spectral analysis (nonparametric) models	"Frequency Response Plots" on page 8-42
Noise Spectrum	• All linear parametric models • Spectral analysis (nonparametric) models	"Noise Spectrum Plots" on page 8-51
Poles and Zeros	All linear parametric models	"Pole and Zero Plots" on page 8-59
Nonlinear ARX	Nonlinear ARX models only	Nonlinear ARX Plots
Hammerstein-Wiener	Hammerstein-Wiener models only	Hammerstein-Wiener Plots

Definition of Confidence Interval for Specific Model Plots

You can display the confidence interval on the following plot types:

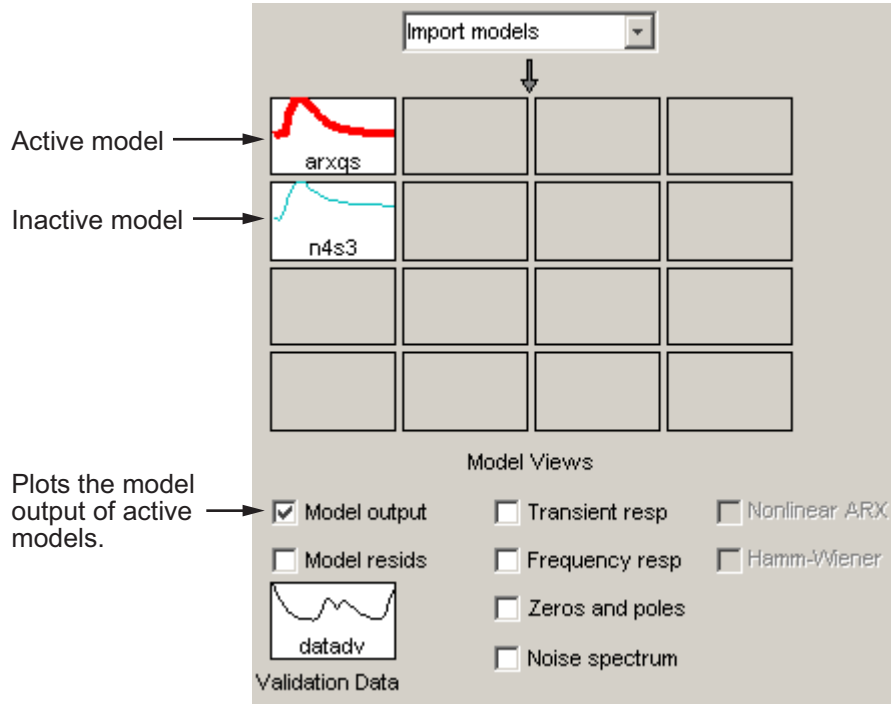
Plot Type	Confidence Interval Corresponds to the Range of ...	More Information on Displaying Confidence Interval
Simulated and Predicted Output	Output values with a specific probability of being the actual output of the system.	Model Output Plots
Residuals	Residual values with a specific probability of being statistically insignificant for the system.	Residuals Plots
Impulse and Step	Response values with a specific probability of being the actual response of the system.	Impulse and Step Plots
Frequency Response	Response values with a specific probability of being the actual response of the system.	Frequency Response Plots
Noise Spectrum	Power-spectrum values with a specific probability of being the actual noise spectrum of the system.	Noise Spectrum Plots
Poles and Zeros	Pole or zero values with a specific probability of being the actual pole or zero of the system.	Pole-Zero Plots

Plotting Models in the GUI

To create one or more plots of your models, select the corresponding check box in the **Model Views** area of the System Identification Tool GUI. An *active* model icon has a thick line in the icon, while an *inactive* model has a thin line. Only active models appear on the selected plots.

To include or exclude a model on a plot, click the corresponding icon in the System Identification Tool GUI. Clicking the model icon updates any plots that are currently open.

For example, in the following figure, **Model output** is selected. In this case, the models `n4s4` is not included on the plot because only `arx441` is active.



Plots Include Only Active Models

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

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

For general information about working with plots in the System Identification Toolbox product , see “Working with Plots” on page 12-13.

Simulating and Predicting Model Output

In this section...

Why Simulate or Predict Model Output

You primarily use a model is to simulate its output, i.e., calculate the output ($y(t)$) for given input values. You can also predict model output, i.e., compute a qualified guess of future output values based on past observations of system's inputs and outputs. For more information, see "Definition: Simulation and Prediction" on page 8-10.

You also validate linear parametric models and nonlinear models by checking how well the simulated or predicted output of the model matches the measured output. You can use either time or frequency domain data for simulation or prediction. For frequency domain data, the simulation and prediction results are products of the Fourier transform of the input and frequency function of the model. For more information, see "Simulation and Prediction in the GUI" on page 8-12 and "Simulation and Prediction at the Command Line" on page 8-18.

Simulation provides a better validation test for the model than prediction. However, how you validate the model output should match how you plan to use the model. For example, if you plan to use your model for control design, you can validate the model by predicting its response over a time horizon that represents the dominating time constants of the model.

Related Examples

"Compare Simulated Output with Measured Data" on page 8-21

"Simulate Model Output with Noise" on page 8-21

"Simulate a Continuous-Time State-Space Model" on page 8-22

"Predict Using Time-Series Model" on page 8-23

Definition: Simulation and Prediction

Simulation means computing the model response using input data and initial conditions. The time samples of the model response match the time samples of the input data used for simulation.

For a continuous-time system, simulation means solving a differential equation. For a discrete-time system, simulation means directly applying the model equations.

For example, consider a dynamic model described by a first-order difference equation that uses a sampling interval of 1 second:

$y(t) + ay(t–1) = bu(t–1)$,

where y is the output and u is the input. For parameter values $a = –0.9$ and $b = 1.5$, the equation becomes:

$y(t) – 0.9y(t–1) = 1.5u(t–1)$.

Suppose you want to compute the values $y(1)$, $y(2)$, $y(3)$,... for given input values $u(0) = 2$, $u(1) = 1$, $u(2) = 4$,...Here, $y(1)$ is the value of output at the first sampling instant. Using initial condition of $y(0) = 0$, the values of $y(t)$ for times $t = 1$, 2 and 3 can be computed as:

$y(1) = 0.9y(0) + 1.5u(0) = 0.9*0 + 1.5*2 = 3$

$y(2) = 0.9y(1) + 1.5u(1) = 0.9*3 + 1.5*1 = 4.2$

$y(3) = 0.9y(2) + 1.5u(2) = 0.9*4.2 + 1.5*4 = 9.78$

...

Prediction forecasts the model response k steps ahead into the future using the current and past values of measured input and output values. k is called the *prediction horizon*, and corresponds to predicting output at time kT_s, where T_s is the sampling interval.

For example, suppose you use sensors to measure the input signal $u(t)$ and output signal $y(t)$ of the physical system, described in the previous first-order equation. At the tenth sampling instant ($t = 10$), the output $y(10)$ is 16 mm and the corresponding input $u(10)$ is 12 N. Now, you want to predict the value of the output at the future time $t = 11$. Using the previous equation:

$y(11) = 0.9y(10) + 1.5u(10)$

Hence, the predicted value of future output $y(11)$ at time $t = 10$ is:

$y(11) = 0.9*16 + 1.5*12 = 32.4$

In general, to predict the model response k steps into the future ($k \geq 1$) from the current time t, you should know the inputs up to time $t+k$ and outputs up to time t:

$$y_p(t+k) = f(u(t+k),u(t+k-1),...,u(t),u(t-1),...,u(0) \\ y(t),y(t-1),y(t-2),...,y(0))$$

$u(0)$ and $y(0)$ are the initial states. `f()` represents the *predictor*, which is a dynamic model whose form depends on the model structure. For example, the one-step-ahead predictor y_p of the model $y(t) + ay(t-1) = bu(t)$ is:

$y_p(t+1) = -ay(t) + bu(t+1)$

The difference between prediction and simulation is that in prediction, the past values of outputs used for calculation are measured values while in

simulation the outputs are themselves a result of calculation using inputs and initial conditions.

The way information in past outputs is used depends on the disturbance model H of the model. For the previous dynamic model, $H(z) = \frac{1}{1 + az^{-1}}$. In models of Output-Error (OE) structure ($H(z) = 1$), there is no information in past outputs that can be used for predicting future output values. In this case, predictions and simulations coincide. For state-space models (`idss`), output-error structure corresponds to models with `K=0`. For polynomial models (`idpoly`), this corresponds to models with polynomials `a=c=d=1`.

Note Prediction with $k=\infty$ means that no previous outputs are used in the computation and prediction returns the same result as simulation.

Both simulation and prediction require initial conditions, which correspond to the states of the model at the beginning of the simulation or prediction.

Tip If you do not know the initial conditions and have input and output measurements available, you can estimate the initial condition using this toolbox.

Simulation and Prediction in the GUI

How to Plot Simulated and Predicted Model Output

To create a model output plot for parametric linear and nonlinear models in the System Identification Tool GUI, select the **Model output** check box in

the **Model Views** area. By default, this operation estimates the initial states from the data and plots the output of selected models for comparison.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool GUI. Active models display a thick line inside the Model Board icon.

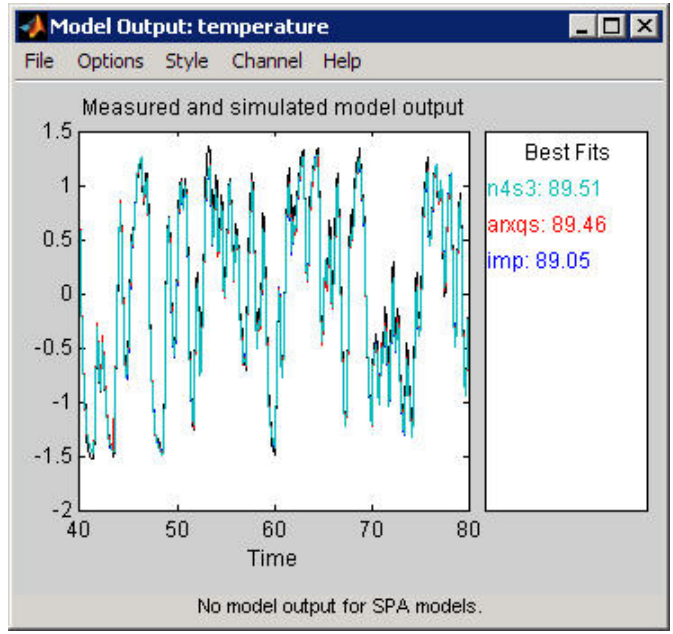
To learn how to interpret the model output plot, see "Interpreting the Model Output Plot" on page 8-13.

To change plot settings, see "Changing Model Output Plot Settings" on page 8-15.

For general information about creating and working with plots, see "Working with Plots" on page 12-13.

Interpreting the Model Output Plot

The following figure shows a sample Model Output plot, created in the System Identification Tool GUI.



The model output plot shows different information depending on the domain of the input-output validation data, as follows:

- For time-domain validation data, the plot shows simulated or predicted model output.
- For frequency-domain data, the plot shows the amplitude of the model response to the frequency-domain input signal. The model response is equal to the product of the Fourier transform of the input and the model's frequency function.
- For frequency-response data, the plot shows the amplitude of the model frequency response.

For linear models, you can estimate a model using time-domain data, and then validate the model using frequency domain data. For nonlinear models, you can only use time-domain data for both estimation and validation.

The right side of the plot displays the percentage of the output that the model reproduces (**Best Fit**), computed using the following equation:

$$\text{Best Fit} = \left(1 - \frac{|y - \hat{y}|}{|y - \bar{y}|}\right) \times 100$$

In this equation, y is the measured output, $\hat{y}$ is the simulated or predicted model output, and $\bar{y}$ is the mean of y. 100% corresponds to a perfect fit, and 0% indicates that the fit is no better than guessing the output to be a constant ($\hat{y} = \bar{y}$).

Because of the definition of **Best Fit**, it is possible for this value to be negative. A negative best fit is worse than 0% and can occur for the following reasons:

- The estimation algorithm failed to converge.
- The model was not estimated by minimizing $|y - \hat{y}|$. **Best Fit** can be negative when you minimized 1-step-ahead prediction during the estimation, but validate using the simulated output $\hat{y}$.
- The validation data set was not preprocessed in the same way as the estimation data set.

Changing Model Output Plot Settings

The following table summarizes the Model Output plot settings.

Model Output Plot Settings

Action	Command
Display confidence intervals.	• To display the dashed lines on either side of the nominal model

Model Output Plot Settings (Continued)

Action	Command
Note Confidence intervals are only available for simulated model output of linear models. Confidence internal are not available for nonlinear ARX and Hammerstein-Wiener models. See "Definition: Confidence Interval" on page 8-17.	curve, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Change between simulated output or predicted output. **Note** Prediction is only available for time-domain validation data.	• Select **Options > Simulated output** or **Options > k step ahead predicted output**. • To change the prediction horizon, select **Options > Set prediction horizon**, and select the number of samples. • To enter your own prediction horizon, select **Options > Set prediction horizon > Other**. Enter the value in terms of the number of samples.
Display the actual output values (**Signal plot**), or the difference between model output and measured output (**Error plot**).	Select **Options > Signal plot** or **Options > Error plot**.

Model Output Plot Settings (Continued)

Action	Command
(Time-domain validation data only) Set the time range for model output and the time interval for which the **Best Fit** value is computed.	Select **Options > Customized time span for fit** and enter the minimum and maximum time values. For example: `[1 20]`
(Multiple-output system only) Select a different output.	Select the output by name in the **Channel** menu.

Definition: Confidence Interval

The *confidence interval* corresponds to the range of output values with a specific probability of being the actual output of the system. The toolbox uses the estimated uncertainty in the model parameters to calculate confidence intervals and assumes the estimates have a Gaussian distribution.

For example, for a 95% confidence interval, the region around the nominal curve represents the range of values that have a 95% probability of being the true system response. You can specify the confidence interval as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution. For example, a probability of 0.99 (99%) corresponds to 2.58 standard deviations.

Note The calculation of the confidence interval assumes that the model sufficiently describes the system dynamics and the model residuals pass independence tests.

In the GUI, you can display a confidence interval on the plot to gain insight into the quality of a linear model. To learn how to show or hide confidence interval, see “Changing Model Output Plot Settings” on page 8-15.

Simulation and Prediction at the Command Line

Summary of Simulation and Prediction Commands

Note If you estimated a linear model from detrended data and want to simulate or predict the output at the original operation conditions, use the `retrend` command to the simulated or predicted output.

Command	Description	Example
`compare`	Use this command for model validation to determine how closely the simulated model response matches the measured output signal . Plots simulated or predicted output of one or more models on top of the measured output. You should use an independent validation data set as input to the model.	To plot five-step-ahead predicted output of the model `mod` against the validation data `data`, use the following command: `compare(data,mod,5)` **Note** Omitting the third argument assumes an infinite horizon and results in the comparison of the simulated response to the input data.
`sim`	Simulate and plot the model output only.	To simulate the response of the model `model` using input data `data`, use the following command: `sim(model,data)`

Command	Description	Example

`predict`	Predict and plot the model output only.	To perform one-step-ahead prediction of the response for the model `model` and input data `data`, use the following command:
		
		`predict(model,data,1)`
		
		Use the following syntax to compute `k`-step-ahead prediction of the output signal using model `m`:
		
		`yhat = predict(m,[y u],k)`
		
		Note that `predict` computes the prediction results only over the time range of `data`. It does not perform any forecasting of results beyond the available data range.

`forecast`	Forecast a time series into the future.	To forecast the value of a time series in an arbitrary number of steps into the future, use the following command: `forecast(model,past_data,K)` Here, `model` is a time series model, `past_data` is a record of the observed values of the time series and `K` is the forecasting horizon.

Initial States in Simulation and Prediction

The process of computing simulated and predicted responses over a time range starts by using the initial conditions to compute the first few output

values. sim, forecast and predict commands provide defaults for handling initial conditions.

Simulation: Default initial conditions are zero for polynomial (idpoly), process (idproc) and transfer-function (idtf) models . For state-space (idss) and grey-box (idgrey) models, the default initial conditions are the internal model initial states (model property x0). You can specify other initial conditions using the InitialCondition simulation option (see simOptions).

Use the compare command to validate models by simulation because its algorithm estimates the initial states of a model to optimize the model fit to a given data set.

If you use sim, the simulated and the measured responses might differ when the initial conditions of the estimated model and the system that measured the validation data set differ—especially at the beginning of the response. To minimize this difference, estimate the initial state values from the data using the findstates command and specify these initial states as input arguments to the sim command. For example, to compute the initial states that optimize the fit of the model m to the output data in z:

```
% Estimate the initial states
X0est = findstates(m,z);
% Simulate the response using estimated initial states
opt = simOptions('InitialCondition',X0est);
sim(m,z.InputData,opt)
```

See Also: sim (for linear models), sim(idnlarx), sim(idnlgrey), sim(idnlhw)

Prediction: Default initial conditions depend on the type of model. You can specify other initial conditions using the InitialCondition option (see predictOptions). For example, to compute the initial states that optimize the 1-step-ahead predicted response of the model m to the output data z:

```
opt = predictOptions('InitialCondition','estimate');
[Yp,X0est] = predict(m,z,1,opt);
```

This command returns the estimated initial states as the output argument `X0est`. For information about other ways to specify initials states, see the `predict` reference page for the corresponding model type.

See Also: `predict`

Compare Simulated Output with Measured Data

This example shows how to validate an estimated model by comparing the simulated model output with measured data.

```
% Create estimation and validation data.
 data1
ze = z1(1:150);
zv = z1(151:300);
% Estimate model.
m= armax(ze,[2 3 1 0]);
% Validate model.
compare(zv,m);
```

Simulate Model Output with Noise

This example shows how you can create input data and a model, and then use the data and the model to simulate output data. In this case, you use the following ARMAX model with Gaussian noise e:

$$y(t) - 1.5y(t-1) + 0.7y(t-2) = u(t-1) + 0.5u(t-2) + e(t) - e(t-1) + 0.2e(t-1)$$

Create the ARMAX model and simulate output data with random binary input `u` using the following commands:

```
% Create an ARMAX model
  m_armax = idpoly([1 -1.5 0.7],[0 1 0.5],[1 -1 0.2]);
% Create a random binary input
  u = idinput(400,'rbs',[0 0.3]);
% Simulate the output data
opt = simOptions('AddNoise',true);
  y = sim(m_armax,u,opt);
```

Note The `'AddNoise'` option specifies to include in the simulation the Gaussian noise e present in the model. Set this option to `false` (default behavior) to simulate the noise-free response to the input `u`, which is equivalent to setting e to zero.

Simulate a Continuous-Time State-Space Model

This example shows how to simulate a continuous-time state-space model using a random binary input `u` and a sampling interval of `0.1` s.

Consider the following state-space model:

$$\dot{x} = \begin{bmatrix} -1 & 1 \\ -0.5 & 0 \end{bmatrix} x + \begin{bmatrix} 1 \\ 0.5 \end{bmatrix} u + \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} e$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} x + e$$

where e is Gaussian white noise with variance `7`.

Use the following commands to simulate the model:

```
% Set up the model matrices
  A = [-1 1;-0.5 0]; B = [1; 0.5];
  C = [1 0]; D = 0; K = [0.5;0.5];
% Create a continuous-time state-space model
% Ts = 0 indicates continuous time
  model_ss = idss(A,B,C,D,K,'Ts',0,'NoiseVariance',7)
% Create a random binary input
  u = idinput(400,'rbs',[0 0.3]);
% Create an iddata object with empty output to represent just
% the input signal
  data = iddata([],u);
  data.ts = 0.1
% Simulate the output using the model
  opt = simOptions('AddNoise',true);
  y=sim(model_ss,data,opt);
```

Predict Using Time-Series Model

This example shows how to evaluate how well a time-series model predicts the response for a given prediction horizon.

In this example, `y` is the original series of monthly sales figures. You use the first half of the measured data to estimate the time-series model and test the model's ability to forecast sales six months ahead using the entire data set.

```
% Select the first half of the data for estimation
% y1 = y(1:48)
% Estimate a fourth-order autoregressive model
% using the first half of the data.
m = ar(y1,4)
% Compute 6-step ahead prediction
yhat = predict(m,y,6)
% Forecast the response 10 steps beyond the available data's time range.
yfuture = forecast(m,y,10);
% Plot the measured, predicted and forecasted outputs
plot(y,yhat,yfuture)
```

Residual Analysis

In this section...

What Is Residual Analysis?

Residuals are differences between the one-step-predicted output from the model and the measured output from the validation data set. Thus, residuals represent the portion of the validation data not explained by the model.

Residual analysis consists of two tests: the whiteness test and the independence test.

According to the *whiteness test* criteria, a good model has the residual autocorrelation function inside the confidence interval of the corresponding estimates, indicating that the residuals are uncorrelated.

According to the *independence test* criteria, a good model has residuals uncorrelated with past inputs. Evidence of correlation indicates that the model does not describe how part of the output relates to the corresponding input. For example, a peak outside the confidence interval for lag k means that the output $y(t)$ that originates from the input $u(t-k)$ is not properly described by the model.

Your model should pass both the whiteness and the independence tests, except in the following cases:

- For output-error (OE) models and when using instrumental-variable (IV) methods, make sure that your model shows independence of `e` and `u`, and pay less attention to the results of the whiteness of `e`.

  In this case, the modeling focus is on the dynamics G and not the disturbance properties H.

- Correlation between residuals and input for negative lags, is not necessarily an indication of an inaccurate model.

  When current residuals at time t affect future input values, there might be feedback in your system. In the case of feedback, concentrate on the positive lags in the cross-correlation plot during model validation.

Supported Model Types

You can validate parametric linear and nonlinear models by checking the behavior of the model residuals. For a description of residual analysis, see "What Residual Plots Show for Different Data Domains" on page 8-25.

Note Residual analysis plots are not available for frequency response (FRD) models. For time-series models, you can only generate model-output plots for parametric models using time-domain time-series (no input) measured data.

What Residual Plots Show for Different Data Domains

Residual analysis plots show different information depending on whether you use time-domain or frequency-domain input-output validation data.

For time-domain validation data, the plot shows the following two axes:

- Autocorrelation function of the residuals for each output
- Cross-correlation between the input and the residuals for each input-output pair

---

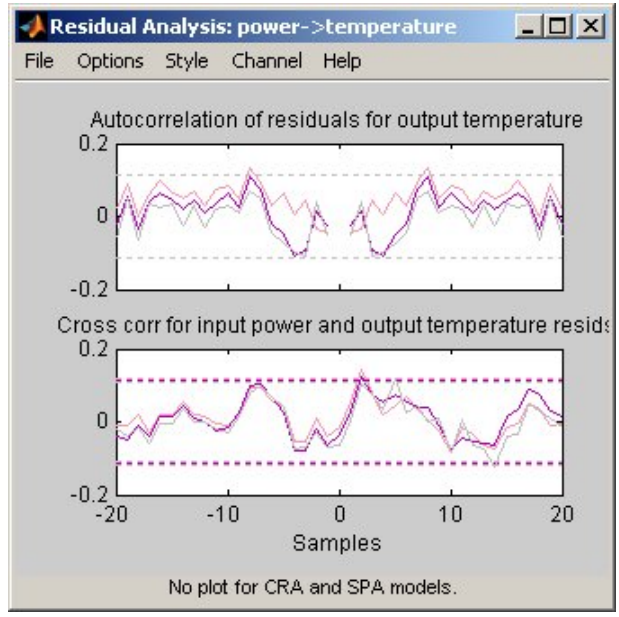
Note For time-series models, the residual analysis plot does not provide any input-residual correlation plots.

For frequency-domain validation data, the plot shows the following two axes:

- Estimated power spectrum of the residuals for each output
- Transfer-function amplitude from the input to the residuals for each input-output pair

For linear models, you can estimate a model using time-domain data, and then validate the model using frequency domain data. For nonlinear models, the System Identification Toolbox product supports only time-domain data.

The following figure shows a sample Residual Analysis plot, created in the System Identification Tool GUI.



Displaying the Confidence Interval

The *confidence interval* corresponds to the range of residual values with a specific probability of being statistically insignificant for the system. The toolbox uses the estimated uncertainty in the model parameters to calculate confidence intervals and assumes the estimates have a Gaussian distribution.

For example, for a 95% confidence interval, the region around zero represents the range of residual values that have a 95% probability of being statistically insignificant. You can specify the confidence interval as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution. For example, a probability of 0.99 (99%) corresponds to 2.58 standard deviations.

You can display a confidence interval on the plot in the GUI to gain insight into the quality of the model. To learn how to show or hide confidence interval, see the description of the plot settings in "How to Plot Residuals Using the GUI" on page 8-27.

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 interval is fixed at 99%.

How to Plot Residuals Using the GUI

To create a residual analysis plot for parametric linear and nonlinear models in the System Identification Tool GUI, select the **Model resids** check box in the **Model Views** area. For general information about creating and working with plots, see "Working with Plots" on page 12-13.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool GUI. Active models display a thick line inside the Model Board icon.

The following table summarizes the Residual Analysis plot settings.

Residual Analysis Plot Settings

Action	Command
Display confidence intervals around zero. **Note** Confidence internal are not available for nonlinear ARX and Hammerstein-Wiener models.	• To display the dashed lines on either side of the nominal model curve, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level** and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Change the number of lags (data samples) for which to compute autocorellation and cross-correlation functions. **Note** For frequency-domain validation data, increasing the number of lags increases the frequency resolution of the residual spectrum and the transfer function.	• Select **Options > Number of lags** and choose the value from the list. • To enter your own lag value, select **Options > Set confidence level > Other**. Enter the value as the number of data samples.
(Multiple-output system only) Select a different input-output pair.	Select the input-output by name in the **Channel** menu.

How to Plot Residuals at the Command Line

The following table summarizes commands that generate residual-analysis plots for linear and nonlinear models. For detailed information about this command, see the corresponding reference page.

Note Apply `pe` and `resid` to one model at a time.

Command	Description	Example
`pe`	Computes and plots model prediction errors.	To plot the prediction errors for the model `model` using data `data`, type the following command: `pe(model,data)`
`resid`	Performs whiteness and independence tests on model residuals, or prediction errors. Uses validation data input as model input.	To plot residual correlations for the model `model` using data `data`, type the following command: `resid(model,data)`

Examine Model Residuals

This example shows how you can use residual analysis to evaluate model quality.

Creating Residual Plots

1. To load the sample System Identification Tool session that contains estimated models, type the following command in the MATLAB Command Window:

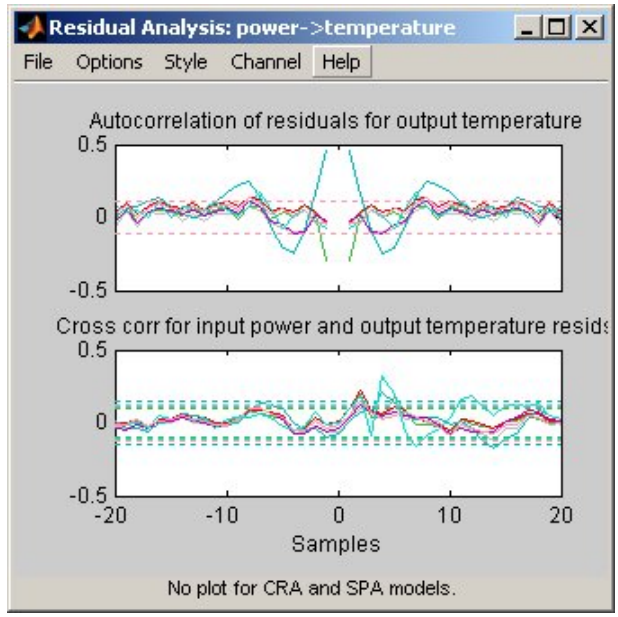
   ```
   ident('dryer2_linear_models')
   ```

2. To generate a residual analysis plot, select the **Model resids** check box in the System Identification Tool GUI.

This opens an empty plot.

3 In the System Identification Tool window, click each model icon to display it on the Residual Analysis plot.

Note For the nonparametric models, `imp` and `spad`, residual analysis plots are not available.



Description of the Residual Plot Axes

The top axes show the autocorrelation of residuals for the output (whiteness test). The horizontal scale is the number of lags, which is the time difference (in samples) between the signals at which the correlation is estimated. The horizontal dashed lines on the plot represent the confidence interval of the corresponding estimates. Any fluctuations within the confidence interval are considered to be insignificant. Four of the models, `arxqs`, `n4s3`, `arx223` and `amx2222`, produce residuals that enter outside the confidence interval. A good

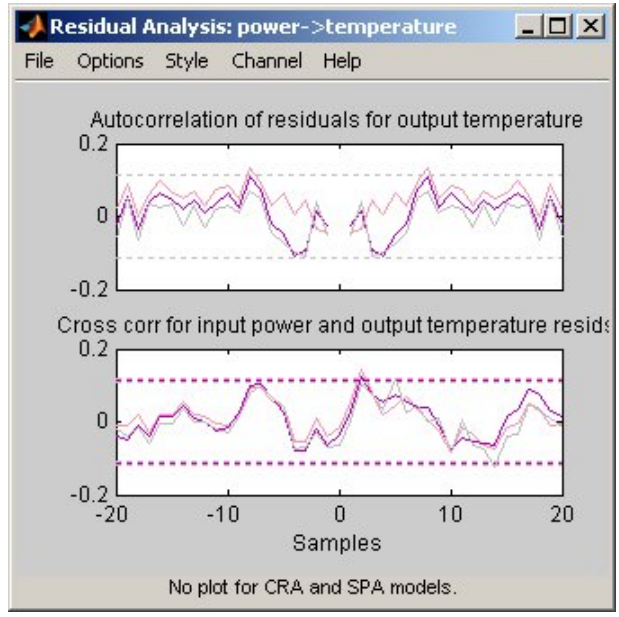
model should have a residual autocorrelation function within the confidence interval, indicating that the residuals are uncorrelated.

The bottom axes show the cross-correlation of the residuals with the input. A good model should have residuals uncorrelated with past inputs (independence test). Evidence of correlation indicates that the model does not describe how the output is formed from the corresponding input. For example, when there is a peak outside the confidence interval for lag *k*, this means that the contribution to the output *y(t)* that originates from the input *u(t-k)* is not properly described by the model. The models `arxqs` and `amx2222` extend beyond the confidence interval and do not perform as well as the other models.

Validating Models Using Analyzing Residuals

To remove models with poor performance from the Residual Analysis plot, click the model icons `arxqs`, `n4s3`, `arx223`, and `amx2222` in the System Identification Tool GUI.

The Residual Analysis plot now includes only the three models that pass the residual tests: `arx692`, `n4s6`, and `amx3322`.



The plots for these models fall within the confidence intervals. Thus, when choosing the best model among several estimated models, it is reasonable to pick `amx3322` because it is a simpler, low-order model.

Impulse and Step Response Plots

In this section...

Supported Models

You can plot the simulated response of a model using impulse and step signals as the input for all linear parametric models and correlation analysis (nonparametric) models.

You can also create step-response plots for nonlinear models. These step and impulse response plots, also called *transient response* plots, provide insight into the characteristics of model dynamics, including peak response and settling time.

Note For frequency-response models, impulse- and step-response plots are not available. For nonlinear models, only step-response plots are available.

Examples

"How to Plot Impulse and Step Response Using the GUI" on page 8-37

"How to Plot Impulse and Step Response at the Command Line" on page 8-40

How Transient Response Helps to Validate Models

Transient response plots provide insight into the basic dynamic properties of the model, such as response times, static gains, and delays.

Transient response plots also help you validate how well a linear parametric model, such as a linear ARX model or a state-space model, captures the

dynamics. For example, you can estimate an impulse or step response from the data using correlation analysis (nonparametric model), and then plot the correlation analysis result on top of the transient responses of the parametric models.

Because nonparametric and parametric models are derived using different algorithms, agreement between these models increases confidence in the parametric model results.

What Does a Transient Response Plot Show?

Transient response plots show the value of the impulse or step response on the vertical axis. The horizontal axis is in units of time you specified for the data used to estimate the model.

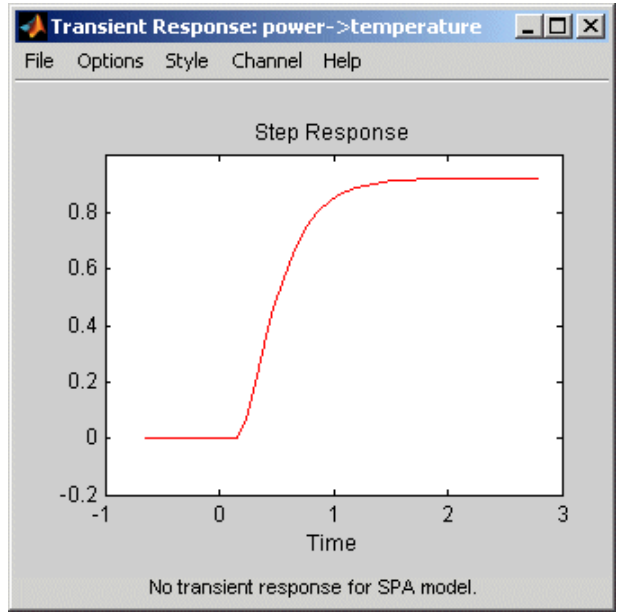
The impulse response of a dynamic model is the output signal that results when the input is an impulse. That is, *u(t)* is zero for all values of *t* except at *t*=0, where *u(0)*=1. In the following difference equation, you can compute the impulse response by setting *y(-T)=y(-2T)*=0, *u(0)*=1, and *u(t>0)*=0.

$$y(t) - 1.5y(t-T) + 0.7y(t-2T) = 0.9u(t) + 0.5u(t-T)$$

The step response is the output signal that results from a step input, where *u(t<0)*=0 and *u(t>0)*=1.

If your model includes a noise model, you can display the transient response of the noise model associated with each output channel. For more information about how to display the transient response of the noise model, see "How to Plot Impulse and Step Response Using the GUI" on page 8-37.

The following figure shows a sample Transient Response plot, created in the System Identification Tool GUI.



Displaying the Confidence Interval

In addition to the transient-response curve, you can display a confidence interval on the plot. To learn how to show or hide confidence interval, see the description of the plot settings in "How to Plot Impulse and Step Response Using the GUI" on page 8-37.

The *confidence interval* corresponds to the range of response values with a specific probability of being the actual response of the system. The toolbox uses the estimated uncertainty in the model parameters to calculate confidence intervals and assumes the estimates have a Gaussian distribution.

For example, for a 95% confidence interval, the region around the nominal curve represents the range where there is a 95% chance that it contains the true system response. You can specify the confidence interval as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution. For example, a probability of 0.99 (99%) corresponds to 2.58 standard deviations.

Note The calculation of the confidence interval assumes that the model sufficiently describes the system dynamics and the model residuals pass independence tests.

How to Plot Impulse and Step Response Using the GUI

To create a transient analysis plot in the System Identification Tool GUI, select the **Transient resp** check box in the **Model Views** area. For general information about creating and working with plots, see "Working with Plots" on page 12-13.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool GUI. Active models display a thick line inside the Model Board icon.

The following table summarizes the Transient Response plot settings.

Transient Response Plot Settings

Action	Command
Display step response for linear or nonlinear model.	Select **Options > Step response**.
Display impulse response for linear model. **Note** Not available for nonlinear models.	Select **Options > Impulse response**.
Display the confidence interval. **Note** Only available for linear models.	• To display the dashed lines on either side of the nominal model curve, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as

Transient Response Plot Settings (Continued)

Action	Command
	the number of standard deviations of a Gaussian distribution.
Change time span over which the impulse or step response is calculated. For a scalar time span *T*, the resulting response is plotted from *-T/4* to *T*. **Note** To change the time span of models you estimated using correlation analysis models, select **Estimate > Correlation models** and reestimate the model using a new time span.	• Select **Options > Time span (time units)**, and choose a new time span in units of time you specified for the model. • To enter your own time span, select **Options > Time span (time units) > Other**, and enter the total response duration. • To use the time span based on model dynamics, type `[]` or `default`. The `default` time span is computed based on the model dynamics and might be different for different models. For nonlinear models, the default time span is 10.
Toggle between line plot or stem plot. **Tip** Use a stem plot for displaying impulse response.	Select **Style > Line plot** or **Style > Stem plot**.

Transient Response Plot Settings (Continued)

Action	Command
(Multiple-output system only) Select an input-output pair to view the noise spectrum corresponding to those channels.	Select the output by name in the **Channel** menu. If the plotted models include a noise model, you can display the transient response properties associated with each output channel. The name of the channel has the format `e@OutputName`, where `OutputName` is the name of the output channel corresponding to the noise model.
(Step response for nonlinear models only) Set level of the input step. **Note** For multiple-input models, the input-step level applies only to the input channel you selected to display in the plot.	Select **Options > Step Size**, and then chose from two options: • **0–>1** sets the lower level to `0` and the upper level to `1`. • **Other** opens the Step Level dialog box, where you enter the values for the lower and upper level values.

More About

"Impulse and Step Response Plots" on page 8-33

How to Plot Impulse and Step Response at the Command Line

You can plot impulse- and step-response plots using the `impulseplot` and `stepplot` commands, respectively. If you want to fetch the response data, use `impulse` and `step` instead.

All plot commands have the same basic syntax, as follows:

- To plot one model, use the syntax *`command(model)`*.
- To plot several models, use the syntax *`command(model1,model2,...,modelN)`*.

In this case, `command` represents any of the plotting commands.

To display confidence intervals for a specified number of standard deviations, use the following syntax:

```
h = impulseplot(model);
showConfidence(h,sd);
```

where `h` is the plot handle returned by `impulseplot`. You could also use the plot handle returned by `stepplot`. `sd` is the number of standard deviations of a Gaussian distribution. For example, a confidence value of 99% for the nominal model curve corresponds to 2.58 standard deviations.

Alternatively, you can turn on the confidence region view interactively by right-clicking on the plot and selecting **Characteristics > Confidence Region**. Use the plot property editor to specify the number of standard deviations.

The following table summarizes commands that generate impulse- and step-response plots. For detailed information about each command, see the corresponding reference page.

Command	Description	Example
`impulse`,`impulseplot`	Plot impulse response for `idpoly`, `idproc`, `idtf`, `idss`, and `idgrey` model objects. **Note** Does not support nonlinear models.	To plot the impulse response of the model `sys`, type the following command: `impulse(sys)`

`step`,`stepplot`	Plots the step response of all linear and nonlinear models.	To plot the step response of the model `sys`, type the following command:
		
		`step(sys)`
		
		To specify the step level offset (`u0`) and amplitude (`A`) for a model:
		
		`opt = stepDataOptions;`
		`opt.InputOffset = u0;`
		`opt.StepAmplitude = A;`
		
		`step(sys,opt)`

More About

"Impulse and Step Response Plots" on page 8-33

Frequency Response Plots

In this section...

What Is Frequency Response?

Frequency response plots show the complex values of a transfer function as a function of frequency.

In the case of linear dynamic systems, the transfer function G is essentially an operator that takes the input u of a linear system to the output y:

$$y = Gu$$

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

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

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

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

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

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

Examples

"How to Plot Bode Plots Using the GUI" on page 8-46

"How to Plot Bode and Nyquist Plots at the Command Line" on page 8-49

How Frequency Response Helps to Validate Models

You can plot the frequency response of a model to gain insight into the characteristics of linear model dynamics, including the frequency of the peak response and stability margins. Frequency-response plots are available for all linear parametric models and spectral analysis (nonparametric) models.

Note Frequency-response plots are not available for nonlinear models. In addition, Nyquist plots do not support time-series models that have no input.

The frequency response of a linear dynamic model describes how the model reacts to sinusoidal inputs. If the input *u(t)* is a sinusoid of a certain frequency, then the output *y(t)* is also a sinusoid of the same frequency. However, the magnitude of the response is different from the magnitude of the input signal, and the phase of the response is shifted relative to the input signal.

Frequency response plots provide insight into linear systems dynamics, such as frequency-dependent gains, resonances, and phase shifts. Frequency response plots also contain information about controller requirements and achievable bandwidths. Finally, frequency response plots can also help you validate how well a linear parametric model, such as a linear ARX model or a state-space model, captures the dynamics.

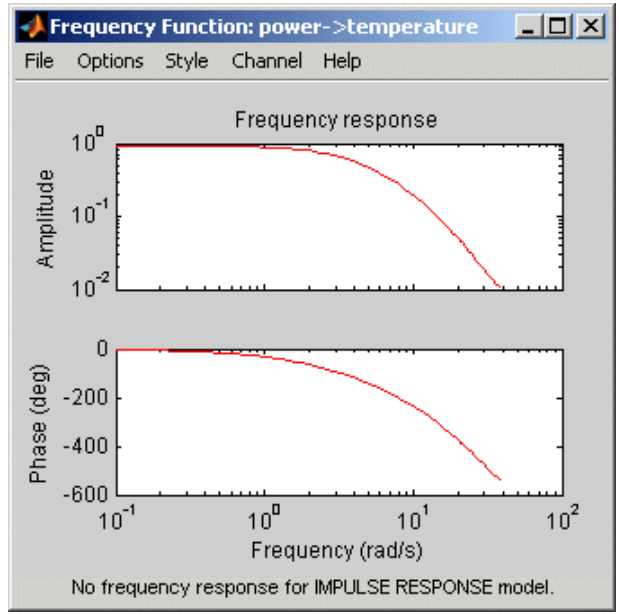
One example of how frequency-response plots help validate other models is that you can estimate a frequency response from the data using spectral analysis (nonparametric model), and then plot the spectral analysis result on top of the frequency response of the parametric models. Because nonparametric and parametric models are derived using different algorithms, agreement between these models increases confidence in the parametric model results.

What Does a Frequency-Response Plot Show?

System Identification Tool GUI supports the following types of frequency-response plots for linear parametric models, linear state-space models, and nonparametric frequency-response models:

- Bode plot of the model response. A Bode plot consists of two plots. The top plot shows the magnitude $|G|$ by which the transfer function G magnifies the amplitude of the sinusoidal input. The bottom plot shows the phase $\varphi = \arg G$ by which the transfer function shifts the input. The input to the system is a sinusoid, and the output is also a sinusoid with the same frequency.
- Plot of the disturbance model, called *noise spectrum*. This plot is the same as a Bode plot of the model response, but it shows the output power spectrum of the noise model instead. For more information, see "Noise Spectrum Plots" on page 8-51.
- (Only in the MATLAB Command Window)
  Nyquist plot. Plots the imaginary versus the real part of the transfer function.

The following figure shows a sample Bode plot of the model dynamics, created in the System Identification Tool GUI.



Displaying the Confidence Interval

In addition to the frequency-response curve, you can display a confidence interval on the plot. To learn how to show or hide confidence interval, see the description of the plot settings in "How to Plot Bode Plots Using the GUI" on page 8-46

The *confidence interval* corresponds to the range of response values with a specific probability of being the actual response of the system. The toolbox uses the estimated uncertainty in the model parameters to calculate confidence intervals and assumes the estimates have a Gaussian distribution.

For example, for a 95% confidence interval, the region around the nominal curve represents the range where there is a 95% chance that it contains the true system response. You can specify the confidence interval as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution. For example, a probability of 0.99 (99%) corresponds to 2.58 standard deviations.

How to Plot Bode Plots Using the GUI

To create a frequency-response plot for parametric linear models in the System Identification Tool GUI, select the **Frequency resp** check box in the **Model Views** area. For general information about creating and working with plots, see "Working with Plots" on page 12-13.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool GUI. Active models display a thick line inside the Model Board icon.

The following table summarizes the Frequency Function plot settings.

Frequency Function Plot Settings

Action	Command
Display the confidence interval.	• To display the dashed lines on either side of the nominal model curve, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Change the frequency values for computing the noise spectrum. The default frequency vector is 128 linearly distributed values, greater than zero and less than	Select **Options > Frequency range** and specify a new frequency vector in units of rad/s. Enter the frequency vector using any one of following methods: • MATLAB expression, such as `[1:100]*pi/100` or `logspace(-3,-1,200)`.

Frequency Function Plot Settings (Continued)

Action	Command
or equal to the Nyquist frequency.	Cannot contain variables in the MATLAB workspace. • Row vector of values, such as `[1:.1:100]` **Note** To restore the default frequency vector, enter `[]`.
Change frequency units between hertz and radians per second.	Select **Style > Frequency (Hz)** or **Style > Frequency (rad/s)**.
Change frequency scale between linear and logarithmic.	Select **Style > Linear frequency scale** or **Style > Log frequency scale**.
Change amplitude scale between linear and logarithmic.	Select **Style > Linear amplitude scale** or **Style > Log amplitude scale**.
(Multiple-output system only) Select an input-output pair to view the noise spectrum corresponding to those channels. **Note** You cannot view cross spectra between different outputs.	Select the output by name in the **Channel** menu.

More About

"Frequency Response Plots" on page 8-42

How to Plot Bode and Nyquist Plots at the Command Line

You can plot Bode and Nyquist plots for linear models using the `bode` and `nyquist` commands. If you want to customize the appearance of the plot, or turn on the confidence region programmatically, use `bodeplot`, and `nyquistplot` instead.

All plot commands have the same basic syntax, as follows:

- To plot one model, use the syntax *`command(model)`*.
- To plot several models, use the syntax *`command(model1,model2,...,modelN)`*.

In this case, *`command`* represents any of the plotting commands.

To display confidence intervals for a specified number of standard deviations, use the following syntax:

```
h=command(model);
showConfidence(h,sd)
```

where `sd` is the number of standard deviations of a Gaussian distribution and command is `bodeplot`or `nyquistplot`. For example, a confidence value of 99% for the nominal model curve corresponds to 2.58 standard deviations.

The following table summarizes commands that generate Bode and Nyquist plots for linear models. For detailed information about each command and how to specify the frequency values for computing the response, see the corresponding reference page.

Command	Description	Example
`bode` and `bodeplot`	Plots the magnitude and phase of the frequency response on a logarithmic frequency scale. **Note** Does not support time-series models.	To create the bode plot of the model, `sys`, use the following command: `bode(sys)`
`nyquist` and `nyquistplot`	Plots the imaginary versus real part of the transfer function. **Note** Does not support time-series models.	To plot the frequency response of the model, `sys`, use the following command: `nyquist(sys)`
`spectrum` and `spectrumplot`	Plots the disturbance spectra of input-output models and output spectra of time series models.	To plot the output spectrum of a time series model, `sys`, with 1 standard deviation confidence region, use the following command: `showConfidence(spectrumplot(sys));`

More About

"Frequency Response Plots" on page 8-42

Noise Spectrum Plots

In this section...

Supported Models

When you estimate the noise model of your linear system, you can plot the spectrum of the estimated noise model. Noise-spectrum plots are available for all linear parametric models and spectral analysis (nonparametric) models.

Note For nonlinear models and correlation analysis models, noise-spectrum plots are not available. For time-series models, you can only generate noise-spectrum plots for parametric and spectral-analysis models.

Examples

"How to Plot the Noise Spectrum Using the GUI" on page 8-54

"How to Plot the Noise Spectrum at the Command Line" on page 8-57

What Does a Noise Spectrum Plot Show?

The general equation of a linear dynamic system is given by:

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

In this equation, G is an operator that takes the input to the output and captures the system dynamics, and v is the additive noise term. The toolbox treats the noise term as filtered white noise, as follows:

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

where $e(t)$ is a white-noise source with variance λ.

The toolbox computes both H and λ during the estimation of the noise model and stores these quantities as model properties. The *H(z)* operator represents the noise model.

Whereas the frequency-response plot shows the response of G, the noise-spectrum plot shows the frequency-response of the noise model H.

For input-output models, the noise spectrum is given by the following equation:

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

For time-series models (no input), the vertical axis of the noise-spectrum plot is the same as the dynamic model spectrum. These axes are the same because there is no input for time series and $y = He$.

Note You can avoid estimating the noise model by selecting the Output-Error model structure or by setting the `DisturbanceModel` property value to `'None'` for a state space model. If you choose to not estimate a noise model for your system, then H and the noise spectrum amplitude are equal to 1 at all frequencies.

Displaying the Confidence Interval

In addition to the noise-spectrum curve, you can display a confidence interval on the plot. To learn how to show or hide confidence interval, see the description of the plot settings in "How to Plot the Noise Spectrum Using the GUI" on page 8-54.

The *confidence interval* corresponds to the range of power-spectrum values with a specific probability of being the actual noise spectrum of the system. The toolbox uses the estimated uncertainty in the model parameters to calculate confidence intervals and assumes the estimates have a Gaussian distribution.

For example, for a 95% confidence interval, the region around the nominal curve represents the range where there is a 95% chance that the true response belongs.. You can specify the confidence interval as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution. For example, a probability of 0.99 (99%) corresponds to 2.58 standard deviations.

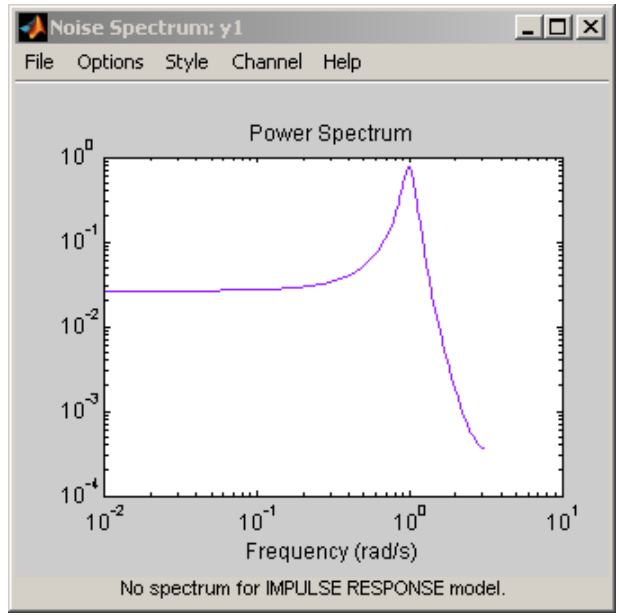
Note The calculation of the confidence interval assumes that the model sufficiently describes the system dynamics and the model residuals pass independence tests.

How to Plot the Noise Spectrum Using the GUI

To create a noise spectrum plot for parametric linear models in the GUI, select the **Noise spectrum** check box in the **Model Views** area. For general information about creating and working with plots, see “Working with Plots” on page 12-13.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool GUI. Active models display a thick line inside the Model Board icon.

The following figure shows a sample Noise Spectrum plot.



The following table summarizes the Noise Spectrum plot settings.

Noise Spectrum Plot Settings

Action	Command
Display the confidence interval.	• To display the dashed lines on either side of the nominal model curve, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Change the frequency values for computing the noise spectrum. The default frequency vector is 128 linearly distributed values, greater than zero and less than or equal to the Nyquist frequency.	Select **Options > Frequency range** and specify a new frequency vector in units of radians per second. Enter the frequency vector using any one of following methods: • MATLAB expression, such as `[1:100]*pi/100` or `logspace(-3,-1,200)`. Cannot contain variables in the MATLAB workspace. • Row vector of values, such as `[1:.1:100]` **Tip** To restore the default frequency vector, enter `[]`.
Change frequency units between hertz and radians per second.	Select **Style > Frequency (Hz)** or **Style > Frequency (rad/s)**.
Change frequency scale between linear and logarithmic.	Select **Style > Linear frequency scale** or **Style > Log frequency scale**.

Noise Spectrum Plot Settings (Continued)

Action	Command
Change amplitude scale between linear and logarithmic.	Select **Style > Linear amplitude scale** or **Style > Log amplitude scale**.
(Multiple-output system only) Select an input-output pair to view the noise spectrum corresponding to those channels. **Note** You cannot view cross spectra between different outputs.	Select the output by name in the **Channel** menu.

More About

"Noise Spectrum Plots" on page 8-51

How to Plot the Noise Spectrum at the Command Line

To plot the disturbance spectrum of an input-output model or the output spectrum of a time series model, use `spectrum`. To customize such plots, or to turn on the confidence region view programmatically for such plots, use `spectrumplot` instead.

To determine if your estimated noise model is good enough, you can compare the output spectrum of the estimated noise-model H to the estimated output spectrum of $v(t)$. To compute $v(t)$, which represents the actual noise term in the system, use the following commands:

```
ysimulated = sim(m,data);
v = ymeasured-ysimulated;
```

`ymeasured` is `data.y`. `v` is the noise term $v(t)$, as described in "What Does a Noise Spectrum Plot Show?" on page 8-51 and corresponds to the difference between the simulated response `ysimulated` and the actual response `ymeasured`.

To compute the frequency-response model of the actual noise, use `spa`:

```
V = spa(v);
```

The toolbox uses the following equation to compute the noise spectrum of the actual noise:

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

The covariance function R_v is given in terms of E, which denotes the mathematical expectation, as follows:

$$R_v(\tau) = Ev(t)v(t-\tau)$$

To compare the parametric noise-model H to the (nonparametric) frequency-response estimate of the actual noise $v(t)$, use `spectrum`:

```
spectrum(V,m)
```

If the parametric and the nonparametric estimates of the noise spectra are different, then you might need a higher-order noise model.

More About

"Noise Spectrum Plots" on page 8-51

Pole and Zero Plots

In this section...

Supported Models

You can create pole-zero plots of linear identified models. To study the poles and zeros of the noise component of an input-output model or a time series model, use `noise2meas` to first extract the noise model as an independent input-output model, whose inputs are the noise channels of the original model.

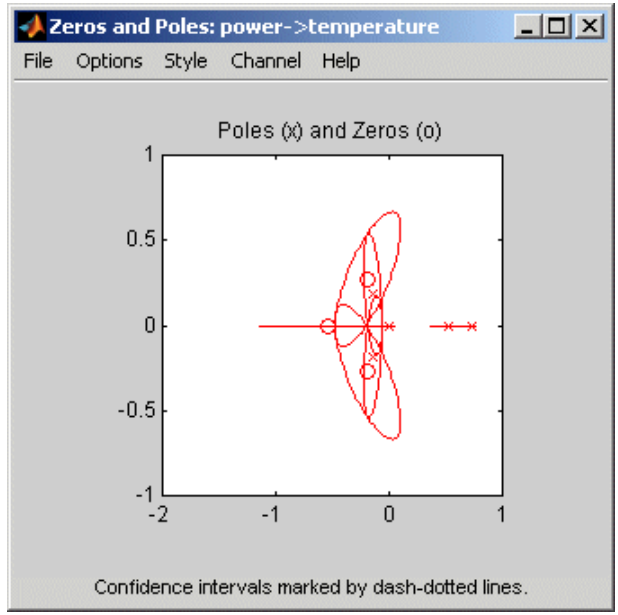
Examples

"How to Plot Model Poles and Zeros Using the GUI" on page 8-63

"How to Plot Poles and Zeros at the Command Line" on page 8-65

What Does a Pole-Zero Plot Show?

The following figure shows a sample pole-zero plot of the model with confidence intervals. `x` indicate poles and `o` indicate zeros.



The general equation of a linear dynamic system is given by:

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

In this equation, G is an operator that takes the input to the output and captures the system dynamics, and v is the additive noise term.

The *poles* of a linear system are the roots of the denominator of the transfer function G. The poles have a direct influence on the dynamic properties of the system. The *zeros* are the roots of the numerator of G. If you estimated a noise model H in addition to the dynamic model G, you can also view the poles and zeros of the noise model.

Zeros and the poles are equivalent ways of describing the coefficients of a linear difference equation, such as the ARX model. Poles are associated with the output side of the difference equation, and zeros are associated with the input side of the equation. The number of poles is equal to the number of sampling intervals between the most-delayed and least-delayed output. The number of zeros) is equal to the number of sampling intervals between the

most-delayed and least-delayed input. For example, there two poles and one zero in the following ARX model:

$$y(t) - 1.5y(t-T) + 0.7y(t-2T) = 0.9u(t) + 0.5u(t-T)$$

Reducing Model Order Using Pole-Zero Plots

You can use pole-zero plots to evaluate whether it might be useful to reduce model order. When confidence intervals for a pole-zero pair overlap, this overlap indicates a possible pole-zero cancelation.

For example, you can use the following syntax to plot a 1-standard-deviation confidence interval around model poles and zeros.

```
showConfidence(iopzplot(model))
```

If poles and zeros overlap, try estimating a lower order model.

Always validate model output and residuals to see if the quality of the fit changes after reducing model order. If the plot indicates pole-zero cancellations, but reducing model order degrades the fit, then the extra poles probably describe noise. In this case, you can choose a different model structure that decouples system dynamics and noise. For example, try ARMAX, Output-Error, or Box-Jenkins polynomial model structures with an A or F polynomial of an order equal to that of the number of uncanceled poles. For more information about estimating linear polynomial models, see "Identifying Input-Output Polynomial Models" on page 3-45.

Displaying the Confidence Interval

In addition, you can display a confidence interval for each pole and zero on the plot. To learn how to show or hide confidence interval, see "How to Plot Model Poles and Zeros Using the GUI" on page 8-63.

The *confidence interval* corresponds to the range of pole or zero values with a specific probability of being the actual pole or zero of the system. The toolbox uses the estimated uncertainty in the model parameters to calculate confidence intervals and assumes the estimates have a Gaussian distribution.

For example, for a 95% confidence interval, the region around the nominal pole or zero value represents the range of values that have a 95% probability of being the true system pole or zero value. You can specify the confidence interval as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution. For example, a probability of 0.99 (99%) corresponds to 2.58 standard deviations.

How to Plot Model Poles and Zeros Using the GUI

To create a pole-zero plot for parametric linear models in the System Identification Tool GUI, select the **Zeros and poles** check box in the **Model Views** area. For general information about creating and working with plots, see "Working with Plots" on page 12-13.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool GUI. Active models display a thick line inside the Model Board icon.

The following table summarizes the Zeros and Poles plot settings.

Zeros and Poles Plot Settings

Action	Command
Display the confidence interval.	• To display the dashed lines on either side of the nominal pole and zero values, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Show real and imaginary axes.	Select **Style > Re/Im-axes**. Select this option again to hide the axes.

Zeros and Poles Plot Settings (Continued)

Action	Command
Show the unit circle.	Select **Style > Unit circle**. Select this option again to hide the unit circle. The unit circle is useful as a reference curve for discrete-time models.
(Multiple-output system only) Select an input-output pair to view the poles and zeros corresponding to those channels.	Select the output by name in the **Channel** menu.

More About

"Pole and Zero Plots" on page 8-59

How to Plot Poles and Zeros at the Command Line

You can create a pole-zero plot for linear identified models using the `iopzmap` and `iopzplot` commands.

To display confidence intervals for a specified number of standard deviations, use the following syntax:

```
h = iopzplot(model);
showConfidence(h,sd)
```

where `sd` is the number of standard deviations of a Gaussian distribution. For example, a confidence value of 99% for the nominal model curve corresponds to 2.58 standard deviations.

Command	Description	Example
`iopzmap`,`iopzplot`	Plots zeros and poles of the model on the S-plane or Z-plane for continuous-time or discrete-time model, respectively.	To plot the poles and zeros of the model `sys`, use the following command: `iopzmap(sys)`

More About

"Pole and Zero Plots" on page 8-59

Analyzing MIMO Models

In this section...

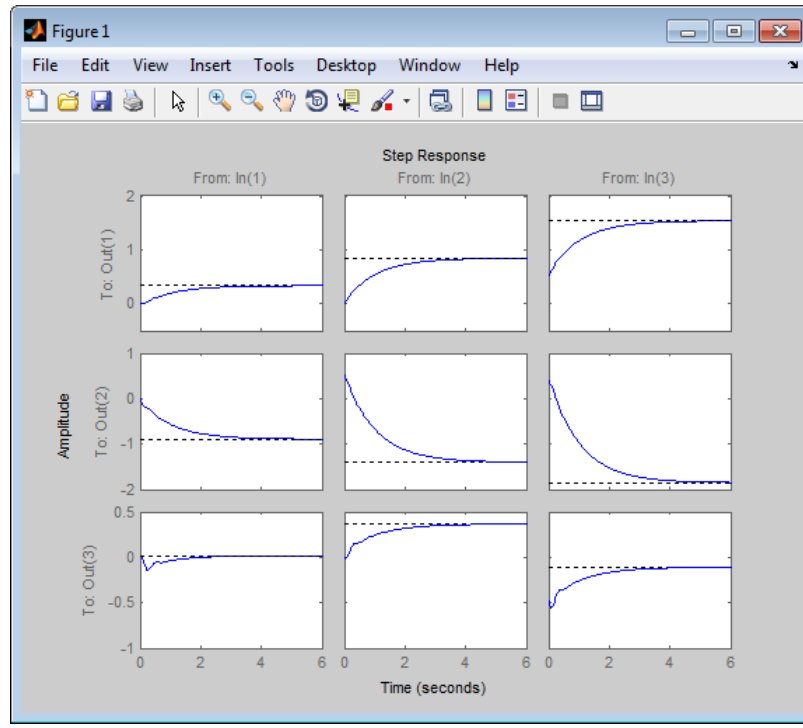
Overview of Analyzing MIMO Models

If you import a MIMO system, or an LTI array containing multiple identified linear models, you can use special features of the right-click menu to group the response plots by input/output (I/O) pairs, or select individual plots for display. For example, generate a random 3-input, 3-output MIMO system and then randomly sample it 10 times. Plot the step response for all the models.

```
sys_mimo=rsample(idss(rss(3,3,3)),10);
step(sys_mimo);
```

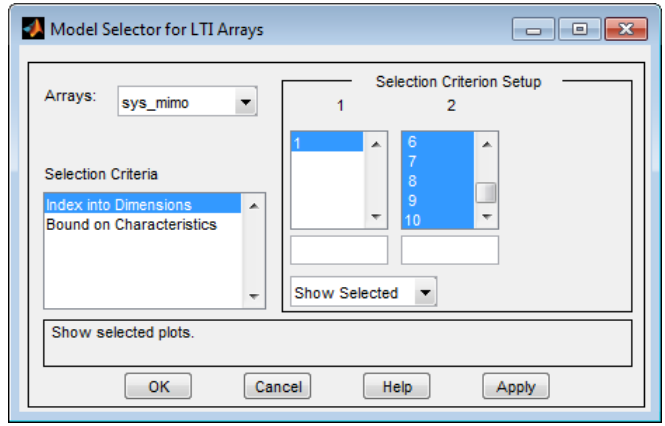
`sys_mimo` is an array of ten 3-input, 3-output systems.

A set of 9 plots appears, one from each input to each output, for the ten model samples.



Array Selector

If you plot an identified linear model array, **Array Selector** appears as an option in the right-click menu. Selecting this option opens the **Model Selector for LTI Arrays**, shown below.



You can use this window to include or exclude models within the LTI array using various criteria.

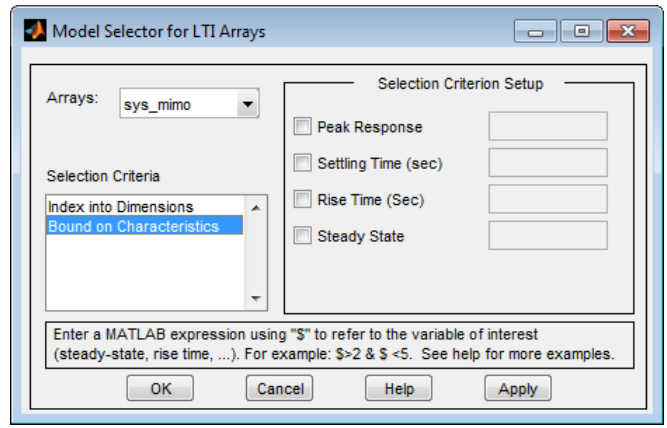
Arrays

Select the LTI array for model selection using the **Arrays** list.

Selection Criteria

There are two selection criteria. The default, **Index into Dimensions**, allows you to include or exclude specified indices of the LTI Array. Select systems from the **Selection Criterion Setup** and specify whether to show or hide the systems using the pull-down menu below the Setup lists.

The second criterion is **Bound on Characteristics**. Selecting this options causes the Model Selector to reconfigure. The reconfigured window is shown below

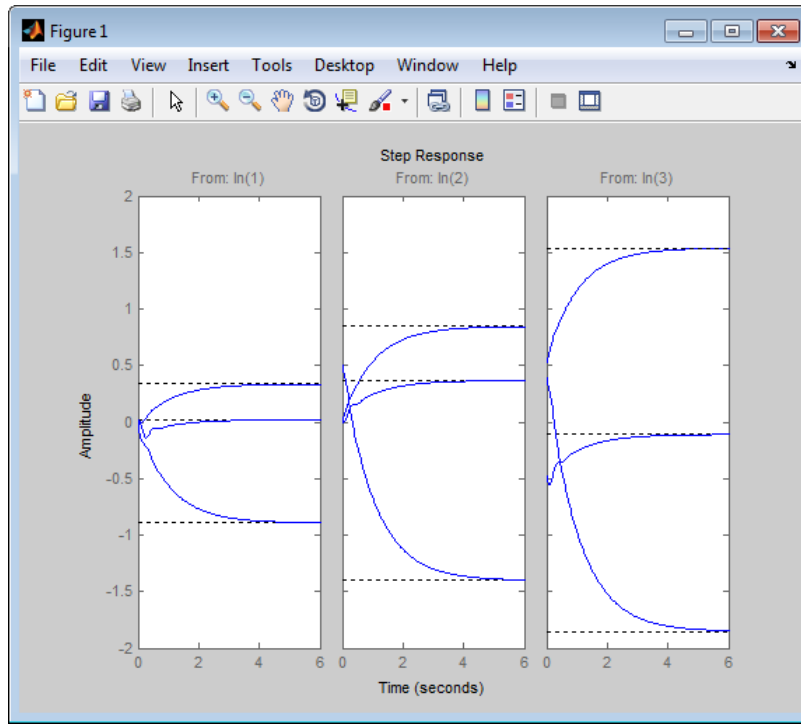


Use this option to select systems for inclusion or exclusion in your response plot based on their time response characteristics. The panel directly above the buttons describes how to set the inclusion or exclusion criteria based on which selection criteria you select from the reconfigured **Selection Criteria Setup** panel.

I/O Grouping for MIMO Models

You can group the plots by inputs, by outputs, or both by selecting **I/O Grouping** and then **Inputs**, **Outputs**, or **All**, respectively, from the right-click menu.

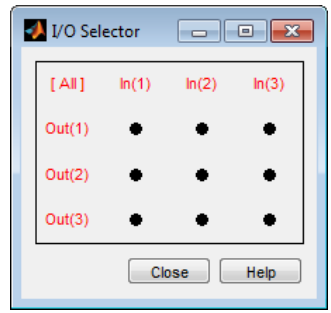
For example, if you select **Outputs**, the step plot reconfigures into 3 plots, one for each input.



Selecting **None** returns to the default configuration, where all I/O pairs are displayed individually.

Selecting I/O Pairs

Another way to organize MIMO system information is to choose **I/O Selector** from the right-click menu, which opens the **I/O Selector** window.



This window automatically configures to the number of I/O pairs in your MIMO system. You can select:

- Any individual plot (only one at a time) by clicking on a button
- Any row or column by clicking on Y(*) or U(*)
- All of the plots by clicking [all]

Using these options, you can inspect individual I/O pairs, or look at particular I/O channels in detail.

Customizing Response Plots Using the Response Plots Property Editor

In this section...

Opening the Property Editor

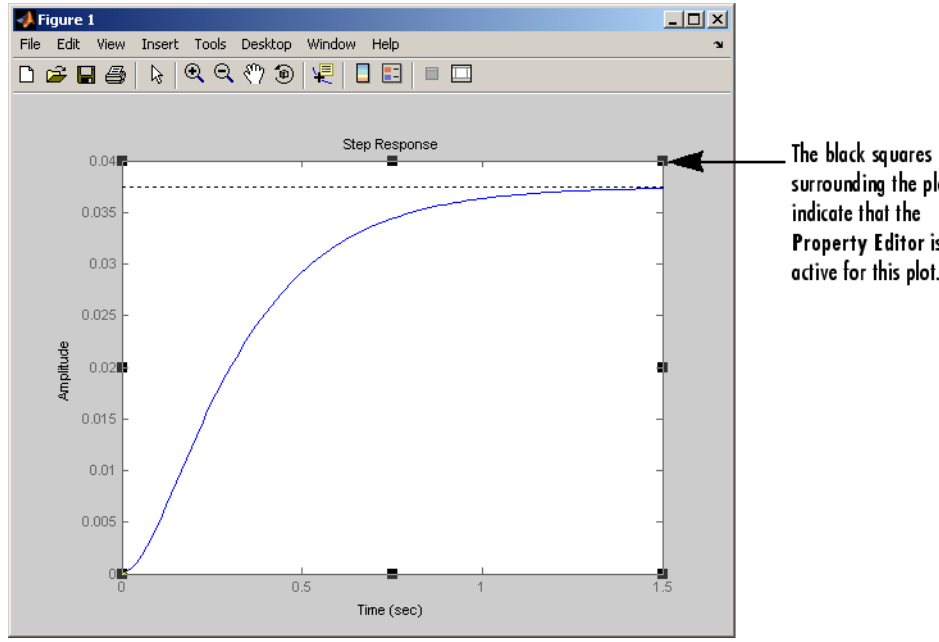
After you create a response plot, there are two ways to open the Property Editor:

- Double-click in the plot region
- Select **Properties** from the right-click menu

Before looking at the Property Editor, open a step response plot using these commands.

```
sys_dc = idtf([1 -0.8],[1 1 2 1]);
step(sys_dc)
```

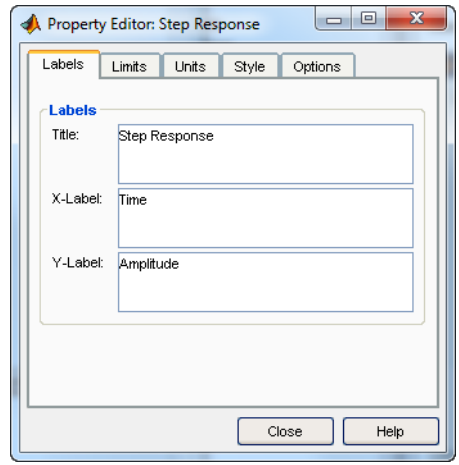
This creates a step plot. Select **Properties** from the right-click menu. Note that when you open the **Property Editor**, a set of black squares appear around the step response, as this figure shows:



SISO System Step Response

Overview of Response Plots Property Editor

This figure shows the **Property Editor** dialog box for a step response.



The Property Editor for Step Response

In general, you can change the following properties of response plots. Note that only the **Labels** and **Limits** panes are available when using the **Property Editor** with Simulink Design Optimization™ software.

- Titles and X- and Y-labels in the **Labels** pane.
- Numerical ranges of the X and Y axes in the **Limits** pane.
- Units where applicable (e.g., rad/s to Hertz) in the **Units** pane.

  If you cannot customize units, as is the case with step responses, the Property Editor will display that no units are available for the selected plot.
- Styles in the **Styles** pane.

  You can show a grid, adjust font properties, such as font size, bold and italics, and change the axes foreground color
- Change options where applicable in the **Options** pane.

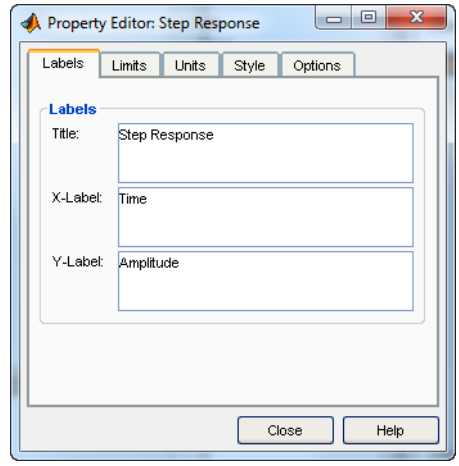
  These include peak response, settling time, phase and gain margins, etc. Plot options change with each plot response type. The Property Editor displays only the options that make sense for the selected response plot. For example, phase and gain margins are not available for step responses.

As you make changes in the Property Editor, they display immediately in the response plot. Conversely, if you make changes in a plot using right-click

menus, the Property Editor for that plot automatically updates. The Property Editor and its associated plot are dynamically linked.

Labels Pane

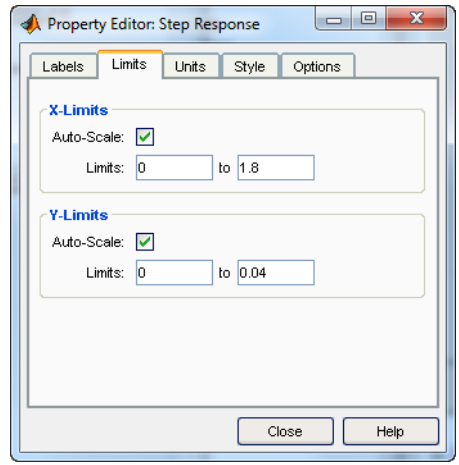
To specify new text for plot titles and axis labels, type the new string in the field next to the label you want to change. Note that the label changes immediately as you type, so you can see how the new text looks as you are typing.



Limits Pane

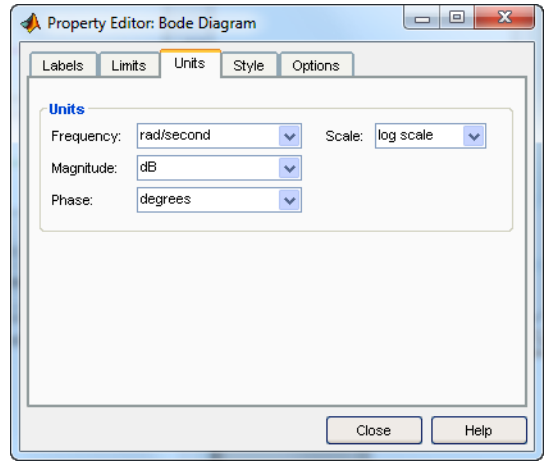
Default values for the axes limits make sure that the maximum and minimum x and y values are displayed. If you want to override the default settings, change the values in the Limits fields. The **Auto-Scale** box automatically clears if you click a different field. The new limits appear immediately in the response plot.

To reestablish the default values, select the **Auto-Scale** box again.



Units Pane

You can use the **Units** pane to change units in your response plot. The contents of this pane depend on the response plot associated with the editor.



The following table lists the options available for the response objects. Use the menus to toggle between units.

Optional Unit Conversions for Response Plots

Response Plot	Unit Conversions
Bode and Bode Magnitude	• Frequency By default, shows `rad/TimeUnit` where `TimeUnit` is the system time units specified in the `TimeUnit` property of the input system. **Frequency Units Options** - `'Hz'` - `'rad/s'` - `'rpm'` - `'kHz'` - `'MHz'` - `'GHz'` - `'rad/nanosecond'` - `'rad/microsecond'` - `'rad/millisecond'` - `'rad/minute'` - `'rad/hour'` - `'rad/day'` - `'rad/week'` - `'rad/month'` - `'rad/year'` - `'cycles/nanosecond'` - `'cycles/microsecond'` - `'cycles/millisecond'` - `'cycles/hour'` - `'cycles/day'`

Optional Unit Conversions for Response Plots (Continued)

Response Plot	Unit Conversions
	- `'cycles/week'` - `'cycles/month'` - `'cycles/year'` • Frequency scale is logarithmic or linear. • Magnitude in decibels (dB) or the absolute value • Phase in degrees or radians
Impulse	• Time. By default, shows the system time units specified in the `TimeUnit` property of the input system. **Time Units Options** - `'nanoseconds'` - `'microseconds'` - `'milliseconds'` - `'seconds'` - `'minutes'` - `'hours'` - `'days'` - `'weeks'` - `'months'` - `'years'`

Optional Unit Conversions for Response Plots (Continued)

Response Plot	Unit Conversions
Nyquist Diagram	• Frequency By default, shows `rad/TimeUnit` where `TimeUnit` is the system time units specified in the `TimeUnit` property of the input system. **Frequency Units Options** - `'Hz'` - `'rad/s'` - `'rpm'` - `'kHz'` - `'MHz'` - `'GHz'` - `'rad/nanosecond'` - `'rad/microsecond'` - `'rad/millisecond'` - `'rad/minute'` - `'rad/hour'` - `'rad/day'` - `'rad/week'` - `'rad/month'` - `'rad/year'` - `'cycles/nanosecond'` - `'cycles/microsecond'` - `'cycles/millisecond'` - `'cycles/hour'` - `'cycles/day'`

Optional Unit Conversions for Response Plots (Continued)

Response Plot	Unit Conversions
	- `'cycles/week'` - `'cycles/month'` - `'cycles/year'`
Pole/Zero Map	• Time. By default, shows the system time units specified in the `TimeUnit` property of the input system. **Time Units Options** - `'nanoseconds'` - `'microseconds'` - `'milliseconds'` - `'seconds'` - `'minutes'` - `'hours'` - `'days'` - `'weeks'` - `'months'` - `'years'` • Frequency By default, shows `rad/TimeUnit` where `TimeUnit` is the system time units specified in the `TimeUnit` property of the input system. **Frequency Units Options** - `'Hz'` - `'rad/s'`

Optional Unit Conversions for Response Plots (Continued)

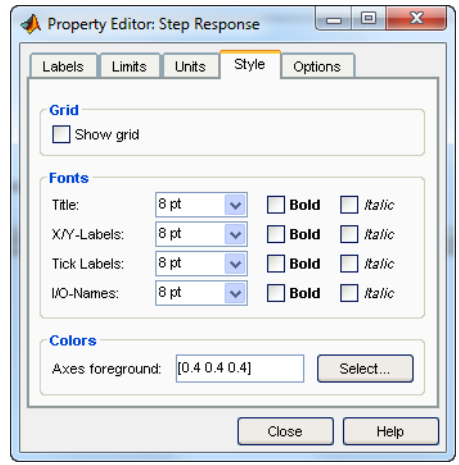
Response Plot	Unit Conversions
	- `'rpm'` - `'kHz'` - `'MHz'` - `'GHz'` - `'rad/nanosecond'` - `'rad/microsecond'` - `'rad/millisecond'` - `'rad/minute'` - `'rad/hour'` - `'rad/day'` - `'rad/week'` - `'rad/month'` - `'rad/year'` - `'cycles/nanosecond'` - `'cycles/microsecond'` - `'cycles/millisecond'` - `'cycles/hour'` - `'cycles/day'` - `'cycles/week'` - `'cycles/month'` - `'cycles/year'`
Step	• Time. By default, shows the system time units specified in the `TimeUnit` property of the input system.

Optional Unit Conversions for Response Plots (Continued)

Response Plot	Unit Conversions
	Time Units Options - `'nanoseconds'` - `'microseconds'` - `'milliseconds'` - `'seconds'` - `'minutes'` - `'hours'` - `'days'` - `'weeks'` - `'months'` - `'years'`

Style Pane

Use the Style pane to toggle grid visibility and set font preferences and axes foreground colors for response plots.



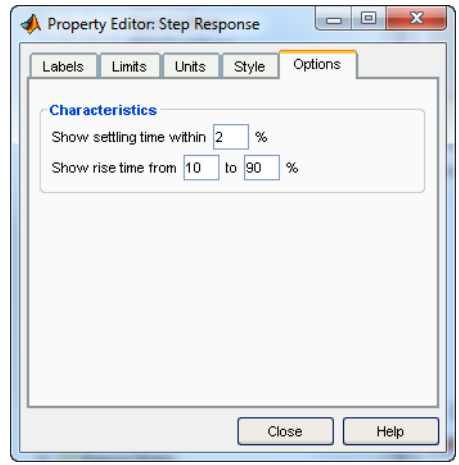
You have the following choices:

- **Grid** — Activate grids by default in new plots.
- **Fonts** — Set the font size, weight (bold), and angle (italic) for fonts used in response plot titles, X/Y-labels, tick labels, and I/O-names.
- **Colors** — Specify the color vector to use for the axes foreground, which includes the X-Y axes, grid lines, and tick labels. Use a three-element vector to represent red, green, and blue (RGB) values. Vector element values can range from 0 to 1.

  If you do not want to specify RGB values numerically, click the **Select** button to open the **Select Color** dialog box.

Options Pane

The **Options** pane allows you to customize response characteristics for plots. Each response plot has its own set of characteristics and optional settings; the table below lists them. Use the check boxes to activate the feature and the fields to specify rise or settling time percentages.



Response Characteristic Options for Response Plots

Plot	Customizable Feature
Bode Diagram and Bode Magnitude	Select lower magnitude limit Adjust phase offsets to keep phase close to a particular value, within a range of ±180°, at a given frequency. Unwrap phase (default is unwrapped)
Impulse	Show settling time within *xx*% (specify the percentage)
Nyquist Diagram	None
Pole/Zero Map	None
Step	Show settling time within *xx*% (specify the percentage) Show rise time from *xx* to *yy*% (specify the percentages)

Editing Subplots Using the Property Editor

If you create more than one plot in a single figure window, you can edit each plot individually. For example, the following code creates a figure with two plots, a step and an impulse response with two randomly selected systems:

```
subplot(2,1,1)
step(rss(2,1))
subplot(2,1,2)
impulse(rss(1,1))
```

After the figure window appears, double-click in the upper (step response) plot to activate the **Property Editor**. You will see a set of small black squares appear around the step response, indicating that it is the active plot for the editor. To switch to the lower (impulse response) plot, just click once in the impulse response plot region. The set of black squares switches to the impulse response, and the **Property Editor** updates as well.

Akaike's Criteria for Model Validation

In this section...

Definition of FPE

Akaike's Final Prediction Error (FPE) criterion provides a measure of model quality by simulating the situation where the model is tested on a different data set. After computing several different models, you can compare them using this criterion. According to Akaike's theory, the most accurate model has the smallest FPE.

Note If you use the same data set for both model estimation and validation, the fit always improves as you increase the model order and, therefore, the flexibility of the model structure.

Akaike's Final Prediction Error (FPE) is defined by the following equation:

$$FPE = V\left(\frac{1+d/N}{1-d/N}\right)$$

where V is the loss function, d is the number of estimated parameters, and N is the number of values in the estimation data set.

The toolbox assumes that the final prediction error is asymptotic for $d<<N$ and uses the following approximation to compute FPE:

$$FPE = V\left(1+2d/N\right)$$

The loss function V is defined by the following equation:

$$V = \det\left(\frac{1}{N} \sum_{1}^{N} \varepsilon\left(t, \theta_N\right) \left(\varepsilon\left(t, \theta_N\right)\right)^T \right)$$

where θ_N represents the estimated parameters.

Computing FPE

You can compute Akaike's Final Prediction Error (FPE) criterion for linear and nonlinear models.

Note FPE for nonlinear ARX models that include a tree partition nonlinearity is not supported.

To compute FPE, use the `fpe` command, as follows:

```
FPE = fpe(m1,m2,m3,...,mN)
```

According to Akaike's theory, the most accurate model has the smallest FPE.

You can also access the FPE value of an estimated model by accessing the `FPE` field of the `EstimationInfo` property of this model. For example, if you estimated the model `m`, you can access its FPE using the following command:

```
m.EstimationInfo.FPE
```

Definition of AIC

Akaike's Information Criterion (AIC) provides a measure of model quality by simulating the situation where the model is tested on a different data set. After computing several different models, you can compare them using this criterion. According to Akaike's theory, the most accurate model has the smallest AIC.

Note If you use the same data set for both model estimation and validation, the fit always improves as you increase the model order and, therefore, the flexibility of the model structure.

Akaike's Information Criterion (AIC) is defined by the following equation:

$$AIC = \log V + \frac{2d}{N}$$

where V is the loss function, d is the number of estimated parameters, and N is the number of values in the estimation data set.

The loss function V is defined by the following equation:

$$V = \det\left(\frac{1}{N} \sum_{1}^{N} \varepsilon\left(t, \theta_N\right) \left(\varepsilon\left(t, \theta_N\right)\right)^T \right)$$

where θ_N represents the estimated parameters.

For $d<<N$:

$$AIC = \log\left(V\left(1 + \frac{2d}{N}\right)\right)$$

Note *AIC* is approximately equal to log(*FPE*).

Computing AIC

Use the `aic` command to compute Akaike's Information Criterion (AIC) for one or more linear or nonlinear models, as follows:

```
AIC = aic(m1,m2,m3,...,mN)
```

According to Akaike's theory, the most accurate model has the smallest AIC.

Computing Model Uncertainty

In this section...

Why Analyze Model Uncertainty?

In addition to estimating model parameters, the toolbox algorithms also estimate variability of the model parameters that result from random disturbances in the output.

Understanding model variability helps you to understand how different your model parameters would be if you repeated the estimation using a different data set (with the same input sequence as the original data set) and the same model structure.

When validating your parametric models, check the uncertainty values. Large uncertainties in the parameters might be caused by high model orders, inadequate excitation, and poor signal-to-noise ratio in the data.

Note You can get model uncertainty data for linear parametric black-box models, and both linear and nonlinear grey-box models. Supported model objects include `idproc`, `idpoly`, `idss`, `idtf`, `idgrey`, `idfrd`, and `idnlgrey`.

What Is Model Covariance?

Uncertainty in the model is called *model covariance*.

When you estimate a model, the covariance matrix of the estimated parameters is stored with the model. Use `getcov` to fetch the covariance matrix. Use `getpvec` to fetch the list of parameters and their individual uncertainties that have been computed using the covariance matrix. The covariance matrix is used to compute all uncertainties in model output, Bode plots, residual plots, and pole-zero plots.

Computing the covariance matrix is based on the assumption that the model structure gives the correct description of the system dynamics. For models that include a disturbance model *H*, a correct uncertainty estimate assumes that the model produces white residuals. To determine whether you can trust the estimated model uncertainty values, perform residual analysis tests on your model, as described in "Residual Analysis" on page 8-24. If your model passes residual analysis tests, there is a good chance that the true system lies within the confidence interval and any parameter uncertainties results from random disturbances in the output.

For output-error models, such as transfer function models, state-space with K=0 and polynomial models of output-error form, with the noise model *H* fixed to `1`, the covariance matrix computation does not assume white residuals. Instead, the covariance is estimated based on the estimated color of the residual correlations. This estimation of the noise color is also performed for state-space models with *K*=`0`, which is equivalent to an output-error model.

Types of Model Uncertainty Information

You can view the following uncertainty information from linear and nonlinear grey-box models:

- Uncertainties of estimated parameters.

  Type `present(model)` at the prompt, where `model` represents the name of a linear or nonlinear model.

- Confidence intervals on the linear model plots, including step-response, impulse-response, Bode, Nyquist, noise spectrum and pole-zero plots.

  Confidence intervals are computed based on the variability in the model parameters. For information about displaying confidence intervals, see the corresponding plot section.

- Covariance matrix of the estimated parameters in linear models and nonlinear grey-box models.

  Use `getcov`.

- Estimated standard deviations of polynomial coefficients, poles/zeros, or state-space matrices using `idssdata`, `tfdata`, `zpkdata`, and `polydata`.

- Simulated output values for linear models with standard deviations using the `sim` command.

  Call the `sim` command with output arguments, where the second output argument is the estimated standard deviation of each output value. For example, type `[ysim,ysimsd]=sim(model,data)`, where `ysim` is the simulated output, `ysimsd` contains the standard deviations on the simulated output, and `data` is the simulation data.

- To perform Monte-Carlo analysis, use `rsample` to generate a random sampling of an identified model in a given confidence region. An array of identified systems of the same structure as the input system is returned. The parameters of the returned models are perturbed about their nominal values in a way that is consistent with the parameter covariance.

- To simulate the effect of parameter uncertainties on a model's response, use `simsd`.

Troubleshooting Models

In this section...

About Troubleshooting Models

During validation, you might find that your model output fits the validation data poorly. You might also find some unexpected or undesirable model characteristics.

If the tips suggested in these sections do not help improve your models, then a good model might not be possible for this data. For example, your data might have poor signal-to-noise ratio, large and nonstationary disturbances, or varying system properties.

Model Order Is Too High or Too Low

When the Model Output plot does not show a good fit, there is a good chance that you need to try a different model order. System identification is largely a trial-and-error process when selecting model structure and model order. Ideally, you want the lowest-order model that adequately captures the system dynamics.

You can estimate the model order as described in "Preliminary Step – Estimating Model Orders and Input Delays" on page 3-53. Typically, you use the suggested order as a starting point to estimate the lowest possible order with different model structures. After each estimation, you monitor the Model Output and the Residual Analysis plots, and then adjust your settings for the next estimation.

When a low-order model fits the validation data poorly, try estimating a higher-order model to see if the fit improves. For example, if a Model Output plot shows that a fourth-order model gives poor results, try estimating an eighth-order model. When a higher-order model improves the fit, you can conclude that higher-order models might be required and linear models might be sufficient.

You should use an independent data set to validate your models. If you use the same data set to both estimate and validate a model, the fit always improves as you increase model order, and you risk overfitting. However, if you use an independent data set to validate your models, the fit eventually deteriorates if your model orders are too high.

High-order models are more expensive to compute and result in greater parameter uncertainty.

Nonlinearity Estimator Produces a Poor Fit

In the case of nonlinear ARX and Hammerstein-Wiener models, the Model Output plot does not show a good fit when the nonlinearity estimator has incorrect complexity.

You specify the complexity of piece-wise-linear, wavelet, sigmoid, and custom networks using the number of units (`NumberOfUnits` nonlinear estimator property). A high number of units indicates a complex nonlinearity estimator. In the case of neural networks, you specify the complexity using the parameters of the network object. For more information, see the Neural Network Toolbox documentation.

To select the appropriate complexity of the nonlinearity estimator, start with a low complexity and validate the model output. Next, increate the complexity and validate the model output again. The model fit degrades when the nonlinearity estimator becomes too complex.

Note To see the model fit degrade when the nonlinearity estimator becomes too complex, you must use an independent data set to validate the data that is different from the estimation data set.

Substantial Noise in the System

There are a couple of indications that you might have substantial noise in your system and might need to use linear model structures that are better equipped to model noise.

One indication of noise is when a state-space model is better than an ARX model at reproducing the measured output; whereas the state-space structure has sufficient flexibility to model noise, the ARX model structure is less able to model noise because the *A* polynomial must account for both the system dynamics and the noise. The following equation represents the ARX model and shows that *A* couples the dynamics and the noise by appearing in the denominator of both the dynamics term and the noise terms:

$$y = \frac{B}{A}u + \frac{1}{A}e$$

Another indication that a noise model is needed appears in residual analysis plots when you see significant autocorrelation of residuals at nonzero lags. For more information about residual analysis, see "Residual Analysis" on page 8-24.

To model noise more carefully, use the ARMAX or the Box-Jenkins model structure, where the dynamics term and the noise term are modeled by different polynomials.

Unstable Models

Unstable Linear Model

You can test whether a *linear model* is unstable is by examining the pole-zero plot of the model, which is described in "Pole and Zero Plots" on page 8-59. The stability threshold for pole values differs for discrete-time and continuous-time models, as follows:

- For stable continuous-time models, the real part of the pole is less than 0.
- For stable discrete-time models, the magnitude of the pole is less than 1.

Note Linear trends might cause linear models to be unstable. However, detrending the model does not guarantee stability.

When an unstable model is OK: In some cases, an unstable model is still a useful model. For example, your system might be unstable without a controller, and you plan to use your model for control design. In this case, you can import your unstable model into Simulink or Control System Toolbox products.

Forcing stability during estimation: If you believe that your system is stable, but your model is unstable, then you can estimate the model with the `Focus` estimation option set to `Stability`. This setting might result in a reduced model quality. For more information about `Focus`, see the various estimation option configuration commands such as `tfestOptions`, `ssestOptions`, `procestOptions` etc..

Allowing for some instability: A more advanced approach to achieving a stable model is by setting the stability threshold estimation option to allow a margin of error. The threshold estimation options are advanced properties of an estimation option set.

- For continuous-time models, set the value of `opt.Advanced.StabilityThreshold.s`. The model is considered stable if the pole on the far right is to the left of *s* stability threshold.
- For discrete-time models, set the value of `opt.Advanced.StabilityThreshold.z`. The model is considered stable if all poles inside the circle centered at the origin and with a radius of equal to the *z* stability threshold.

Unstable Nonlinear Models

To test if a *nonlinear model* is unstable is to plot the simulated model output on top of the validation data. If the simulated output diverges from measured output, the model is unstable. However, agreement between model output and measured output does not guarantee stability.

Missing Input Variables

If the Model Output plot and Residual Analysis plot shows a poor fit and you have already tried different structures and orders and modeled noise, it might be that there are one or more missing inputs that have a significant effect on the output.

Try including other measured signals in your input data, and then estimating the models again.

Inputs need not be control signals. Any measurable signal can be considered an input, including measurable disturbances.

Complicated Nonlinearities

If the Model Output plot and Residual Analysis plot shows a poor fit, consider if nonlinear effects are present in the system.

You can model the nonlinearities by performing a simple transformation on the signals to make the problem linear in the new variables. For example, if electrical power is the driving stimulus in a heating process and temperature is the output, you can form a simple product of voltage and current measurements.

If your problem is sufficiently complex and you do not have physical insight into the problem, you might try fitting nonlinear black-box models.

Next Steps After Getting an Accurate Model

For linear parametric models, you can perform the following operations:

- Transform between continuous-time and discrete-time representation.

  See "Transforming Between Discrete-Time and Continuous-Time Representations" on page 3-139.

- Transform between linear model representations, such as between polynomial, state-space, and zero-pole representations.

  See "Transforming Between Linear Model Representations" on page 3-156.

- Extract numerical data from transfer functions, pole-zero models, and state-space matrices.

  See "Extracting Numerical Model Data" on page 3-136.

For nonlinear black-box models (`idnlarx` and `idnlhw`objects), you can compute a linear approximation of the nonlinear model. See "Linear Approximation of Nonlinear Black-Box Models" on page 4-79.

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.

If you have the Control System Toolbox software installed, you can import your linear plant model for control-system design. For more information, see "Using Identified Models for Control Design Applications" on page 10-2.

Finally, if you have Simulink software installed, you can exchange data between the System Identification Toolbox software and the Simulink environment. For more information, see "Simulating Identified Model Output in Simulink" on page 11-5.

Spectrum Estimation Using Complex Data - Marple's Test Case

This example shows how to perform spectral estimation on time series data. We use Marple's test case (The complex data in L. Marple: S.L. Marple, Jr, Digital Spectral Analysis with Applications, Prentice-Hall, Englewood Cliffs, NJ 1987.)

Test Data

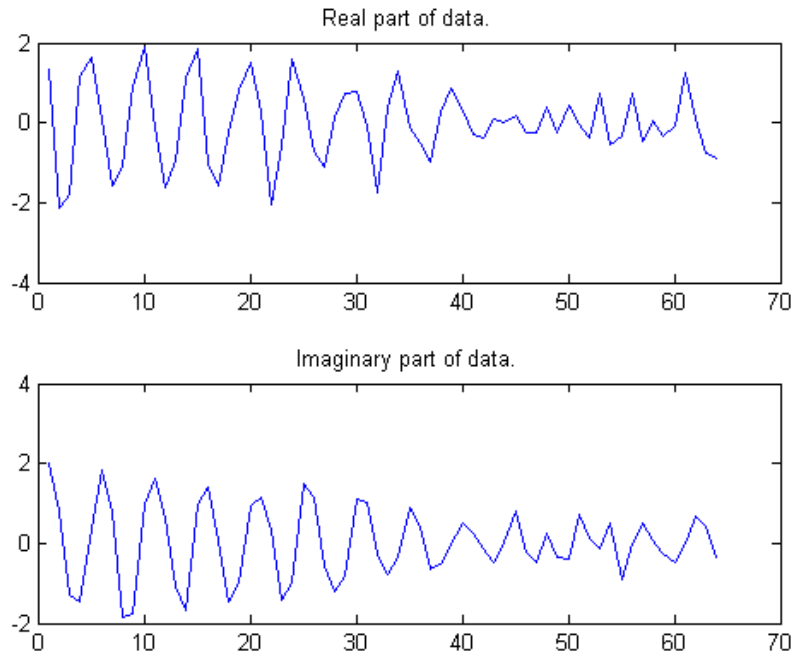
Let us begin by loading the test data:

```
load marple
```

Most of the routines in System Identification Toolbox™ support complex data. For plotting we examine the real and imaginary parts of the data separately, however.

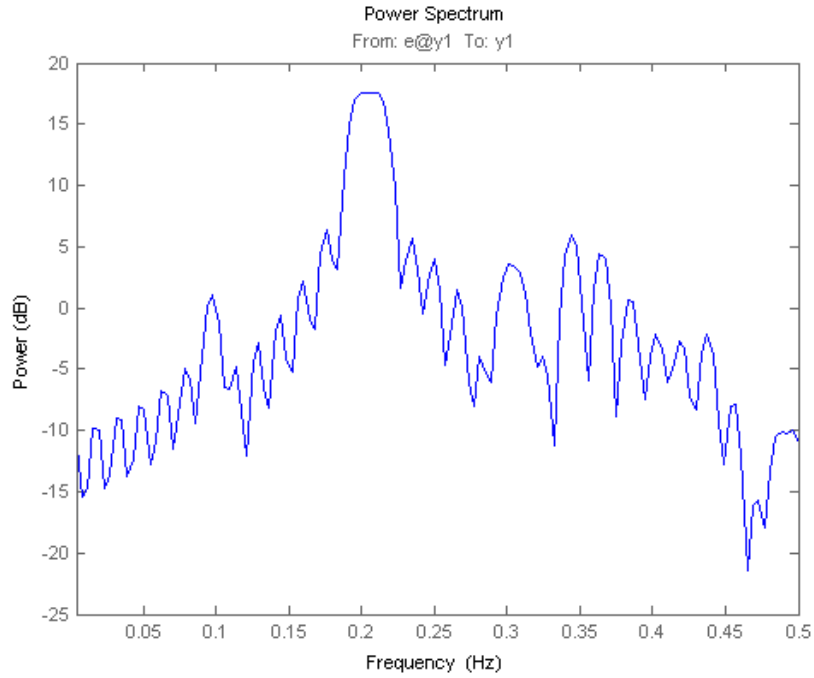
First, take a look at the data:

```
subplot(211),plot(real(marple)),title('Real part of data.')
subplot(212),plot(imag(marple)),title('Imaginary part of data.')
```



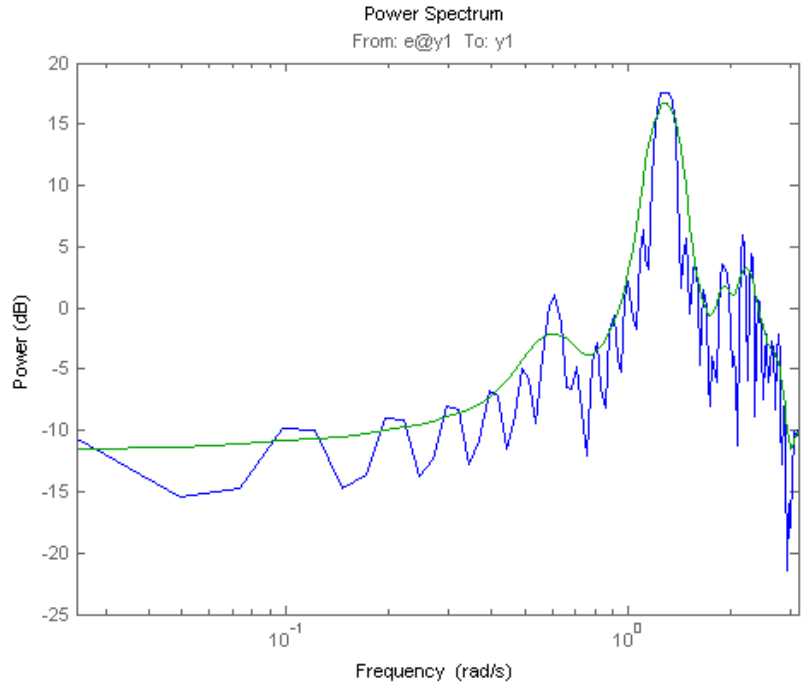
As a preliminary analysis step, let us check the periodogram of the data:

```
per = etfe(marple);
w = per.Frequency;
clf
h = spectrumplot(per,w);
opt = getoptions(h);
opt.FreqScale = 'linear';
opt.FreqUnits = 'Hz';
setoptions(h,opt)
```



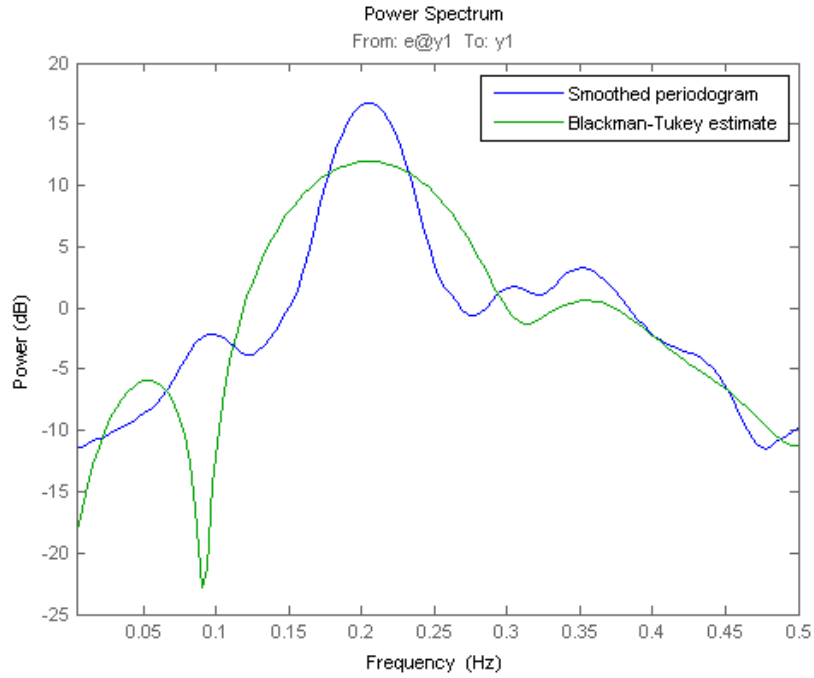
Since the data record is only 64 samples, and the periodogram is computed for 128 frequencies, we clearly see the oscillations from the narrow frequency window. We therefore apply some smoothing to the periodogram (corresponding to a frequency resolution of 1/32 Hz):

```
sp = etfe(marple,32);
spectrumplot(per,sp,w);
```



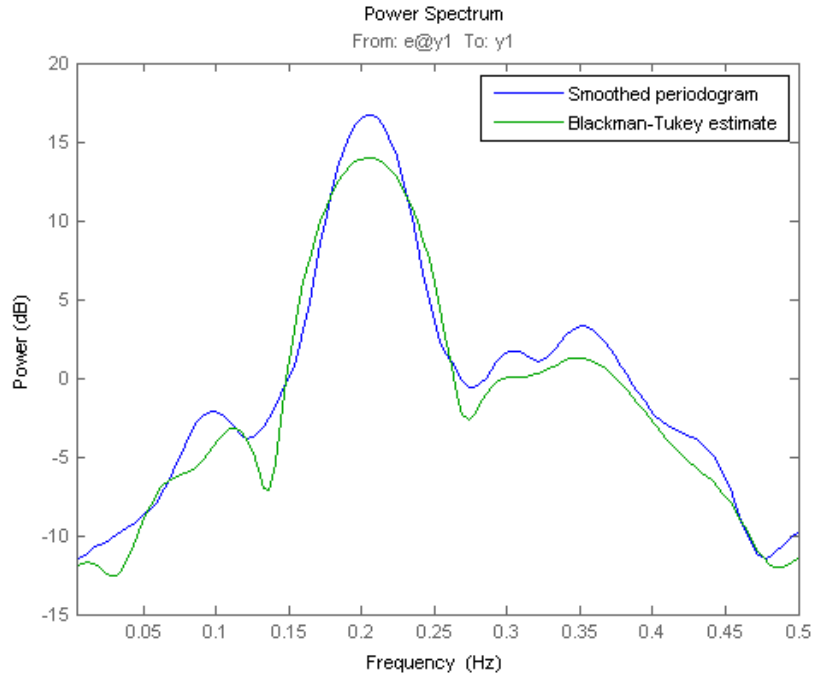
Let us now try the Blackman-Tukey approach to spectrum estimation:

```
ssm = spa(marple); % Function spa performs spectral estimation
spectrumplot(sp,'b',ssm,'g',w,opt);
legend({'Smoothed periodogram','Blackman-Tukey estimate'});
```



The default window length gives a very narrow lag window for this small amount of data. We can choose a larger lag window by:

```
ss20 = spa(marple,20);
spectrumplot(sp,'b',ss20,'g',w,opt);
legend({'Smoothed periodogram','Blackman-Tukey estimate'});
```



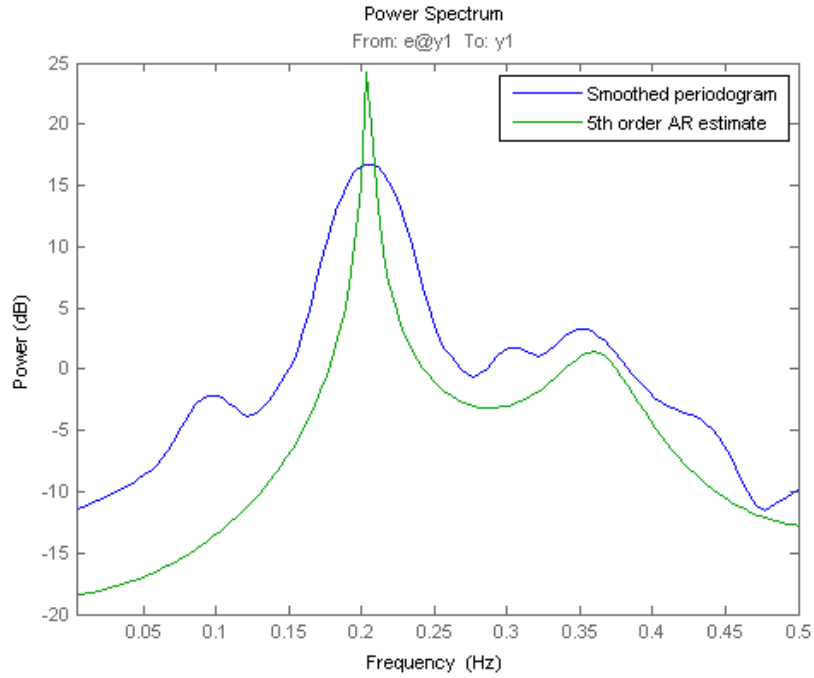
Estimating an Autoregressive (AR) Model

A parametric 5-order AR-model is computed by:

```
t5 = ar(marple,5);
```

Compare with the periodogram estimate:

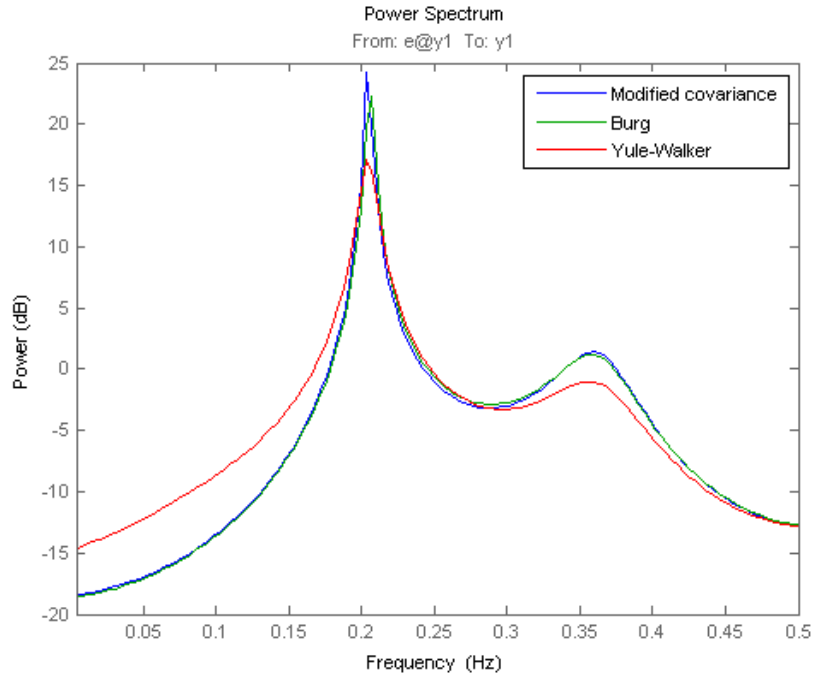
```
spectrumplot(sp,'b',t5,'g',w,opt);
legend({'Smoothed periodogram','5th order AR estimate'});
```



The AR-command in fact covers 20 different methods for spectrum estimation. The above one was what is known as 'the modified covariance estimate' in Marple's book.

Some other well known ones are obtained with:

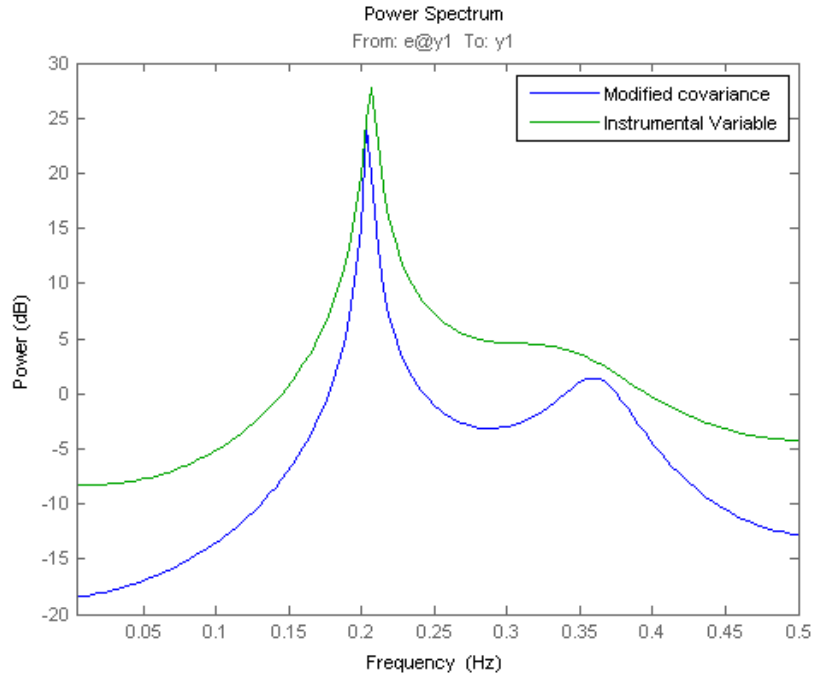
```
tb5 = ar(marple,5,'burg');      % Burg's method
ty5 = ar(marple,5,'yw');        % The Yule-Walker method
spectrumplot(t5,tb5,ty5,w,opt);
legend({'Modified covariance','Burg','Yule-Walker'})
```



Estimating AR Model using Instrumental Variable Approach

AR-modeling can also be done using the Instrumental Variable approach. For this, we use the function `ivar`:

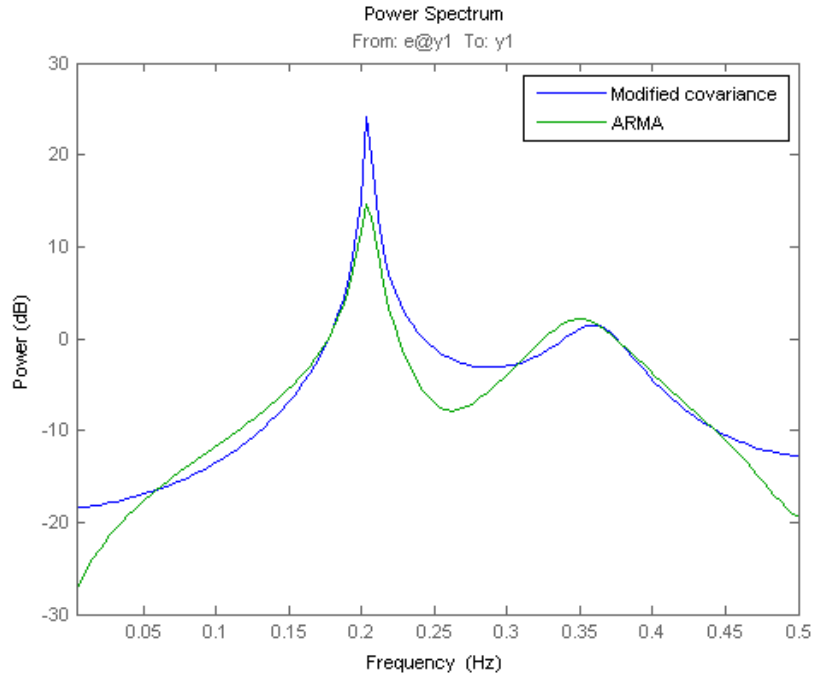
```
ti = ivar(marple,4);
spectrumplot(t5,ti,w,opt);
legend({'Modified covariance','Instrumental Variable'})
```



Autoregressive-Moving Average (ARMA) Model of the Spectra

Furthermore, System Identification Toolbox covers ARMA-modeling of spectra:

```
ta44 = armax(marple,[4 4]); % 4 AR-parameters and 4 MA-parameters
spectrumplot(t5,ta44,w,opt);
legend({'Modified covariance','ARMA'})
```



Additional Information

For more information on identification of dynamic systems with System Identification Toolbox visit the System Identification Toolbox product information page.

9

Setting Toolbox Preferences

Toolbox Preferences Editor

In this section...

Overview of the Toolbox Preferences Editor

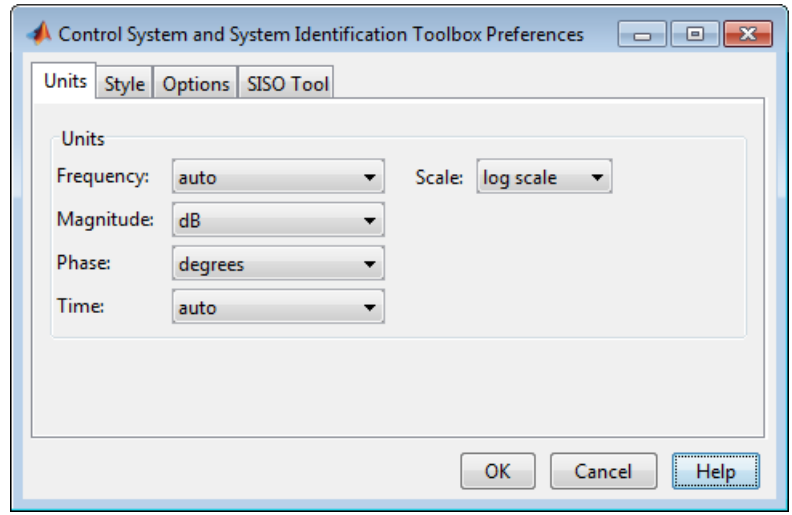
The Toolbox Preferences editor allows you to set plot preferences that will persist from session to session.

Opening the Toolbox Preferences Editor

To open the Toolbox Preferences editor, select **Toolbox Preferences** from the **File** menu of the LTI Viewer or the SISO Design Tool. Alternatively, you can type

```
identpref
```

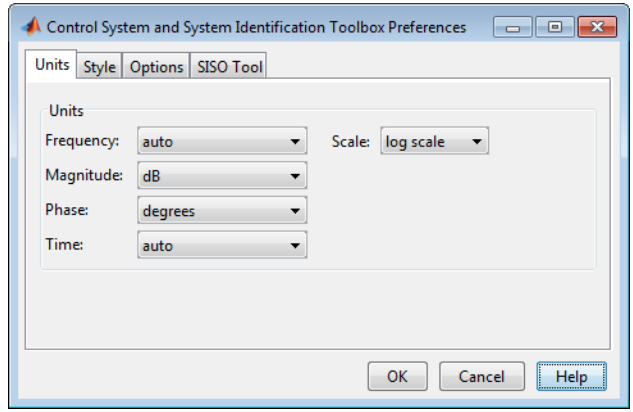
at the MATLAB prompt.



Control System Toolbox™ Preferences Editor

Note The SISO Design Tool requires the Control System Toolbox software.

Units Pane



Use the **Units** pane to set preferences for the following:

- **Frequency**

  The default `auto` option uses `rad/TimeUnit` as the frequency units relative to the system time units, where `TimeUnit` is the system time units specified in the `TimeUnit` property of the system on frequency-domain plots. For multiple systems with different time units, the units of the first system is used.

  For the frequency axis, you can select logarithmic or linear scales.

  ### Other Frequency Units Options

  - `'Hz'`
  - `'rad/s'`
  - `'rpm'`
  - `'kHz'`
  - `'MHz'`
  - `'GHz'`
  - `'rad/nanosecond'`
  - `'rad/microsecond'`

- `'rad/millisecond'`
- `'rad/minute'`
- `'rad/hour'`
- `'rad/day'`
- `'rad/week'`
- `'rad/month'`
- `'rad/year'`
- `'cycles/nanosecond'`
- `'cycles/microsecond'`
- `'cycles/millisecond'`
- `'cycles/hour'`
- `'cycles/day'`
- `'cycles/week'`
- `'cycles/month'`
- `'cycles/year'`

- **Magnitude** — Decibels (dB) or absolute value (abs)
- **Phase** — Degrees or radians
- **Time**

  The default `auto` option uses the time units specified in the `TimeUnit` property of the system on the time- and frequency-domain plots. For multiple systems with different time units, the units of the first system is used.

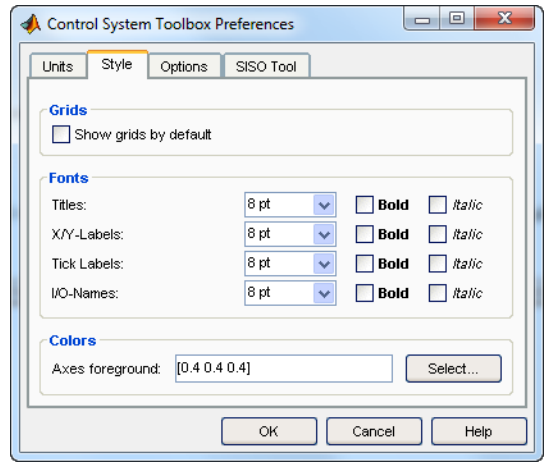
  ### Other Time Units Options

  - `'nanoseconds'`
  - `'microseconds'`
  - `'milliseconds'`
  - `'seconds'`

- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Style Pane

Use the **Style** pane to toggle grid visibility and set font preferences and axes foreground colors for all plots you create. This figure shows the Style pane.



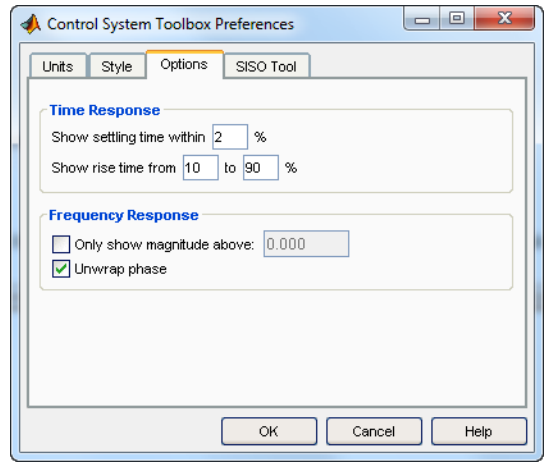
You have the following choices:

- **Grid** — Activate grids by default in new plots.
- **Fonts** — Set the font size, weight (bold), and angle (italic).
- **Colors** — Specify the color vector to use for the axes foreground, which includes the X-Y axes, grid lines, and tick labels. Use a three-element vector to represent red, green, and blue (RGB) values. Vector element values can range from 0 to 1.

  If you do not want to specify RGB values numerically, click the **Select** button to open the **Select Colors** dialog box.

Options Pane

The Options pane has selections for time responses and frequency responses. This figure shows the Options pane with default settings.

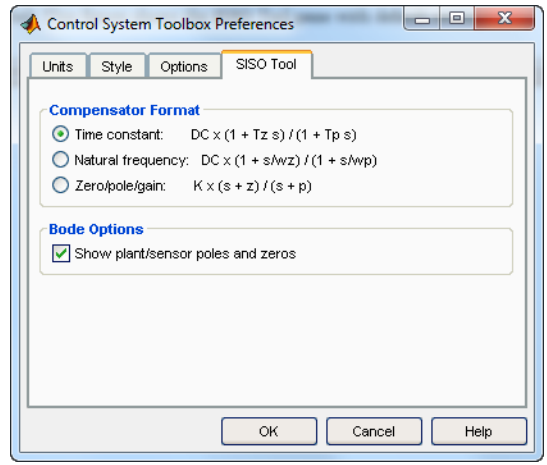


The following are the available options for the Options pane:

- **Time Response**:
  - Show settling time within *xx*%— You can set the threshold of the settling time calculation to any percentage from 0 to 100%. The default is 2%.
  - Specify rise time from *xx*% to *yy*%— The standard definition of rise time is the time it takes the signal to go from 10% to 90% of the final value. You can choose any percentages you like (from 0% to 100%), provided that the first value is smaller than the second.
- **Frequency Response:**
  - Only show magnitude above *xx*—Specify a lower limit for magnitude values in response plots so that you can focus on a region of interest.
  - Unwrap phase—By default, the phase is unwrapped. Wrap the phrase by clearing this box. If the phase is wrapped, all phase values are shifted such that their equivalent value displays in the range [-180°, 180°).

SISO Tool Pane

The SISO Tool pane has settings for the SISO Design Tool. This figure shows the SISO Tool pane with default settings.



You can make the following selections:

- **Compensator Format** — Select the time constant, natural frequency, or zero/pole/gain format. The time constant format is a factorization of the compensator transfer function of the form

$$DC \times \frac{(1+Tz_1 s)}{(1+Tp_1 s)} \cdots$$

where DC is compensator DC gain, Tz_1, Tz_2, ..., are the zero time constants, and Tp_1, Tp_2, ..., are the pole time constants.

The natural frequency format is

$$DC \times \frac{\left(1+s/\omega_{z_1}\right)}{\left(1+s/\omega_{p_1}\right)} \cdots$$

where *DC* is compensator DC gain, ω_{z1}, and ω_{z2}, ... and ω_{p1}, ω_{p2}, ..., are the natural frequencies of the zeros and poles, respectively.

The zero/pole/gain format is

$$K \times \frac{(s + z_1)}{(s + p_1)}$$

where K is the overall compensator gain, and z_1, z_2, ... and p_1, p_2, ..., are the zero and pole locations, respectively.

- **Bode Options** — By default, the SISO Design Tool shows the plant and sensor poles and zeros as blue x's and o's, respectively. Clear this box to eliminate the plant's poles and zeros from the Bode plot. Note that the compensator poles and zeros (in red) will still appear.

10

Control Design Applications

Using Identified Models for Control Design Applications

In this section...

How Control System Toolbox Software Works with Identified Models

System Identification Toolbox software integrates with Control System Toolbox software by providing a plant for control design.

Control System Toolbox software also provides the LTI Viewer GUI to extend System Identification Toolbox functionality for linear model analysis.

Control System Toolbox software supports only linear models. If you identified a nonlinear plant model using System Identification Toolbox software, you must linearize it before you can work with this model in the Control System Toolbox software. For more information, see the `linapp`, `linearize(idnlarx)`, or `linearize(idnlhw)` reference page.

Note You can only use the System Identification Toolbox software to linearize nonlinear ARX (`idnlarx`) and Hammerstein-Wiener (`idnlhw`) models. Linearization of nonlinear grey-box (`idnlgrey`) models is not supported.

Using balred to Reduce Model Order

In some cases, the order of your identified model might be higher than necessary to capture the dynamics. If you have the Control System Toolbox

software, you can use `balred` to compute a state-spate model approximation with a reduced model order.

For more information, see `balred`.

To learn how you can reduce model order using pole-zero plots, see "Reducing Model Order Using Pole-Zero Plots" on page 8-61.

Compensator Design Using Control System Toolbox Software

After you estimate a plant model using System Identification Toolbox software, you can use Control System Toolbox software to design a controller for this plant.

System Identification Toolbox models in the MATLAB workspace are immediately available to Control System Toolbox commands. However, if you used the System Identification Tool GUI to estimate models, you must first export the models to the MATLAB workspace. To export a model from the GUI, drag the model icon to the **To Workspace** rectangle.

Control System Toolbox software provides both the SISO Design Tool GUI and commands for working at the command line. You can import linear models directly into SISO Design Tool using the following command:

```
sisotool(model)
```

Converting Models to LTI Objects

You can convert linear identified models into numeric LTI models (`ss`, `tf`, `zpk`) of Control System Toolbox.

The following table summarizes the commands for transforming linear state-space and polynomial models to an LTI object.

Commands for Converting Models to LTI Objects

Command	Description	Example
`frd`	Convert to frequency-response representation.	`ss_sys = frd(model)`
`ss`	Convert to state-space representation.	`ss_sys = ss(model)`
`tf`	Convert to transfer-function form.	`tf_sys = tf(model)`
`zpk`	Convert to zero-pole form.	`zpk_sys = zpk(model)`

The following code converts the noise component of a linear identified model, `sys`, to a numeric state-space model:

```
noise_model_ss = idss(sys,'noise');
```

To convert both the measured and noise components of a linear identified model, `sys`, to a numeric state-space model:

```
model_ss = idss(sys,'augmented');
```

For more information about subreferencing the dynamic or the noise model, see "Separation of Measured and Noise Components of Models" on page 3-161.

Viewing Model Response Using the LTI Viewer

What Is the LTI Viewer?

If you have the Control System Toolbox software, you can plot models in the LTI Viewer from either the System Identification Tool GUI or the MATLAB Command Window.

The LTI Viewer is a graphical user interface for viewing and manipulating the response plots of linear models.

Note The LTI Viewer does not display model uncertainty.

For more information about working with plots in the LTI Viewer, see the Control System Toolbox documentation.

Displaying Identified Models in the LTI Viewer

When the MATLAB software is installed, the System Identification Tool GUI contains the **To LTI Viewer** rectangle. To plot models in the LTI Viewer, drag and drop the corresponding icon to the **To LTI Viewer** rectangle in the System Identification Tool GUI.

Alternatively, use the following syntax when working at the command line to view a model in the LTI Viewer:

```
view(model)
```

Combining Model Objects

If you have the Control System Toolbox software, you can combine linear model objects, such as `idtf`, `idgrey`, `idpoly`, `idproc`, and `idss` model objects, similar to the way you combine LTI objects. The result of these operations is a numeric LTI model that belongs to the Control System Toolbox software. The only exceptions are the model stacking and model concatenation operations, which deliver results as identified models.

For example, you can perform the following operations on identified models:

- `G1+G2`
- `G1*G2`
- `append(G1,G2)`
- `feedback(G1,G2)`

Using System Identification Toolbox Software with Control System Toolbox Software

This example shows how to create and plot models using the System Identification Toolbox software and Control System Toolbox software.

Construct a random numeric model using the Control System Toolbox software.

```
rng('default');
sys0 = drss(4,3,2);
```

`rng('default')` specifies the setting of the random number generator as its default setting.

`sys0` is a fourth-order numeric state-space model with three outputs and two inputs.

Convert `sys0` to an identified state-space model and set its output noise variance.

```
sys = idss(sys0);
sys.NoiseVariance = 0.1*eye(3);
```

Generate input data for simulating the output.

```
u = iddata([],idinput([800 2],'rbs'));
```

Simulate the model output with added noise.

```
opt = simOptions('AddNoise',true);
y = sim(sys,u,opt);
```

`sim` requires System Identification Toolbox software.

`opt` is an option set specifying simulation options.

`y` is the simulated output for `sys`.

Create an input-output (`iddata`) object.

```
data = [y u];
```

Estimate the state-space model from the generated data using `ssest`.

```
estimated_ss = ssest(data(1:400));
```

`ssest` requires System Identification Toolbox software.

`estimated_ss` is an identified state-space model.

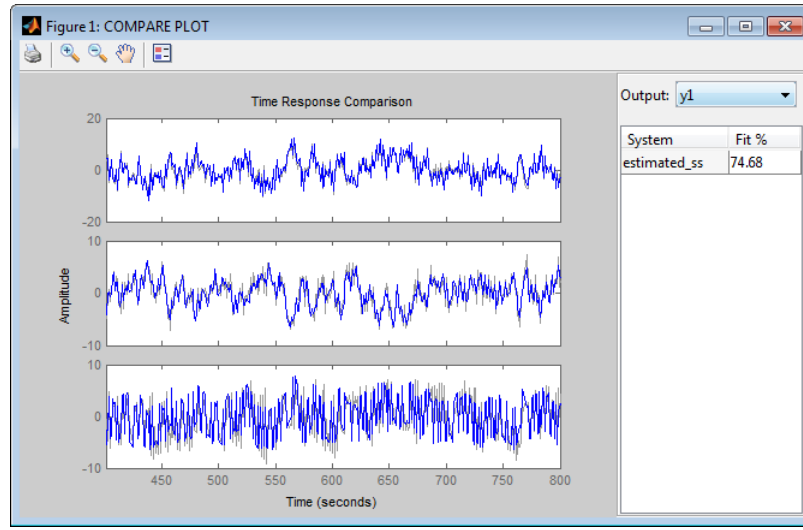
Convert the identified state-space model to a numeric transfer function

```
sys_tf = tf(estimated_ss);
```

`tf` requires Control System Toolbox software.

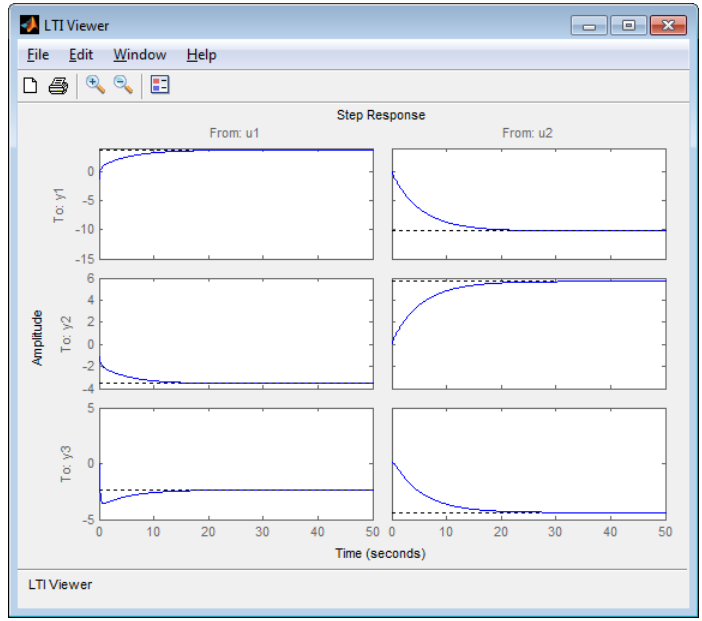
Plot the model output for identified state-space model.

```
compare(data(401:800),estimated_ss)
```



Plot the response of identified model using the LTI Viewer.

```
view(estimated_ss);
```


LTI Viewer
File Edit Window Help
Step Response
From: u1
From: u2
To: y1
To: y2
To: y3
Amplitude
Time (seconds)
LTI Viewer

11

System Identification Toolbox Blocks

Using System Identification Toolbox Blocks in Simulink Models

System Identification Toolbox software provides blocks for sharing information between the MATLAB and Simulink environments.

You can use the System Identification Toolbox block library to perform the following tasks:

- Stream time-domain data source (`iddata` object) into a Simulink model.
- Export data from a simulation in Simulink software as a System Identification Toolbox data object (`iddata` object).
- Import estimated models into a Simulink model, and simulate the models with or without noise.

  The model you import might be a component of a larger system modeled in Simulink. For example, if you identified a plant model using the System Identification Toolbox software, you can import this plant into a Simulink model for control design.
- Estimate parameters of linear polynomial models during simulation from single-output data.

To open the System Identification Toolbox block library, on the **Home** tab, in the **Simulink** section, click **Simulink Library**. In the Library Browser, select **System Identification Toolbox**.

You can also open the System Identification Toolbox block library directly by typing the following command at the MATLAB prompt:

```
slident
```

To get help on a block, right-click the block in the Library Browser, and select **Help**.

Preparing Data

The following table summarizes the blocks you use to transfer data between the MATLAB and Simulink environments.

After you add a block to the Simulink model, double-click the block to specify block parameters. For an example of bringing data into a Simulink model, see the tutorial on estimating process models in the *System Identification Toolbox Getting Started Guide*.

Block	Description
Iddata Sink	Export input and output signals to the MATLAB workspace as an `iddata` object.
Iddata Source	Import `iddata` object from the MATLAB workspace. Input and output ports of the block correspond to input and output signals of the data. These inputs and outputs provide signals to blocks that are connected to this data block.

For information about configuring each block, see the corresponding reference pages.

Identifying Linear Models

The following table summarizes the blocks you use to estimate model parameters in a Simulink model during simulation and export the results to the MATLAB environment.

After you add a block to the model, double-click the block to specify block parameters.

Block	Description
AutoRegressive model estimator	Estimate AR model parameters from time-series data, which has one output and no input.
AutoRegressive Moving Average with eXternal input model estimator	Estimate ARMAX model parameters from input/output data.
AutoRegressive with eXternal input model estimator	Estimate ARX model parameters from input/output data.
Box-Jenkins model estimator	Estimate BJ model parameters from input/output data.
Output-error model estimator	Estimate OE model parameters from input/output data.
General model estimator using Predictive Error Method	Estimate ARX, ARMAX, Box-Jenkins, and Output-Error models (`idpoly` objects) from single-input and single output data using general prediction-error method.

For information about configuring each block, see the corresponding reference pages.

Simulating Identified Model Output in Simulink

In this section...

When to Use Simulation Blocks

Add model simulation blocks to your Simulink model from the System Identification Toolbox block library when you want to:

- Represent the dynamics of a physical component in a Simulink model using a data-based nonlinear model.
- Replace a complex Simulink subsystem with a simpler data-based nonlinear model.

You use the model simulation blocks to import the models you identified using System Identification Toolbox software from the MATLAB workspace into the Simulink environment. For a list of System Identification Toolbox simulation blocks, see "Summary of Simulation Blocks" on page 11-5.

Summary of Simulation Blocks

The following table summarizes the blocks you use to import models from the MATLAB environment into a Simulink model for simulation. Importing a model corresponds to entering the model variable name in the block parameter dialog box.

Block	Description
Idmodel	Simulate a linear identified model in Simulink, including process (`idproc`), linear polynomial (`idpoly`), state-space (`idss`), grey-box (`idgrey`) and transfer-function (`idtf`) models.
Nonlinear ARX Model	Simulate `idnlarx` model in Simulink.

Block	Description
Hammerstein-Wiener Model	Simulate `idnlhw` model in Simulink.
Nonlinear Grey-Box Model	Simulate nonlinear ODE (`idnlgrey` model object) in Simulink.

After you import the model into Simulink software, use the block parameter dialog box to specify the initial conditions for simulating that block. (See "Specifying Initial Conditions for Simulation" on page 11-6.) For information about configuring each block, see the corresponding reference pages.

Specifying Initial Conditions for Simulation

For accurate simulation of a linear or a nonlinear model, you can use default initial conditions or specify the initial conditions for simulation using the block parameters dialog box.

Specifying Initial States of Linear Models

For `idss` and `idgrey` models, specify the initial states for simulation in the **Initial state** field of the Function Block Parameters: Idmodel dialog box:

- To specify the initial states values as zero, use `'z'`.
- To use the initial states values stored in the `X0` property of the model, use `'m'`.
- To match the simulated response of the model to a certain input/output data set, first use `findstates(idParameteric)` to estimate initial states values that maximize the fit. Specify the estimated initial states values in the **Initial state** field.

For `idpoly`, `idtf` and `idproc` models, the default initial states values are zero. If you want to specify different values, such as maximize fit to a given

output data, convert the model to `idss` object, and then specify its initial states as described previously. For example, for the following `idpoly` model:

```
m1=idpoly([1 2 1],[2 2]);
```

the initial states correspond to those of the equivalent state-space model:

```
m2=idss(m1);
```

Specifying Initial States of Nonlinear ARX Models

The states of a nonlinear ARX model correspond to the dynamic elements of the nonlinear ARX model structure, which are the model regressors. *Regressors* can be the delayed input/output variables (standard regressors) or user-defined transformations of delayed input/output variables (custom regressors). For more information about the states of a nonlinear ARX model, see the `idnlarx` reference page.

For simulating nonlinear ARX models, you can specify the initial conditions as input/output values, or as a vector. For more information about specifying initial conditions for simulation, see the IDNLARX Model reference page.

Specifying Initial States of Hammerstein-Wiener Models

The states of a Hammerstein-Wiener model correspond to the states of the embedded linear (`idpoly` or `idss`) model. For more information about the states of a Hammerstein-Wiener model, see the `idnlhw` reference page.

The default initial state for simulating a Hammerstein-Wiener model is 0. For more information about specifying initial conditions for simulation, see the IDNLHW Model reference page.

Simulate Identified Model Using Simulink Software

This example shows how to set the initial states for simulating a model such that the simulation provides a best fit to measured input-output data.

Prerequisites

Estimate a model, `M`, using a multiple-experiment data set, `Z`, which contains data from three experiments — `z1`, `z2`, and `z3`:

```
% Load multi-experiment data.
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos',...
'data', 'twobodiesdata'));

% Create an iddata object to store the multi-experiment data.
z1=iddata(y1, u1, 0.005,'Tstart',0);
z2=iddata(y2, u2, 0.005,'Tstart',0);
z3=iddata(y3, u3, 0.005,'Tstart',0);
Z = merge(z1,z2,z3);

% Estimate a 5th order state-space model.
opt = n4sidOptions('Focus','simulation');
[M,x0] = n4sid(Z,5,opt);
```

To simulate the model using input `u2`, use `x0(:,2)` as the initial states. `x0(:,2)` is computed to maximize the fit between the measured output, `y2`, and the response of `M`.

To compute initial states that maximizes the fit to the corresponding output `y2`, and simulate the model using the second experiment:

1. Extract the initial states that correspond to the second experiment for simulation :

   ```
   X0est = x0(:,2);
   ```

2. Open the System Identification Toolbox library by typing the following command at the MATLAB prompt:

   ```
   slident
   ```

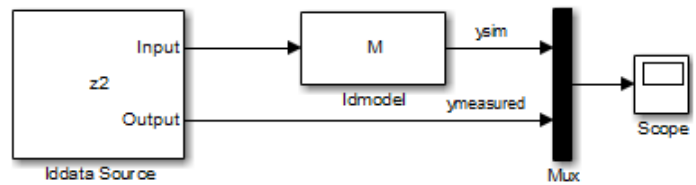
3. Open a new Simulink model window. Then, drag and drop an `Idmodel` block from the library into the model window.

4. Open the Function Block Parameters dialog box by double-clicking the `Idmodel` block. Specify the following block parameters:

   a. In the **Model variable** field, type `M` to specify the estimated model.

   b. In the **Initial state** field, type `X0est` to specify the estimated initial states. Click **OK**.

5. Drag and drop an `Iddata Source` block into the model window. Then, configure the model, as shown in the following figure.



6. Simulate the model for 2 seconds, and compare the simulated output `ysim` with the measured output `ymeasured` using the Scope block.

12

System Identification Tool GUI

Steps for Using the System Identification Tool GUI

A typical workflow in the System Identification Tool GUI includes the following steps:

1. Import your data into the MATLAB workspace, as described in "Representing Data in MATLAB Workspace" on page 2-9.
2. Start a new session in the System Identification Tool GUI, or open a saved session. For more information, see "Starting a New Session in the GUI" on page 12-4.
3. Import data into the GUI from the MATLAB workspace. For more information, see "Importing Data into the GUI" on page 2-17.
4. Plot and preprocess data to prepare it for system identification. For example, you can remove constant offsets or linear trends (for linear models only), filter data, or select data regions of interest. For more information, see "Preprocess Data".
5. Specify the data for estimation and validation. For more information, see "Specifying Estimation and Validation Data" on page 2-35.
6. Select the model type to estimate using the **Estimate** menu.
7. Validate models. For more information, see "Model Validation".
8. Export models to the MATLAB workspace for further analysis. For more information, see "Exporting Models from the GUI to the MATLAB Workspace" on page 12-12.

Working with the System Identification Tool GUI

In this section...

Starting and Managing GUI Sessions

What Is a System Identification Tool Session?

A *session* represents the total progress of your identification process, including any data sets and models in the System Identification Tool GUI.

You can save a session to a file with a `.sid` extension. For example, you can save different stages of your progress as different sessions so that you can revert to any stage by simply opening the corresponding session.

To start a new session, see "Starting a New Session in the GUI" on page 12-4.

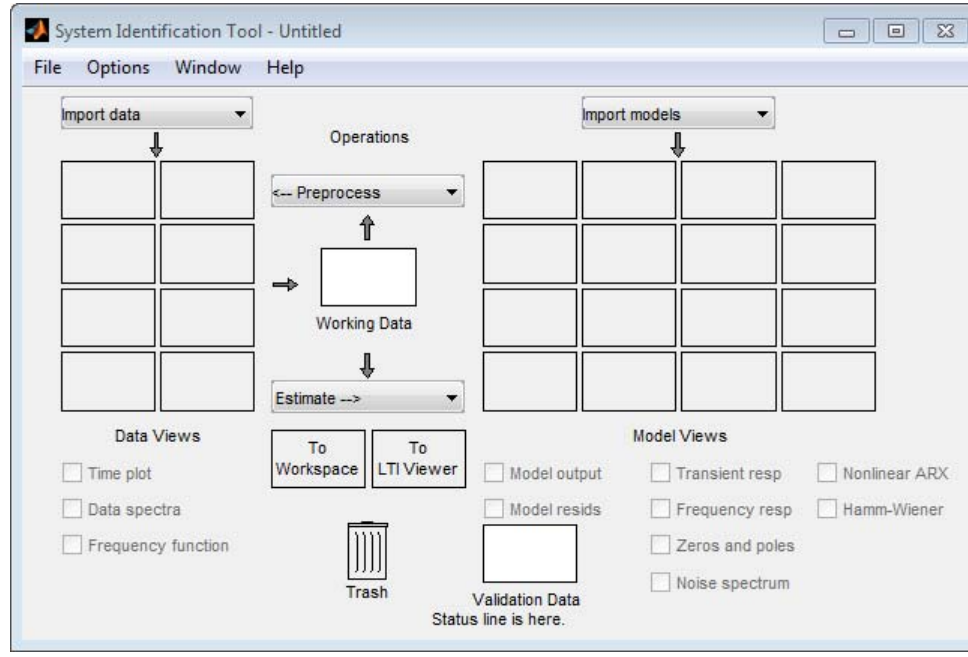
For more information about the steps for using the System Identification Tool GUI, see "Steps for Using the System Identification Tool GUI" on page 12-2.

Starting a New Session in the GUI

To start a new session in the System Identification Tool GUI, type the following command in the MATLAB Command Window:

```
ident
```

Alternatively, you can start a new session by selecting the **Apps** tab of MATLAB desktop. In the **Apps** section, click **System Identification**. This action opens the System Identification Tool.

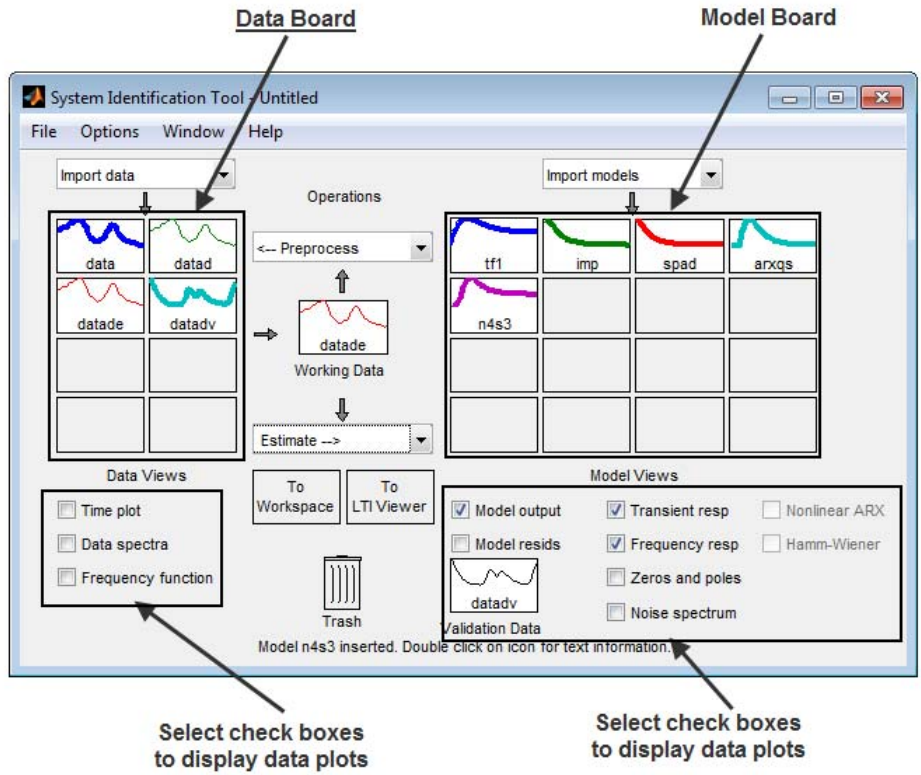


Note Only one session can be open at a time.

You can also start a new session by closing the current session using **File > Close session**. This toolbox prompts you to save your current session if it is not already saved.

Description of the System Identification Tool Window

The following figure describes the different areas in the System Identification Tool GUI.



The layout of the window organizes tasks and information from left to right. This organization follows a typical workflow, where you start in the top-left corner by importing data into the System Identification Tool GUI using the **Import data** menu and end in the bottom-right corner by plotting the characteristics of your estimated model on model plots. For more information about using the System Identification Tool GUI, see “Steps for Using the System Identification Tool GUI” on page 12-2.

The **Data Board** area, located below the **Import data** menu in the System Identification Tool GUI, contains rectangular icons that represent the data you imported into the GUI.

The Model Board, located to the right of the **<--Preprocess** menu in the System Identification Tool GUI, contains rectangular icons that represent the models you estimated or imported into the GUI. You can drag and drop model icons in the Model Board into open dialog boxes.

Opening a Saved Session

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

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

`session` is the file name of the session you want to open and `path` is the location of the session file. Session files have the extension `.sid`. When the session file in on the `matlabpath`, you can omit the `path` argument.

If the System Identification Tool GUI is already open, you can open a session by selecting **File > Open session**.

Note If there is data in the System Identification Tool GUI, you must close the current session before you can open a new session by selecting **File > Close session**.

Saving, Merging, and Closing Sessions

The following table summarizes the menu commands for saving, merging, and closing sessions in the System Identification Tool GUI.

Task	Command	Comment
Close the current session and start a new session.	**File > Close session**	You are prompted to save the current session before closing it.
Merge the current session with a previously saved session.	**File > Merge session**	You must start a new session and import data or models before you can select to merge it with a previously saved session. You are prompted to select the session file to merge with the current. This operation combines the data and the models of both sessions in the current session.
Save the current session.	**File > Save**	Useful for saving the session repeatedly after you have already saved the session once.
Save the current session under a new name.	**File > Save As**	Useful when you want to save your work incrementally. This command lets you revert to a previous stage, if necessary.

Deleting a Session

To delete a saved session, you must delete the corresponding session file.

Managing Models

- "Exporting Models from the GUI to the MATLAB Workspace" on page 12-12

Importing Models into the GUI

You can import System Identification Toolbox models from the MATLAB workspace into the System Identification Tool GUI. If you have Control System Toolbox software, you can also import any models (LTI objects) you created using this toolbox.

The following procedure assumes that you begin with the System Identification Tool GUI already open. If this window is not open, type the following command at the prompt:

```
ident
```

To import models into the System Identification Tool GUI:

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

Viewing Model Properties

You can get information about each model in the System Identification Tool GUI by right-clicking the corresponding model icon.

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

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

Renaming Models and Changing Display Color

You can rename a model and change its display color by double-clicking the model icon in the System Identification Tool GUI.

The Data/model Info dialog box opens. This dialog box describes both the contents and the properties of the model. The object description area displays the syntax of the operations you used to create the model in the GUI.

To rename the model, enter a new name in the **Model name** field.

You can also specify a new display color using three RGB values in the **Color** field. Each value is between `0` to `1` and indicates the relative presence of red, green, and blue, respectively. For more information about specifying default data color, see "Customizing the System Identification Tool GUI" on page 12-17.

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

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

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

Finally, you can enter comments about the origin and state of the model in the **Diary And Notes** area.

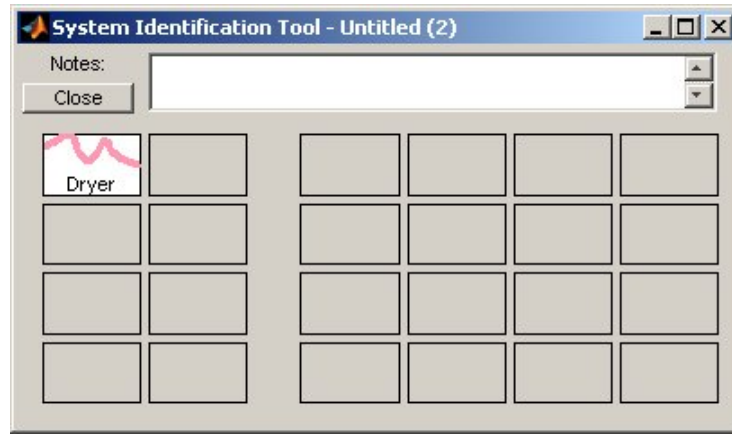
To view model properties in the MATLAB Command Window, click **Present**.

Organizing Model Icons

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

Note You cannot drag and drop a model icon into the data area on the left.

When you need additional space for organizing model icons, select **Options > Extra model/data board** in the System Identification Tool GUI. This action opens an extra session window with blank rectangles. The new window is an extension of the current session and does not represent a new session.



Tip When you import or estimate models and there is insufficient space for the icons, an additional session window opens automatically.

You can drag and drop model icons between the main System Identification Tool GUI and any extra session windows.

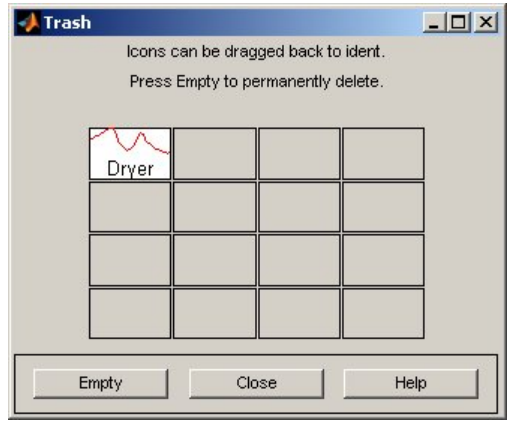
Type comments in the **Notes** field to describe the models. When you save a session, as described in "Saving, Merging, and Closing Sessions" on page 12-6, all additional windows and notes are also saved.

Deleting Models in the GUI

To delete models in the System Identification Tool GUI, drag and drop the corresponding icon into **Trash**. Moving items to **Trash** does not permanently delete these items.

To restore a model from **Trash**, drag its icon from **Trash** to the Model Board in the System Identification Tool GUI. You can view the **Trash** contents by double-clicking the **Trash** icon.

Note You must restore a model to the Model Board; you cannot drag model icons to the Data Board.



To permanently delete all items in **Trash**, select **Options > Empty trash**.

Exiting a session empties **Trash** automatically.

Exporting Models from the GUI to the MATLAB Workspace

The models you create in the System Identification Tool GUI are not available in the MATLAB workspace until you export them. Exporting is necessary when you need to perform an operation on the model that is only available at the command line. Exporting models to the MATLAB workspace also makes them available to the Simulink software or another toolbox, such as the Control System Toolbox product.

To export a model to the MATLAB workspace, drag and drop the corresponding icon to the **To Workspace** rectangle.

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

Working with Plots

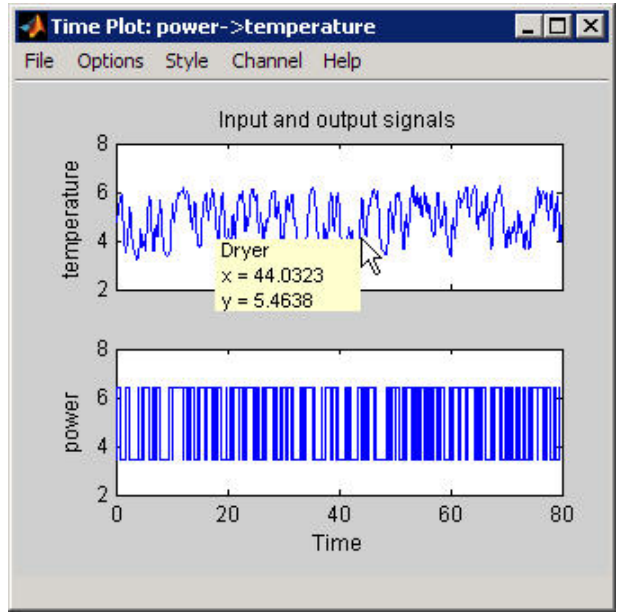
Identifying Data Sets and Models on Plots

You can identify data sets and models on a plot by color: the color of the line in the data or model icon in the System Identification Tool GUI matches the line color on the plots.

You can also display data tips for each line on the plot by clicking a plot curve and holding down the mouse button.

Note You must disable zoom by selecting **Style > Zoom** before you can display data tips. For more information about enabling zoom, see "Magnifying Plots" on page 12-14.

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



Data Tip on a Plot

Changing and Restoring Default Axis Limits

There are two ways to change which portion of the plot is currently in view:

- "Magnifying Plots" on page 12-14
- "Setting Axis Limits" on page 12-15

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

Tip To verify that zoom is active, click the **Style** menu. A check mark should appear next to **Zoom**.

You can adjust magnification in the following ways:

- To zoom in default increments, left-click the portion of the plot you want to center in the plot window.
- To zoom in on a specific region, click and drag a rectangle that identifies the region for magnification. When you release the mouse button, the selected region is displayed.
- To zoom out, right-click on the plot.

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

Setting Axis Limits. You can change axis limits for the vertical and the horizontal axes of the input and output channels that are currently displayed on the plot.

1. Select **Options > Set axes limits** to open the Limits dialog box.
2. Specify a new range for each axis by editing its lower and upper limits. The limits must be entered using the format *[LowerLimit UpperLimit]*. Click **Apply**. For example:

   ```
   [0.1 100]
   ```

   Note To restore full axis limits, select the **Auto** check box to the right of the axis name, and click **Apply**.

   ---

3. To plot data on a linear scale, clear the **Log** check box to the right of the axis name, and click **Apply**.

   Note To revert to base-10 logarithmic scale, select the **Log** check box to the right of the axis name, and click **Apply**.

   ---

4. Click **Close**.

Note To view the entire data range, select **Options > Autorange** in the plot window.

Selecting Measured and Noise Channels in Plots

Model inputs and outputs are called *channels*. When you create a plot of a multivariable input-output data set or model, the plot only shows one input-output channel pair at a time. The selected channel names are displayed in the title bar of the plot window.

Note When you select to plot multiple data sets, and each data set contains several input and output channels, the **Channel** menu lists channel pairs from all data sets.

You can select a different input-output channel pair from the **Channel** menu in any System Identification Toolbox plot window.

The **Channel** menu uses the following notation for channels: `u1->y2` means that the plot displays a transfer function from input channel `u1` to output channel `y2`. System Identification Toolbox estimates as many noise sources as there are output channels. In general, `e@ynam` indicates that the noise source corresponds to the output with name `ynam`.

For example, `e@y3->y1` means that the transfer function from the noise channel (associated with `y3`) to output channel `y2` is displayed. For more information about noise channels, see "Separation of Measured and Noise Components of Models" on page 3-161.

Tip When you import data into the System Identification Tool GUI, it is helpful to assign meaningful channel names in the Import Data dialog box. For more information about importing data, see "Importing Data into the GUI" on page 2-17.

Grid and Line Styles in Plots

There are several **Style** options that are common to all plot types. These include the following:

- "Grid Lines" on page 12-17
- "Solid or Dashed Lines" on page 12-17

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

Solid or Dashed Lines. To display currently visible lines as a combination of solid, dashed, dotted, and dash-dotted line style, select **Style > Separate linestyles**.

To display all solid lines, select **Style > All solid lines**. This choice is the default.

All line styles match the color of the corresponding data or model icon in the System Identification Tool GUI.

Opening a Plot in a MATLAB Figure Window

The MATLAB Figure window provides editing and printing commands for plots that are not available in the System Identification Toolbox plot window. To take advantage of this functionality, you can first create a plot in the System Identification Tool GUI, and then open it in a MATLAB Figure window to fine-tune the display.

After you create the plot, as described in "Plotting Models in the GUI" on page 8-7, select **File > Copy figure** in the plot window. This command opens the plot in a MATLAB Figure window.

Printing Plots

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

Customizing the System Identification Tool GUI

- "Types of GUI Customization" on page 12-18

Types of GUI Customization

The System Identification Tool GUI lets you customize the window behavior and appearance. For example, you can set the size and position of specific dialog boxes and modify the appearance of plots.

You can save the session to save the customized GUI state.

Advanced users might choose to edit the file that controls default settings, as described in "Modifying idlayout.m" on page 12-19.

Saving Session Preferences

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

- Size and position of the System Identification Tool GUI
- Sizes and positions of dialog boxes
- Four recently used sessions
- Plot options, such as line styles, zoom, grid, and whether the input is plotted using zero-order hold or first-order hold between samples

You can only edit `idprefs.mat` by changing preferences in the GUI.

The `idprefs.mat` file is located in the same folder as `startup.m`, by default. To change the location where your preferences are saved, use the `midprefs` command with the new path as the argument. For example:

```
midprefs('c:\matlab\toolbox\local\')
```

You can also type `midprefs` and browse to the desired folder.

To restore the default preferences, select **Options > Default preferences**.

Modifying idlayout.m

Advanced users might want to customize the default plot options by editing `idlayout.m`.

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

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

You can customize the following plot options in `idlayout.m`:

- Order in which colors are assigned to data and model icons
- Line colors on plots
- Axis limits and tick marks
- Plot options, set in the plot menus
- Font size

Note When you save preferences using **Options > Save preferences** to `idprefs.mat`, these preferences override the defaults in `idlayout.m`. To give `idlayout.m` precedence every time you start a new session, select **Options > Default preferences**.

Index

A

B

C

D

E

F

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z