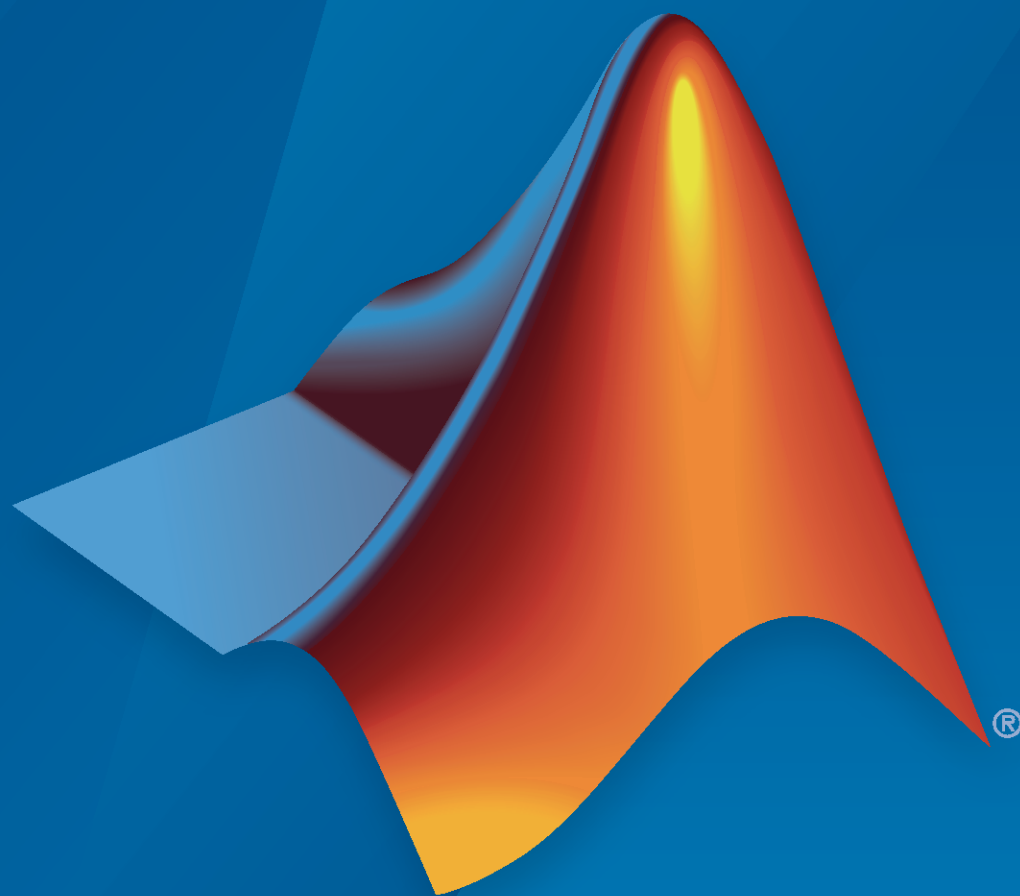# System Identification Toolbox™

User's Guide

Lennart Ljung

MATLAB®&SIMULINK®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

Contents

Choosing Your System Identification Approach

1

2   Data Import and Processing

3

<div align="right">

Transform Data

</div>

4

<div align="right">

Linear Model Identification

</div>

Identifying Process Models

5

Identifying Input-Output Polynomial Models

6

Identifying State-Space Models

7

Identifying Transfer Function Models

8

Identifying Frequency-Response Models

9

Identifying Impulse-Response Models

10

Nonlinear Black-Box Model Identification

11

Identify Hammerstein-Wiener Models

12

13

ODE Parameter Estimation (Grey-Box Modeling)

Time Series Identification

14

Recursive Model Identification

15

Online Estimation

16

Setting Toolbox Preferences

18

Control Design Applications

19

System Identification Toolbox Blocks

20

System Identification App

21

System Identification UI Help

22

23

Choosing Your System Identification Approach

Acknowledgments

What Are Model Objects?

Model Objects Represent Linear Systems

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

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

The main families of model objects are:

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

About Model Data

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

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

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

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

See Also

More About

- "Types of Model Objects" on page 1-4

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.

GENERALIZED MODELS

Control Design Blocks

NUMERIC MODELS	Dynamic Control Design Blocks	Static Control Design Blocks

Numeric LTI Models
 	
`tf`   `pid`
`zpk`   `pidstd`
`ss`   `pid2`
`frd`   `pidstd2`
 	
Sparse Models
 	
`sparss`   `mechss`
 	
Identified LTI Models
 	
`idgrey`   `idfrd`
`idtf`   `idpoly`
`idss`   `idproc`
 	
Identified Nonlinear Models
 	
`idnlarx`
`idnlgrey`
`idnlhw`
	Tunable
	`tunableTF`
	`tunableSS`
	`tunableGain`
	`tunablePID`
	`tunablePID2`
	
	Uncertain
	`ultidyn`
	`umargin`
	`udyn`
	
	Analysis Point Block
	`AnalysisPoint`
	
	Generalized LTI Models
	
	Tunable and/or Uncertain
	`genss`
	`genfrd`
	
	Uncertain Only
	`uss`
	`ufrd`
		Tunable
		`realp`
		
		Uncertain
		`ureal`
		`ucomplex`
		`ucomplexm`
		
		Generalized Matrices
		
		Tunable
		`genmat`
		
		Uncertain
		`umat`

DYNAMIC SYSTEM MODELS		**STATIC MODELS**

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

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

See Also

More About

- "What Are Model Objects?" on page 1-3
- "Dynamic System Models" on page 1-6
- "Numeric Models" on page 1-8

Dynamic System Models

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

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

The following table lists the Dynamic System Models.

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

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

See Also

More About

- "Numeric Linear Time Invariant (LTI) Models" on page 1-8
- "Identified LTI Models" on page 1-8
- "Identified Nonlinear Models" on page 1-9

Numeric Models

Numeric Linear Time Invariant (LTI) Models

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

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

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

Creating Numeric LTI Models

For information about creating numeric LTI models, see:

- "Transfer Functions" (Control System Toolbox)
- "State-Space Models" (Control System Toolbox)
- "Frequency Response Data (FRD) Models" (Control System Toolbox)
- "Proportional-Integral-Derivative (PID) Controllers" (Control System Toolbox)

Applications of Numeric LTI Models

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

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

Identified LTI Models

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

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

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

Identified Nonlinear Models

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

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

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

About Identified Linear Models

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

Model Type	Transfer Functions G and H	Configuration Method

Process model (`idproc`)	Represents a process model, which provides options to represent the noise dynamics as either first- or second-order ARMA process (that is, $H(s)= C(s)/A(s)$, where $C(s)$ and $A(s)$ are monic polynomials of equal degree). The measured component, $G(s)$, is represented by a transfer function expressed in pole-zero form.	For process (and grey-box) models, the noise component is often treated as an on-demand extension to an otherwise measured component-centric representation. For these models, you can add a noise component by using the `DisturbanceModel` estimation option. For example:
		
		`model = procest(data,'P1D')`
		
		estimates a model whose equation is:
		
		$$y(s) = K_p \frac{1}{(T_{p1}s + 1)} e^{-sTd} u(s) + e(s).$$
		
		To add a second order noise component to the model, use:
		
		`Options = procestOptions('DisturbanceModel','ARMA1'`
		`model = procest(data,'P1D',Options);`
		
		This model has the equation:
		
		$$y(s) = K_p \frac{1}{(T_{p1}s + 1)} e^{-sTd} u(s) + \frac{1 + c_1 s}{1 + d_1 s} e(s)$$
		
		where the coefficients $c1$ and $d1$ parameterize the noise component of the model. If you are constructing a process model using the `idproc` command, specify the structure of the measured component using the `Type` input argument and the noise component by using the `NoiseTF` name-value pair. For example,
		
		`model = idproc('P1','Kp',1,'Tp1',1,'NoiseTF',...`
		`struct('num',[1 0.1],'den',[1 0.5]))`
		
		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

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

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

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

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

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

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

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

Structured Estimations

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

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

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

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

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

Estimation Options

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

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

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

Information about the options used to create an estimated model is stored in the `OptionsUsed` field of the model's `Report` property. For more information, see "Estimation Report" on page 1-21.

See Also

More About

- "Types of Model Objects" on page 1-4
- "Available Linear Models" on page 1-19
- "About Identified Nonlinear Models" on page 11-2

Linear Model Structures

About System Identification Toolbox Model Objects

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

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

This toolbox includes nine classes for representing models. For example, `idss` represents linear state-space models and `idnlarx` represents nonlinear ARX models. For a complete list of available model objects, see "Available Linear Models" on page 1-19 and "Available Nonlinear Models" on page 11-10.

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

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

Every class has a special method, called the *constructor*, for creating objects of that class. Using a constructor creates an instance of the corresponding class or *instantiates the object*. The constructor name is the same as the class name. For example, `idss` and `idnlarx` are both the name of the class and the name of the constructor for instantiating the linear state-space models and nonlinear ARX models, respectively.

When to Construct a Model Structure Independently of Estimation

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

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

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

In most cases, you can use the estimation commands to both construct and estimate the model—without having to construct the model object independently. For example, the estimation command `tfest` creates a transfer function model using data and the number of poles and zeros of the model. Similarly, `nlarx` creates a nonlinear ARX model using data and model orders and delays that define the regressor configuration. For information about how to both construct and estimate models with a single command, see "Model Estimation Commands" on page 1-31.

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

Commands for Constructing Linear Model Structures

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

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

For information about how to both construct and estimate models with a single command, see "Model Estimation Commands" on page 1-31.

Summary of Model Constructors

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

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

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.

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

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

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

Viewing Model Properties and Estimated Parameters

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

Task	Command	Example

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

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

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

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

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

See Also

Validate each model directly after estimation to help fine-tune your modeling strategy. When you do not achieve a satisfactory model, you can try a different model structure and order, or try another identification algorithm. For more information about validating and troubleshooting models, see "Validating Models After Estimation" on page 17-2.

Available Linear Models

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

Model Type	Usage	Learn More
Transfer function (`idtf`)	Use this structure to represent transfer functions: $$y = \frac{num}{den}u + e$$ where *num* and *den* are polynomials of arbitrary lengths. You can specify initial guesses for, and estimate, *num*, *den*, and transport delays.	"Transfer Function Models"
Process model (`idproc`)	Use this structure to represent process models that are low order transfer functions expressed in pole-zero form. They include integrator, delay, zero, and up to 3 poles.	"Process Models"
State-space model (`idss`)	Use this structure to represent known state-space structures and black-box structures. You can fix certain parameters to known values and estimate the remaining parameters. You can also prescribe minimum/ maximum bounds on the values of the free parameters. If you need to specify parameter dependencies or parameterize the state-space matrices using your own parameters, use a grey-box model.	"State-Space Models"

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

See Also

More About

- "Linear Model Structures" on page 1-15
- "About Identified Linear Models" on page 1-10

Estimation Report

What is an Estimation Report?

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

Specifically, the estimation report has the following information:

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

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

You can use the report to:

- Keep an estimation log, such as the data, default and other settings used, and estimated results such as parameter values, initial conditions, and fit. See "Access Estimation Report" on page 1-21.
- Compare options or results of separate estimations. See "Compare Estimated Models Using Estimation Report" on page 1-22.
- Configure another estimation using the previously specified options. See "Analyze and Refine Estimation Results Using Estimation Report" on page 1-23.

Access Estimation Report

This example shows how to access the estimation report.

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

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

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

sys_report =
            Status: 'Estimated using TFEST'
```

```
           Method: 'TFEST'
  InitializeMethod: 'iv'
         N4Weight: 'Not applicable'
        N4Horizon: 'Not applicable'
  InitialCondition: 'estimate'
              Fit: [1x1 struct]
       Parameters: [1x1 struct]
      OptionsUsed: [1x1 idoptions.tfest]
        RandState: []
         DataUsed: [1x1 struct]
      Termination: [1x1 struct]
```

Explore the options used during the estimation.

```
sys.Report.OptionsUsed
```

```
Option set for the tfest command:

    InitializeMethod: 'iv'
   InitializeOptions: [1x1 struct]
    InitialCondition: 'auto'
             Display: 'off'
    InputInterSample: 'auto'
         InputOffset: []
        OutputOffset: []
   EstimateCovariance: 1
      Regularization: [1x1 struct]
        SearchMethod: 'auto'
       SearchOptions: '<Optimization options set>'
     WeightingFilter: []
     EnforceStability: 0
        OutputWeight: []
            Advanced: [1x1 struct]
```

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

```
sys.Report.Fit
```

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

Compare Estimated Models Using Estimation Report

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

Load estimation data.

```
load iddata1 z1;
```

Estimate a transfer function model.

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

Estimate a state-space model.

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

Estimate an ARX model.

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

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

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

```
fit_tf = 70.7720
```

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

```
fit_ss = 76.3808
```

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

```
fit_arx = 68.7220
```

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

Analyze and Refine Estimation Results Using Estimation Report

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

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

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

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

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

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

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

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

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

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

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

See Also

More About

- "About Identified Linear Models" on page 1-10
- "About Identified Nonlinear Models" on page 11-2

Imposing Constraints on Model Parameter Values

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

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

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

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

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

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

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

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

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

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

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

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

Recommended Model Estimation Sequence

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

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

Note Model structure is not the only factor that determines model accuracy. If your model is poor, you might need to preprocess your data by removing outliers or filtering noise. For more information, see "Ways to Prepare Data for System Identification" on page 2-5.

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

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

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

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

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

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

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

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

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

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

For general information about choosing you model strategy, see "System Identification Overview". For information about validating models, see "Validating Models After Estimation" on page 17-2.

Supported Models for Time- and Frequency-Domain Data

Supported Models for Time-Domain Data

Continuous-Time Models

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

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

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

Discrete-Time Models

You can estimate all linear on page 1-15 and nonlinear on page 11-6 models supported by the System Identification Toolbox product as discrete-time models, except process models, which are defined only in continuous-time..

ODEs (Grey-Box Models)

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

Nonlinear Models

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

You can also estimate nonlinear grey-box models from time-domain data. See "Estimate Nonlinear Grey-Box Models" on page 13-25.

Supported Models for Frequency-Domain Data

There are two types of frequency-domain data:

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

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

Continuous-Time Models

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

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

- State-space models using continuous- or discrete-time data.

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

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

Discrete-Time Models

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

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

ODEs (Grey-Box Models)

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

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

Nonlinear Black-Box Models

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

See Also

"Supported Continuous- and Discrete-Time Models" on page 1-29

Supported Continuous- and Discrete-Time Models

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

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

Supported Continuous-Time Models

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

Supported Discrete-Time Models

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

Model Estimation Commands

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

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

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

Commands for Constructing and Estimating Models

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

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

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

Modeling Multiple-Output Systems

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 1-32
- "Modeling Multiple Outputs as a Combination of Single-Output Models" on page 1-32

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 1-33.

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 2-73.

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

- For working in the System Identification app, see "Create Data Sets from a Subset of Signal Channels" on page 2-24.
- For working at the command line, see the "Select Data Channels, I/O Data and Experiments in iddata Objects" on page 2-37.

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 2-40 or "Adding Input or Output Channels in idfrd Objects" on page 2-63.

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

Improving Multiple-Output Estimation Results by Weighing Outputs During Estimation

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

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

Note

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

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

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

Regularized Estimates of Model Parameters

What Is Regularization?

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

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

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

MSE = |Bias|^2 + Variance

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

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

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

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

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

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

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

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

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

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

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

The square matrix R gives additional freedom for:

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

For structured models such as grey-box models, you may want to keep the estimated parameters close to their guess values to maintain the physical validity of the estimated model. This can be achieved by generalizing the penalty term to $\lambda(\theta - \theta^*)^T R(\theta - \theta^*)$, such that the cost function becomes:

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

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

In regularization:

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

A formal interpretation in a Bayesian setting is that θ has a prior distribution that is Gaussian with mean θ^* and covariance matrix $\sigma^2/\lambda R^{-1}$, where σ^2 is the variance of $\varepsilon(t)$. The use of regularization can therefore be linked to some prior information about the system. This could be quite soft, such as the system is stable.

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

When to Use Regularization

Use regularization for:

- Identifying overparameterized models.
- Imposing *a priori* knowledge of model parameters in structured models.
- Incorporating knowledge of system behavior in ARX and FIR models.

Identifying Overparameterized Models

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

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

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

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

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

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

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

Imposing A Priori Knowledge of Model Parameters in Structured Models

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

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

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

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

Incorporating Knowledge of System Behavior in ARX and FIR Models

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

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

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

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

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

```
% Specify regularization constants.
[L, R] = arxRegul(ze,[15 15 1]);
optARX = arxOptions;
optARX.Regularization.Lambda = L;
optARX.Regularization.R = R;
% Estimate ARX model.
```

```
sysARX = arx(ze,[15 15 1],optARX);
% Convert model to continuous time.
sysc = d2c(sysARX);
% Convert model to transfer function.
sys2 = idtf(sysc);
% Validate the models sys1 and sys2.
compare(zv,sys1,sys2)
```

Choosing Regularization Constants

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

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

Use the following techniques for determining λ and R values:

- Incorporate prior information using tunable kernels.
- Perform cross-validation tests.

Incorporate Prior Information Using Tunable Kernels

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

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

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

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

You can estimate the parameters of the kernel by adjusting them to the measured data using the `RegularizationKernel` input of the `arxRegul` command. For example, the `DC` kernel estimates all three parameters while the `TC` kernel links $\rho = \sqrt{\mu}$. This technique of tuning kernels applies to all linear-in-parameter models such as ARX and FIR models.

Perform Cross-Validation Tests

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

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

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

2   Estimate a model for each candidate regularization constant set.

3   Compare the model fit to the validation data.

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

For example:

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

References

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

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

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

See Also

Related Examples

- "Estimate Regularized ARX Model Using System Identification App" on page 1-40
- "Regularized Identification of Dynamic Systems" on page 1-55

More About

- "Loss Function and Model Quality Metrics" on page 1-46

Estimate Regularized ARX Model Using System Identification App

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

Open a saved System Identification App session.

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

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

- Estimation data `eData`

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

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

- Transfer function model `trueSys`

  `trueSys` is the transfer function model used to generate the estimation data `eData` described previously. You also use the impulse response of this model later to compare the impulse responses of estimated ARX models.
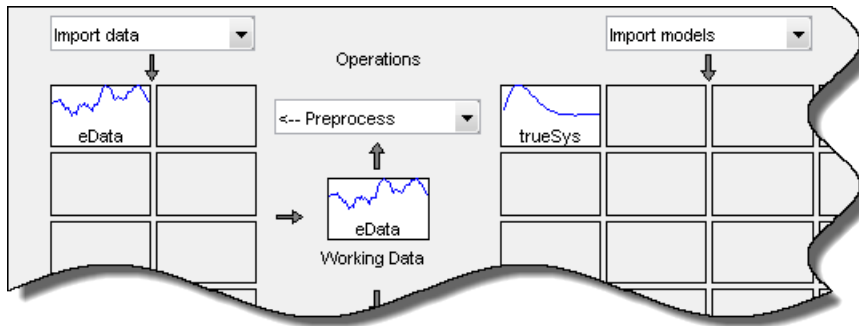
Estimate a 50th-order ARX model.

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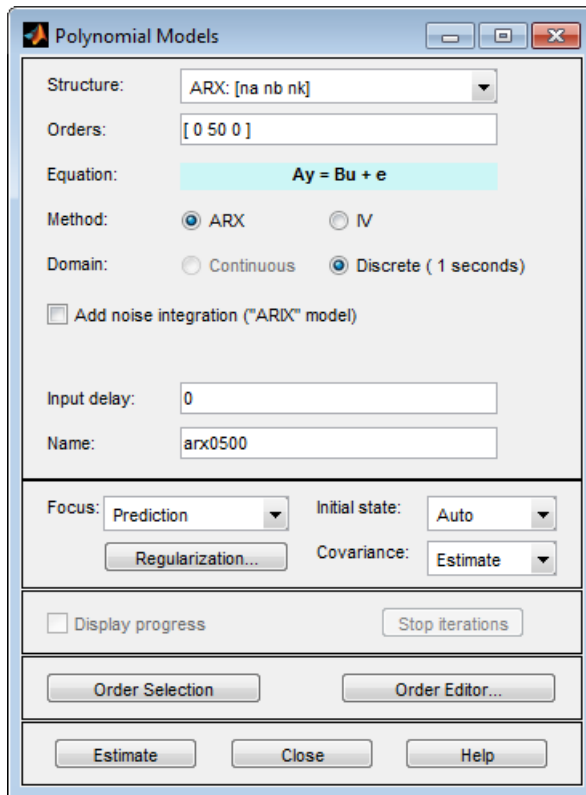
4 Click **Estimate** to estimate the model.

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

Estimate a 50th-order regularized ARX model.

1 In the Polynomial Models dialog box, click **Regularization**.

2 In the Regularization Options dialog box, select TC from the **Regularization Kernel** drop-down list.
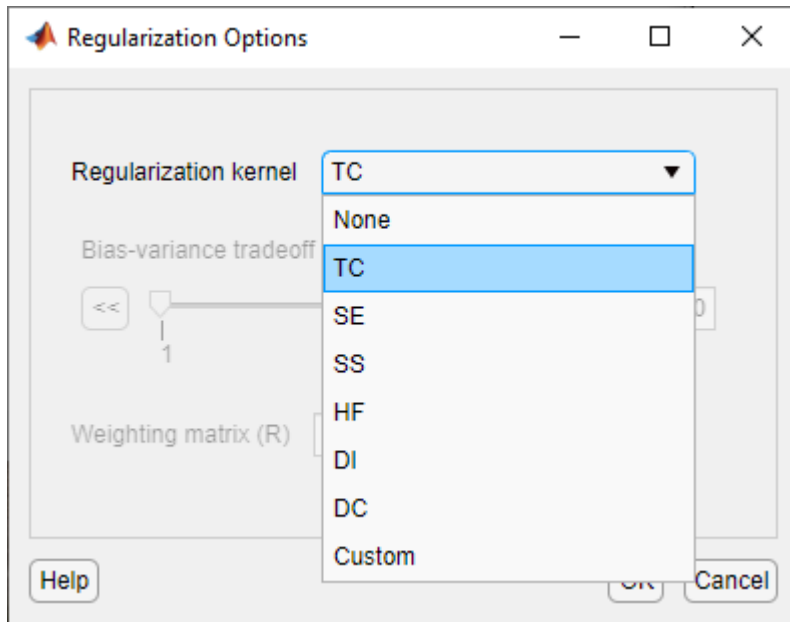
Specifying this option automatically determines regularization constants using the `TC` regularization kernel. To learn more, see the `arxRegul` reference page.

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

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

4 Click **Estimate**.

A model `arx0500reg` is added to the System Identification app.
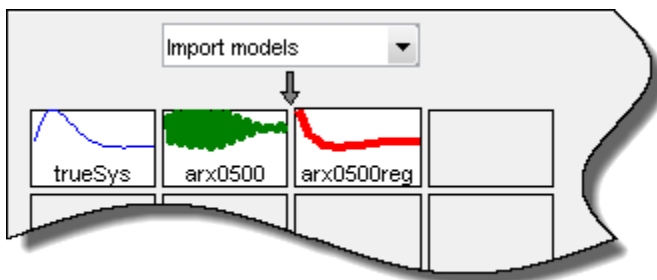
Compare the unregularized and regularized model outputs to estimation data.

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

The Measured and simulated model output plot shows that both the models have an 84% fit with the data.
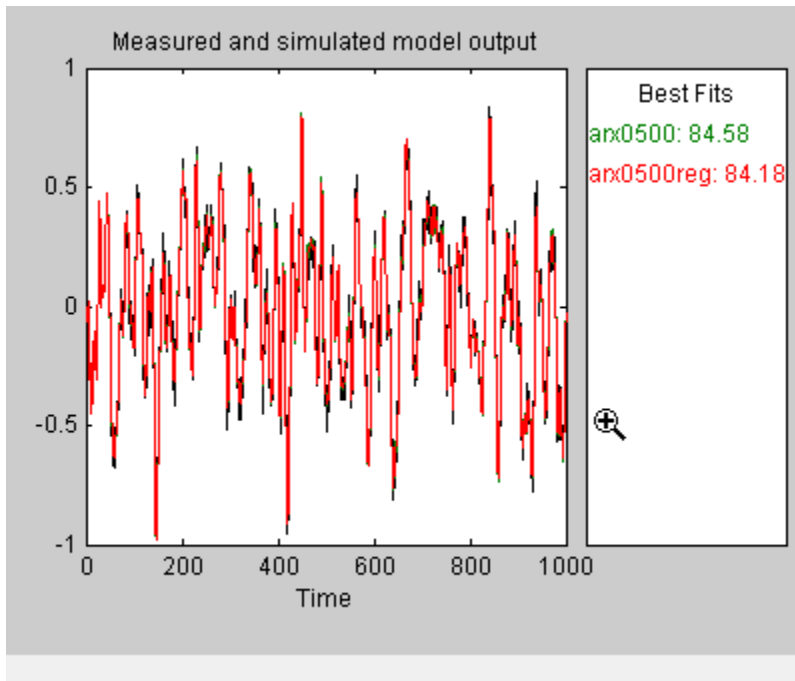
Determine if regularization leads to parameter values with less variance.

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

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

2   Select the **Transient resp** check box to open the Transient Response plot window.
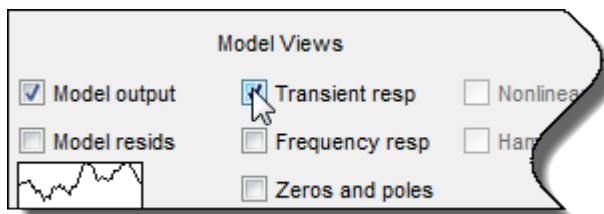
By default, the plot shows the step response.

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

4 Select **Options > Show 99% confidence intervals** to plot the confidence intervals.
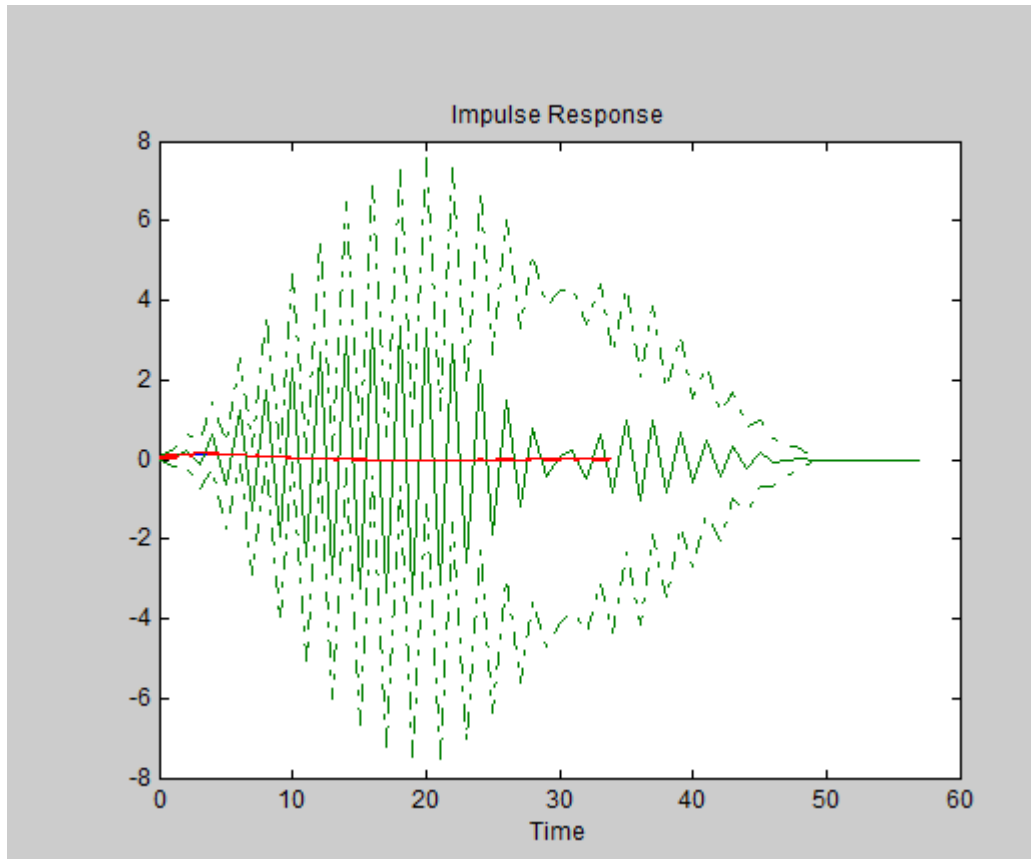
The plot shows that the impulse response of the unregularized model `arx0500` is far off from the true system and has large uncertainties.

To get a closer look at the model fits to the data and the variances, magnify a portion of the plot.
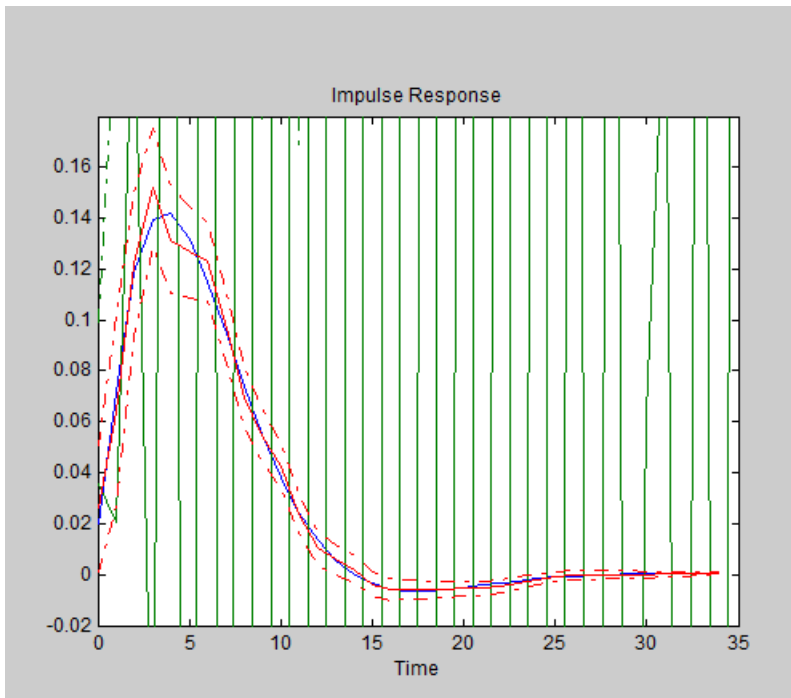
The fit of the regularized ARX model `arx0500reg` closely matches the impulse response of the true system and the variance is greatly reduced as compared to the unregularized model.

See Also

Related Examples

- "Regularized Identification of Dynamic Systems" on page 1-55

More About

- "Regularized Estimates of Model Parameters" on page 1-34

Loss Function and Model Quality Metrics

What is a Loss Function?

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

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

where:

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

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

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

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

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

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

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

Options to Configure the Loss Function

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

Estimation Option	Description	Notes
Focus	Focus option affects how $e(t)$ in the loss function is computed: • When Focus is `'prediction'`, $e(t)$ represents 1-step ahead prediction error: $\qquad e_p(t) = y_{measured}(t) - y_{predicted}(t)$ • When Focus is `'simulation'`, $e(t)$ represents the simulation error: $\qquad e_s(t) = y_{measured}(t) - y_{simulated}(t)$ --- **Note** For models whose noise component is trivial, ($H(q) = 1$), $e_p(t)$, and $e_s(t)$ are equivalent. --- The Focus option can also be interpreted as a weighting filter in the loss function. For more information, see "Effect of Focus and WeightingFilter Options on the Loss Function" on page 1-50.	• Specify the Focus option in the estimation option sets. • The estimation option sets for oe and tfest do not have a Focus option because the noise-component for the estimated models is trivial, and so $e_p(t)$ and $e_s(t)$ are equivalent.
WeightingFilter	When you specify a weighting filter, prefiltered prediction or simulation error is minimized: $\qquad e_f(t) = \mathscr{L}(e(t))$ where $\mathscr{L}(.)$ is a linear filter. The WeightingFilter option can be interpreted as a custom weighting filter that is applied to the loss function. For more information, see "Effect of Focus and WeightingFilter Options on the Loss Function" on page 1-50.	• Specify the WeightingFilter option in the estimation option sets. Not all options for WeightingFilter are available for all estimation commands.

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

Estimation Option	Description	Notes
`OutputWeight`	`OutputWeight` option configures the weighting matrix $W(\theta)$ in the loss function and lets you control the relative importance of output channels during multi-output estimations. • When `OutputWeight` is `'noise'`, $W(\theta)$ equals the inverse of the estimated variance of error $e(t)$: $$W(\theta) = \left(\tfrac{1}{N}E^T(\theta) \quad E(\theta)\right)^{-1}$$ Because W depends on θ, the weighting is determined as a part of the estimation. Minimization of the loss function with this weight simplifies the loss function to: $$V(\theta) = det\left(\tfrac{1}{N}E^T(\theta) \quad E(\theta)\right)$$ Using the inverse of the noise variance is the optimal weighting in the maximum likelihood sense. • When `OutputWeight` is an *ny*-by-*ny* positive semidefinite matrix, a constant weighting is used. This loss function then becomes a weighted sum of squared errors.	• Specify the `OutputWeight` option in the estimation option sets. Not all options for `OutputWeight` are available for all estimation commands. • `OutputWeight` is not available for polynomial model estimation because such models are always estimated one output at a time. • `OutputWeight` cannot be `'noise'` when `SearchMethod` is `'lsqnonlin'`.
`ErrorThreshold`	`ErrorThreshold` option specifies the threshold for when to adjust the weight of large errors from quadratic to linear. Errors larger than `ErrorThreshold` times the estimated standard deviation have a linear weight in the loss function. $$V(\theta) = \quad \tfrac{1}{N}\left(\sum_{t \in I} e^T(t,\theta) \quad W(\theta) \quad e(t,\theta) + \sum_{t \in J} v^T(t,\theta) \quad W(\theta) \quad v(t,\theta)\right)$$ where: • I represents those time instants for which $\lvert e(t)\rvert < \rho*\sigma$, where ρ is the error threshold. • J represents the complement of I, that is, the time instants for which $\lvert e(t)\rvert >= \rho*\sigma$. • σ is the estimated standard deviation of the error. The error $v(t,\theta)$ is defined as: $$v(t,\theta) = e(t,\theta)*\sigma\frac{\rho}{\sqrt{\lvert e(t,\theta)\rvert}}$$	• Specify the `ErrorThreshold` option in the estimation option sets. • A typical value for the error threshold $\rho = 1.6$ minimizes the effect of data outliers on the estimation results.

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

Effect of Focus and WeightingFilter Options on the Loss Function

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

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

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

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

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

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

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

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

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

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

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

Substituting for $E(\omega)$ gives:

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

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

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

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

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

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

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

- **`Focus` is `'prediction'`** — The software minimizes the weighted prediction error $e_f(t) = \mathscr{L}(e_p(t))$, and the estimated model has the form:

  $y(t) = G(q)u(t) + H_1(q)e(t)$

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

  When H is parameterized independent of G, you can treat the filter $\mathscr{L}(.)$ as a way of affecting the estimation bias distribution. That is, you can shape the trade-off between fitting G to the system frequency response and fitting $H/\mathscr{L}$ to the disturbance spectrum when minimizing the loss function. For more details see, section 14.4 in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

- **`Focus` is `'simulation'`** — The software first estimates G by minimizing the weighted simulation error $e_f(t) = \mathscr{L}(e_s(t))$, where $e_s(t) = y_{measured}(t) - G(q)u_{measured}(t)$. Once G is estimated, the software fixes it and computes H by minimizing pure prediction errors $e(t)$ using unfiltered data. The estimated model has the form:

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

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

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

Model Quality Metrics

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

- `FitPercent`, `LossFcn`, and `MSE` are measures of the actual quantity that is minimized during the estimation. For example, if `Focus` is `'simulation'`, these quantities are computed for the simulation error $e_s(t)$. Similarly, if you specify the `WeightingFilter` option, then `LossFcn`, `FPE`, and `MSE` are computed using filtered residuals $e_f(t)$.

- `FPE`, `AIC`, `nAIC`, `AICc`, and `BIC` measures are computed as properties of the output disturbance according to the relationship:

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

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

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

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

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

Quality Metric	Description
FitPercent	Normalized Root Mean Squared Error (NRMSE) expressed as a percentage, defined as: $$FitPercent = \ 100\left(1 - \frac{\|y_{measured} - y_{model}\|}{\|y_{measured} - \overline{y_{measured}}\|}\right)$$ where: • $y_{measured}$ is the measured output data. • $\overline{y_{measured}}$ is its (channel-wise) mean. • y_{model} is the simulated or predicted response of the model, governed by the `Focus`. • ‖.‖ indicates the 2-norm of a vector. `FitPercent` varies between `-Inf` (bad fit) to 100 (perfect fit). If the value is equal to zero, then the model is no better at fitting the measured data than a straight line equal to the mean of the data.
LossFcn	Value of the loss function when the estimation completes. It contains effects of error thresholds, output weight, and regularization used for estimation.
MSE	Mean Squared Error measure, defined as: $$MSE = \ \frac{1}{N}\sum_{t=1}^{N} e^T(t)\ e(t)$$ where: • $e(t)$ is the signal whose norm is minimized for estimation. • N is the number of data samples in the estimation dataset.
FPE	Akaike's Final Prediction Error (FPE), defined as: $$FPE = det\left(\frac{1}{N}E^T\ E\right)\left(\frac{1 + \frac{n_p}{N}}{1 - \frac{n_p}{N}}\right)$$ where: • n_p is the number of free parameters in the model. n_p includes the number of estimated initial states. • N is the number of samples in the estimation dataset. • E is the N-by-n_y matrix of prediction errors, where n_y is the number of output channels.
AIC	A raw measure of Akaike's Information Criterion, defined as: $$AIC = N * log\left(det\left(\frac{1}{N}E^T\ E\right)\right) + 2 * n_p + N\left(n_y * log(2\pi) + 1\right)$$

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

See Also

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

More About

- "System Identification Overview"
- "Simulate and Predict Identified Model Output" on page 17-6
- "Assigning Estimation Weightings" on page 6-16
- "Modeling Multiple-Output Systems" on page 1-32
- "Regularized Estimates of Model Parameters" on page 1-34

Regularized Identification of Dynamic Systems

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

What is Regularization

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

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

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

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

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

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

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

Bias - Variance Tradeoff in FIR modeling

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

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

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

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

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

Its impulse response is shown in Figure 1:

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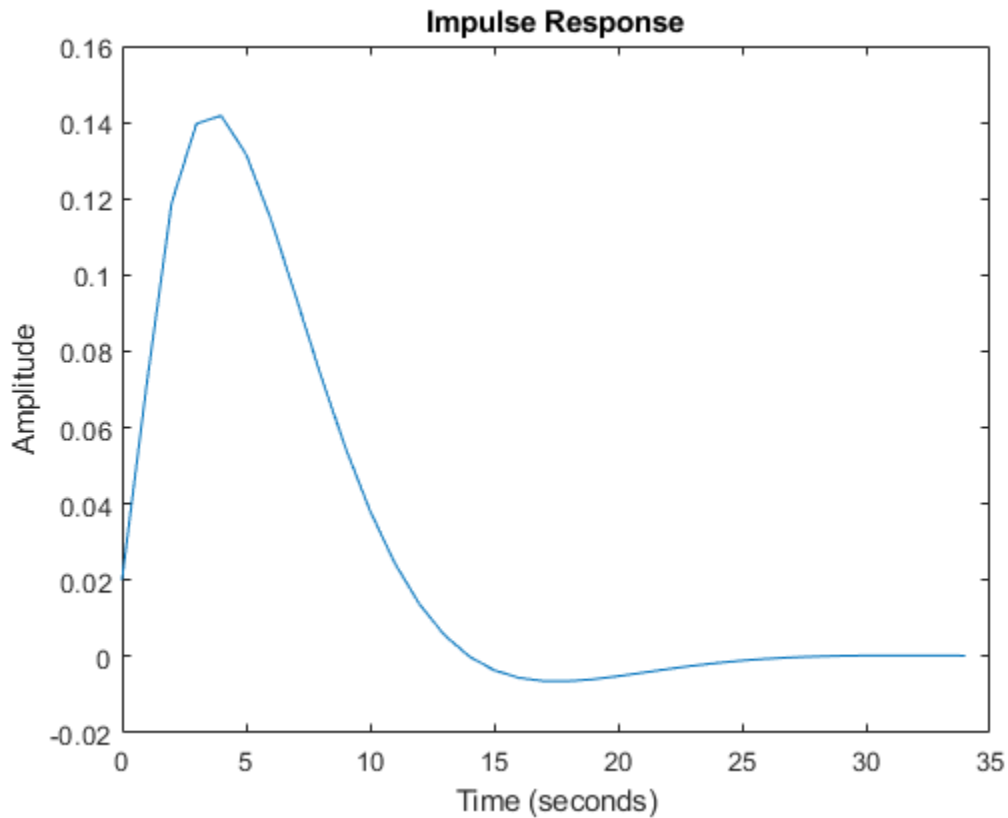

Figure 1: The true impulse response.

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

```
load regularizationExampleData.mat eData
plot(eData)
```
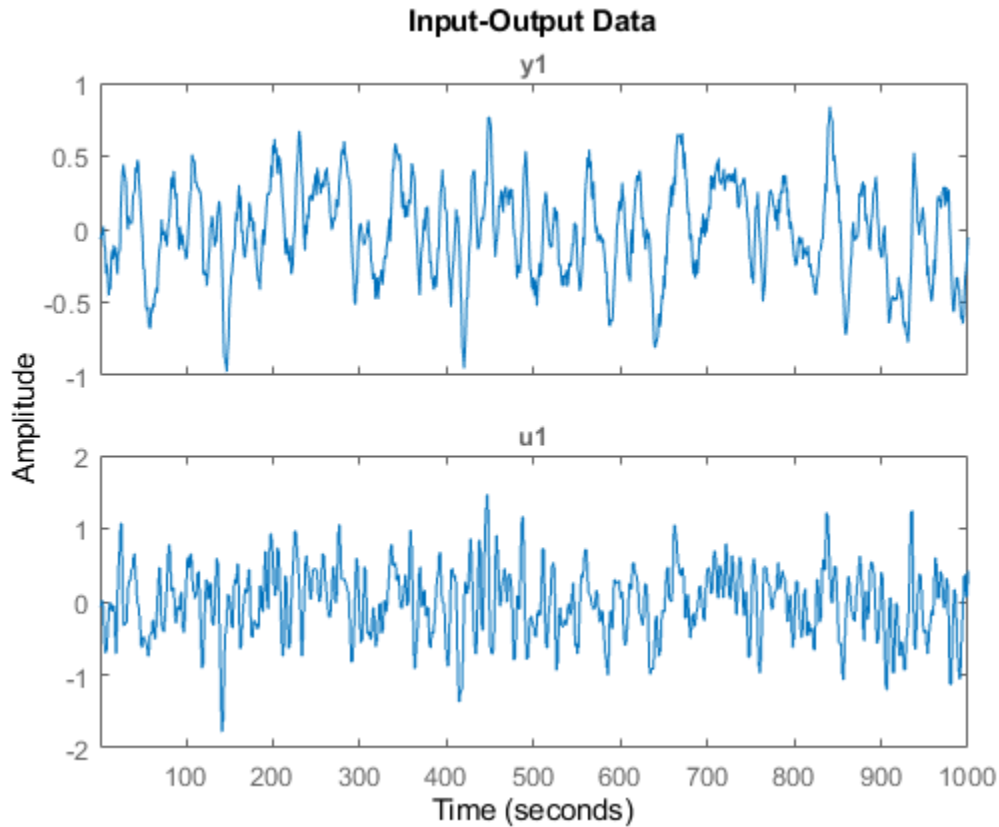
Figure 2: The data used for estimation.

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

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

[y13,~,~,y13sd] = impulse(m13,t);
plot(t,y0,t,y13)
xlabel('Time (seconds)'), ylabel('Amplitude'), title('Impulse Response')
legend('True system','13:th order FIR model')
```
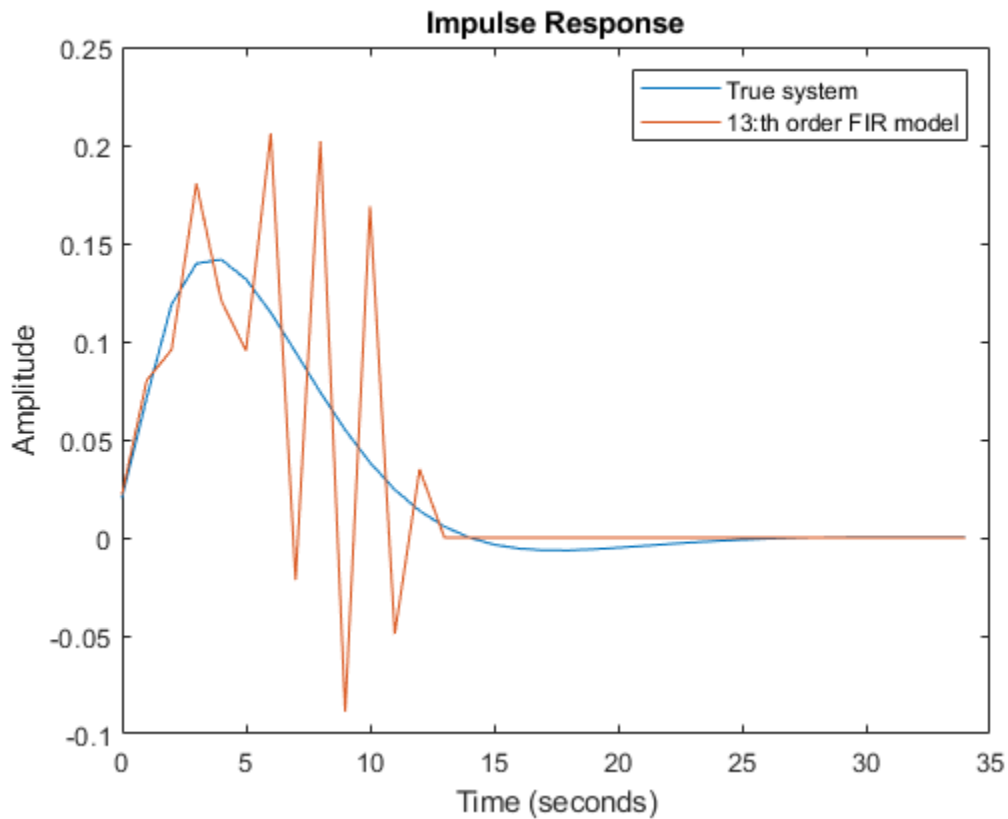
Figure 3: The true impulse response together with the estimate for order `nb = 13`.

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

```
plot(t,y0,t,y13,t,y13+y13sd,'r:',t,y13-y13sd,'r:')
xlabel('Time (seconds)'), ylabel('Amplitude'), title('Impulse Response')
legend('True system','13:th order FIR model','Bounds')
```
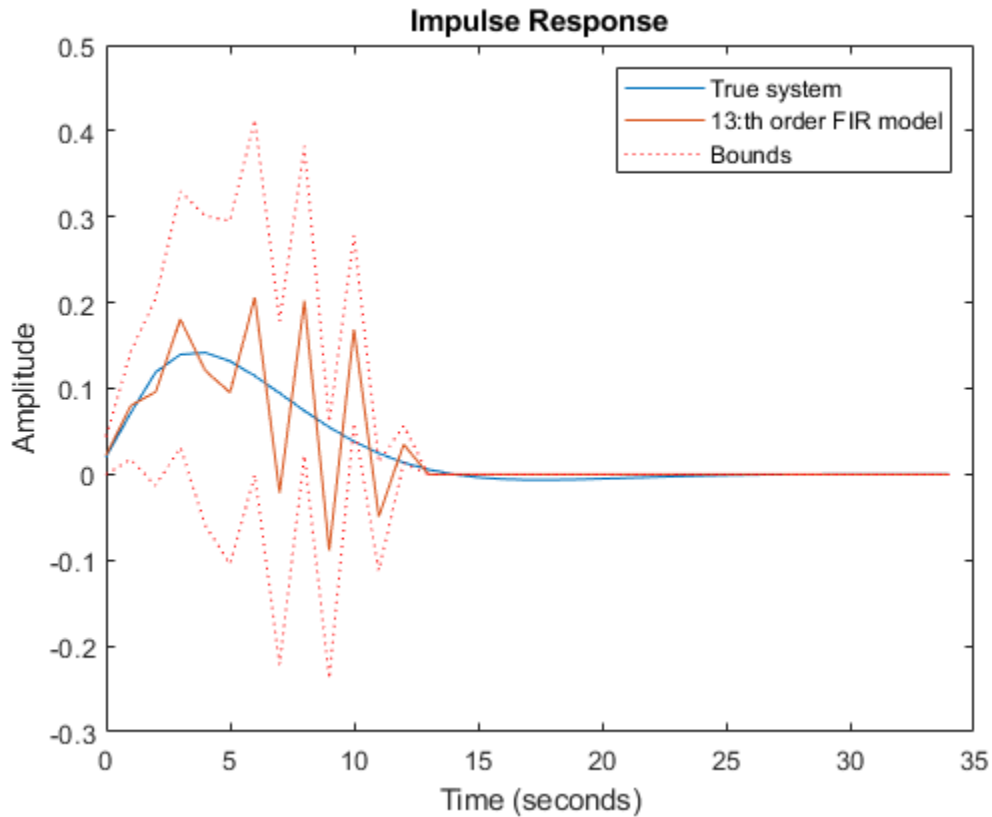
Figure 4: Estimated response with confidence bounds corresponding to 1 s.d.

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

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

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

```
[y50r,~,~,y50rsd] = impulse(m50r,t);
plot(t,y0,t,y50r,t,y50r+y50rsd,'r:',t,y50r-y50rsd,'r:')
xlabel('Time (seconds)'), ylabel('Amplitude'), title('Impulse Response')
legend('True system','50:th order regularized estimate')
```
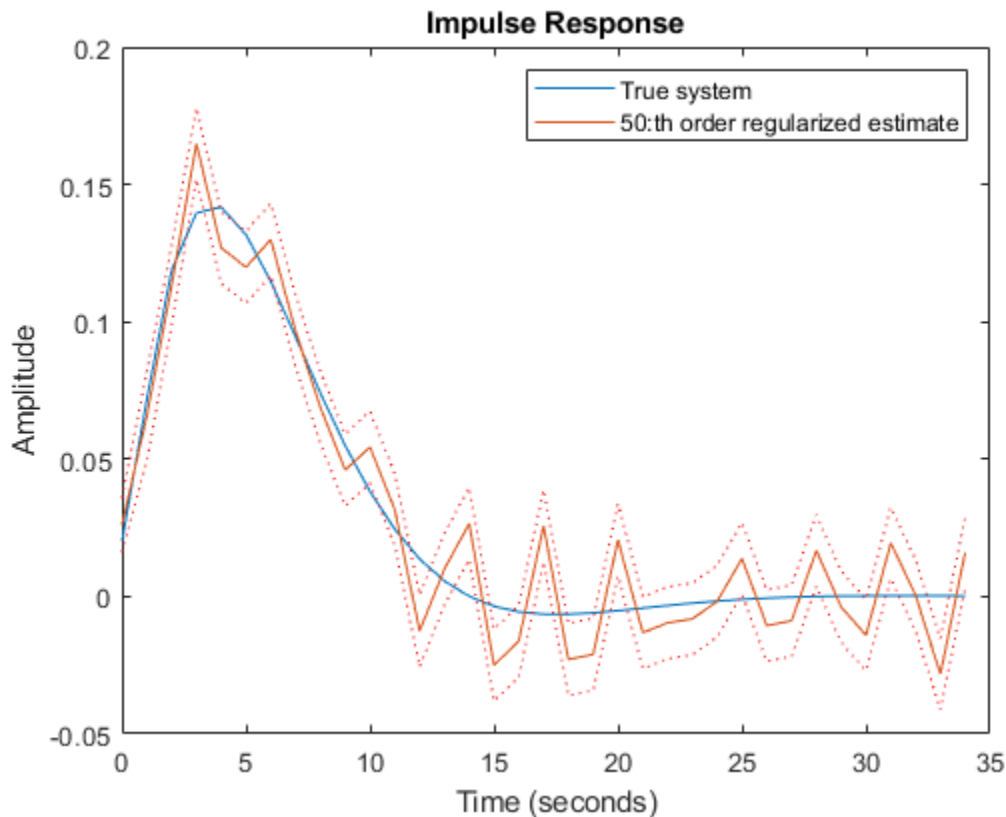
Figure 5: The true impulse response together with the ridge-regularized estimate for order `nb` = 50.

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

Automatic Determination of Regularization Constants for FIR Models

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

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

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

Using the estimated values of R and λ in ARX leads to an error norm of 0.0461 and the response is shown in Figure 6. This kind of tuned regularization is what is achieved also by the `impulseest` command. As the figure shows, the fit to the impulse response as well as the variance is greatly

reduced as compared to the unregularized estimates. The price is a bias in the response estimate, which seems to be insignificant for this example.

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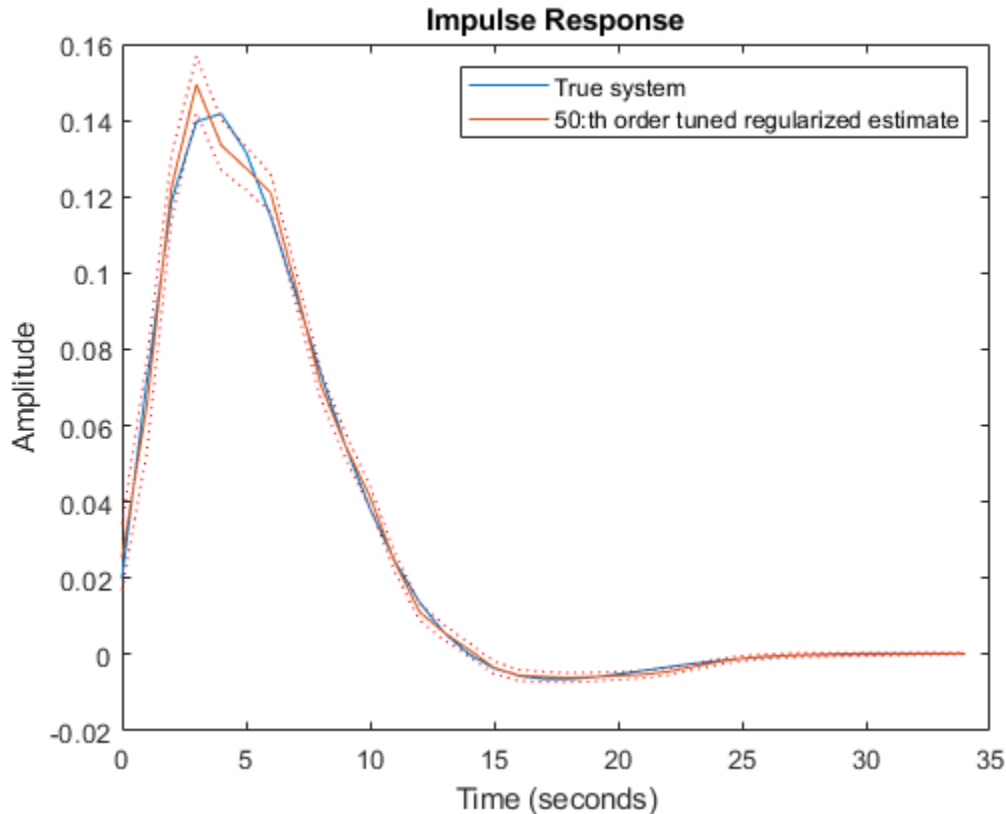

Figure 6: The true impulse response together with the tuned regularized estimate for order `nb` = 50.

Using Regularized ARX-models for Estimating State-Space Models

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

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

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

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

clf
subplot(211)
```

```
plot(t, yG)
title('Impulse Response of G(q)'), ylabel('Amplitude')

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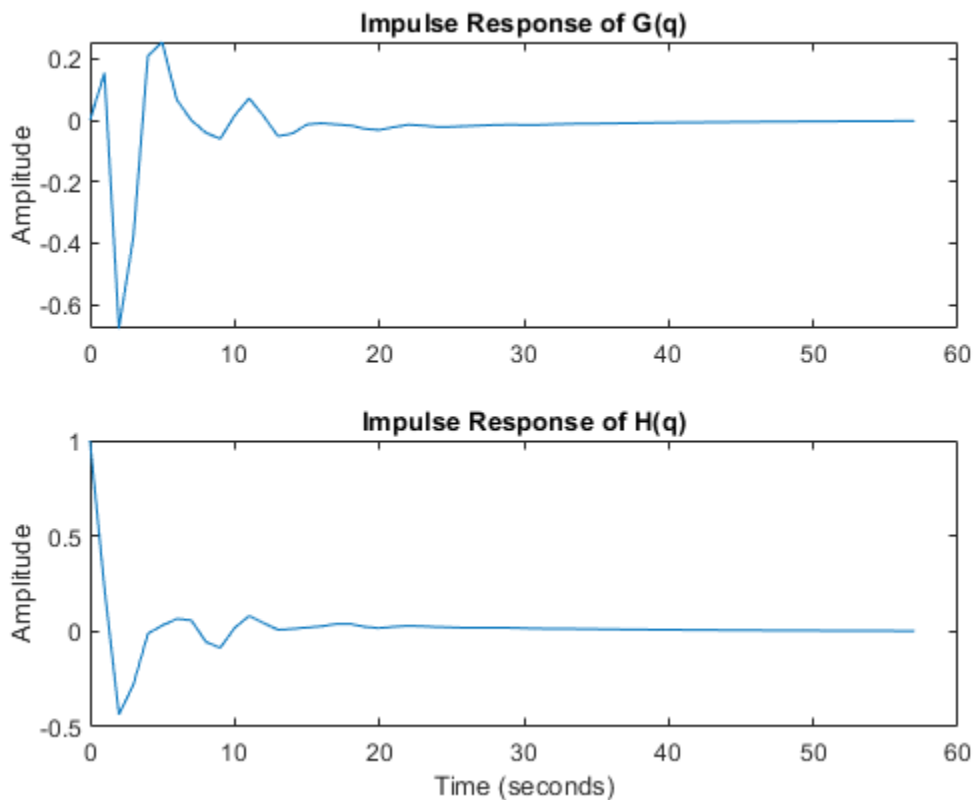

Figure 7: The impulse responses of G(q) (top) and H(q) (bottom).

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

```
load regularizationExampleData.mat m0simdata
clf
plot(m0simdata)
```
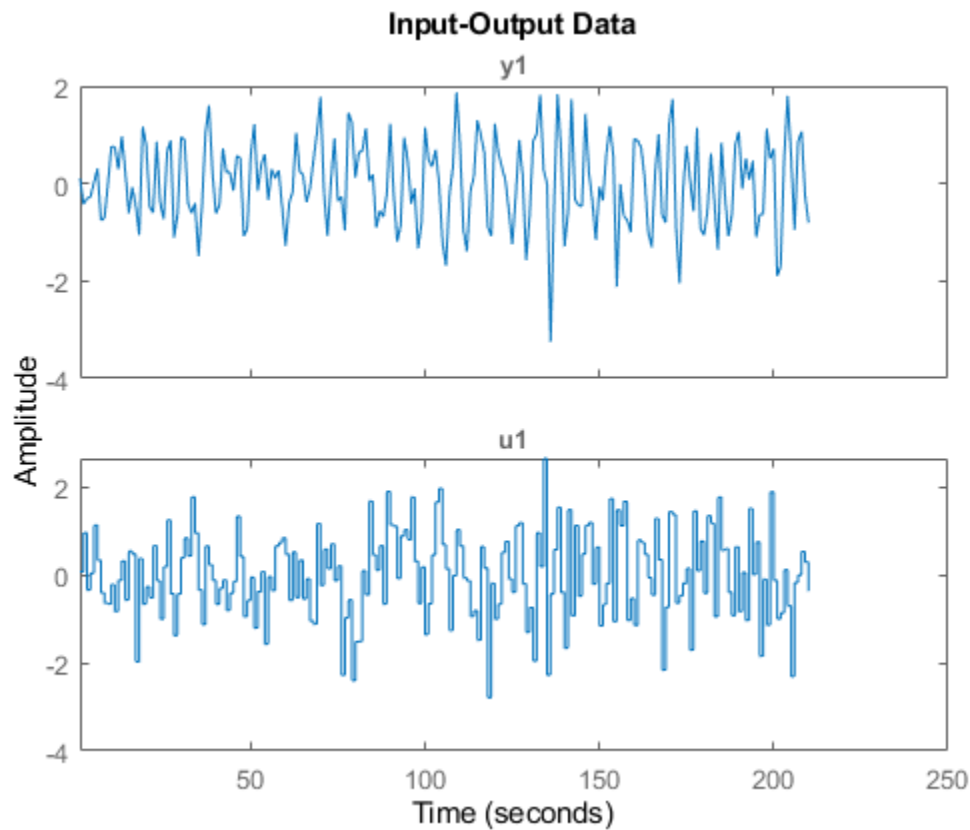
Figure 8: The data to be used for estimation.

To estimate the impulse responses of `m0` from these data, we can naturally employ state-space models in the innovations form (or equivalently ARMAX models) and compute the impulse response using the `impulse` command as before. For computing the state-space model, we can use a syntax such as:

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

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

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

- *Use the Akaike criterion AIC*: Estimate models for orders `k = 1,...,maxo` using the whole data set, and then pick that model that minimizes `aic(mk)`.

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

The "Oracle" test: In addition to the CV and AIC tests, one can also check for what order `k` the fit between the true impulse response of `G(q)` (or `H(q)`) and the estimated model is maximized. This of course requires knowledge of the true system `m0` which is impractical. However, if we do carry on this comparison for our example where `m0` is known, we find that `k = 12` gives the best fit of estimated

model's impulse response to that of m0 (=|G(q)|). Similarly, we find that k = 3 gives the best fit of estimated model's noise component's impulse response to that of the noise component of m0 (=|H(q)|). The Oracle test sets a reference point for comparison of the quality of models generated by using various orders and regularization parameters.

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

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

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

plot(t,yG, t,y12, t,y15, t,y3)
xlabel('Time (seconds)'), ylabel('Amplitude'), title('Impulse Response')
legend('True G(q)',...
    sprintf('Oracle choice: %2.4g%%',100*(1-goodnessOfFit(y12,yG,'NRMSE'))),...
    sprintf('CV choice: %2.4g%%',100*(1-goodnessOfFit(y15,yG,'NRMSE'))),...
    sprintf('AIC choice: %2.4g%%',100*(1-goodnessOfFit(y3,yG,'NRMSE'))))
```
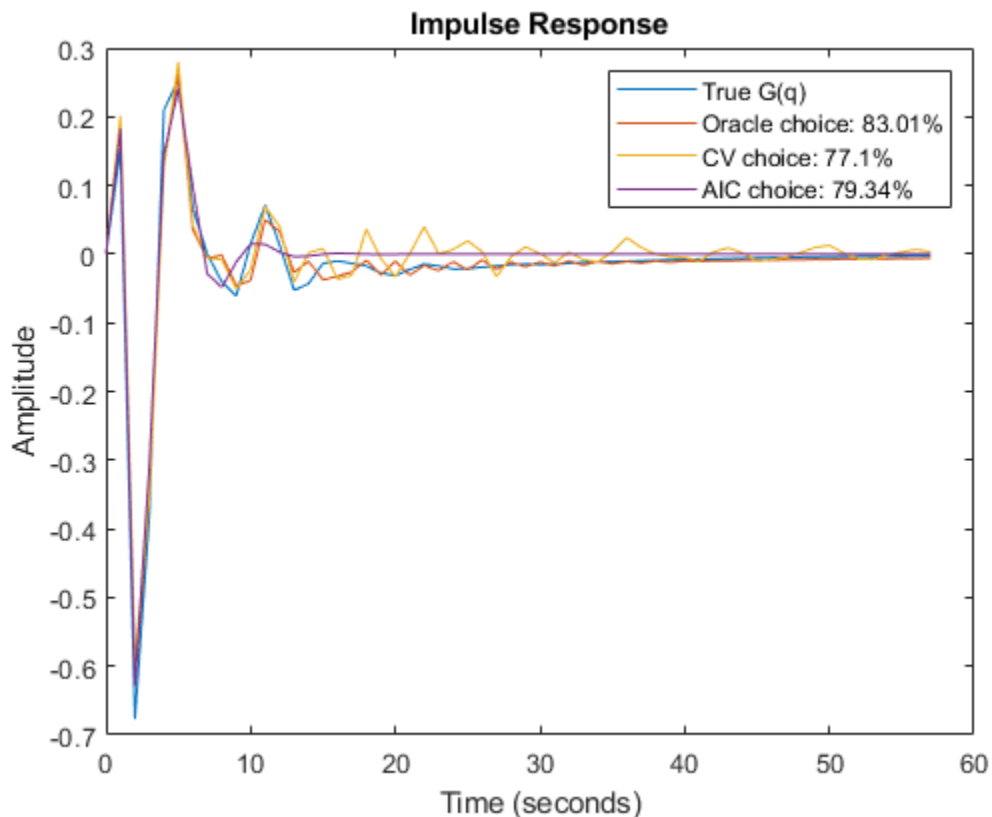

Figure 9: The true impulse response of G(q) compared to estimated models of various orders.

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

```
plot(t,yH, t,yH3, t,yH15, t,yH3)
xlabel('Time (seconds)'), ylabel('Amplitude'), title('Impulse Response')
legend('True H(q)',...
    sprintf('Oracle choice: %2.4g%%',100*(1-goodnessOfFit(yH3,yH,'NRMSE'))),...
    sprintf('CV choice: %2.4g%%',100*(1-goodnessOfFit(yH15,yH,'NRMSE'))),...
    sprintf('AIC choice: %2.4g%%',100*(1-goodnessOfFit(yH3,yH,'NRMSE'))))
```
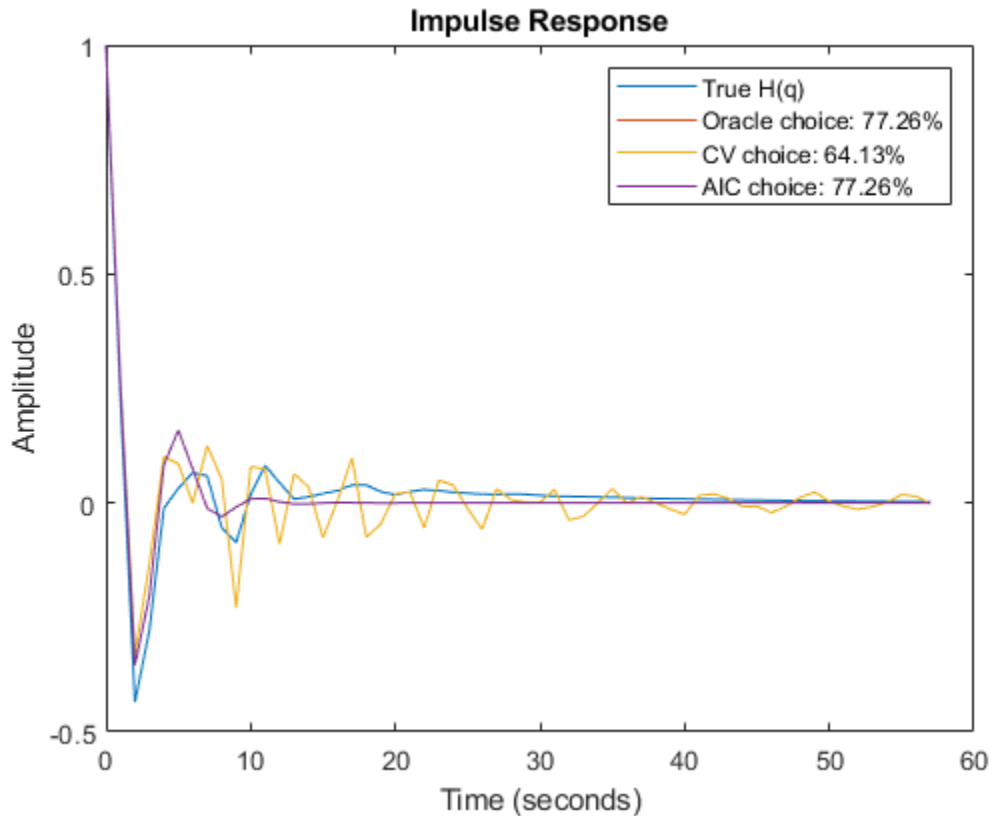

Figure 10: The true impulse response of `H(q)` compared to estimated noise models of various orders.

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

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

```
aopt = arxOptions;
[Lambda, R] = arxRegul(m0simdata, [5 60 0], arxRegulOptions('RegularizationKernel','TC'));
aopt.Regularization.R = R;
aopt.Regularization.Lambda = Lambda;
mr = arx(m0simdata, [5 60 0], aopt);
nmr = noise2meas(mr);
ymr = impulse(mr, t);
yHmr = impulse(nmr, t);
fprintf('Goodness of fit for ARX model is: %2.4g%%\n',100*(1-goodnessOfFit(ymr,yG,'NRMSE')))
fprintf('Goodness of fit for noise component of ARX model is: %2.4g%%\n',100*(1-goodnessOfFit(yHm
```

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

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

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

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

```
plot(t,yG, t,y12, t,ymr, t,y7mr)
xlabel('Time (seconds)'), ylabel('Amplitude'), title('Impulse Response')
legend('True G(q)',...
    sprintf('Oracle choice: %2.4g%%',100*(1-goodnessOfFit(y12,yG,'NRMSE'))),...
    sprintf('High order regularized: %2.4g%%',100*(1-goodnessOfFit(ymr,yG,'NRMSE'))),...
    sprintf('Reduced order: %2.4g%%',100*(1-goodnessOfFit(y7mr,yG,'NRMSE'))))
```
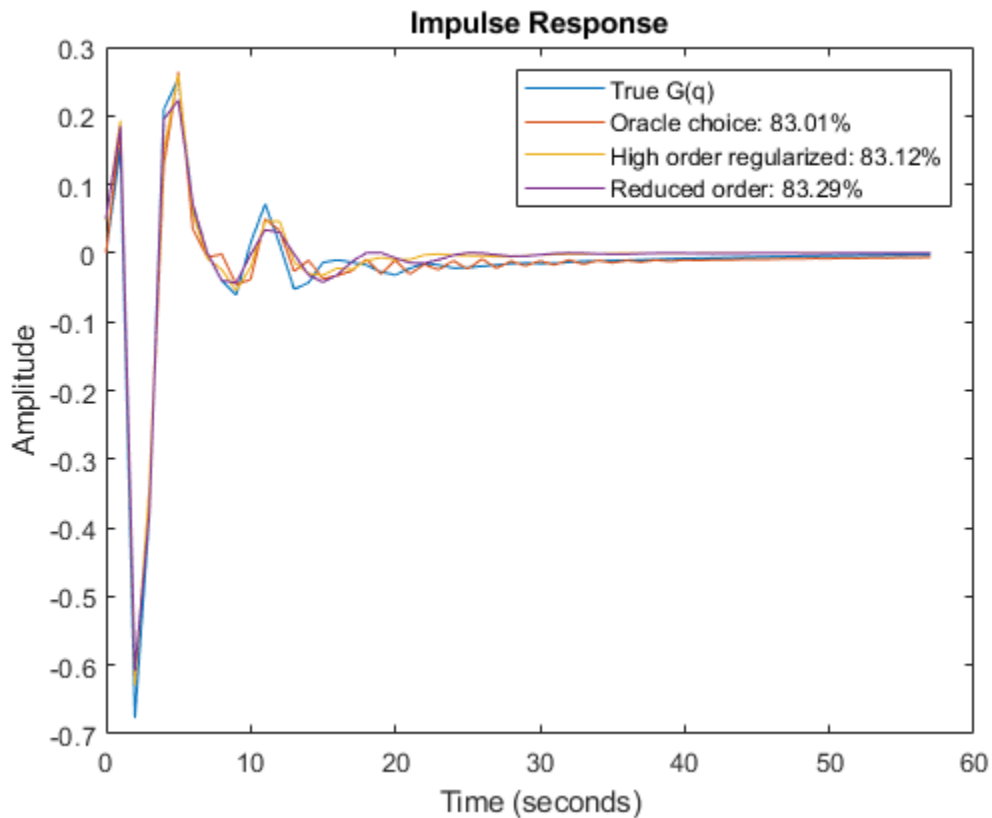

Figure 11: The regularized models compared to the Oracle choice for $G(q)$.

```
plot(t,yH, t,yH3, t,yHmr, t,y7Hmr)
xlabel('Time (seconds)'), ylabel('Amplitude'), title('Impulse Response')
```

```
legend('True H(q)',...
    sprintf('Oracle choice: %2.4g%%',100*(1-goodnessOfFit(yH3,yH,'NRMSE'))),...
    sprintf('High order regularized: %2.4g%%',100*(1-goodnessOfFit(yHmr,yH,'NRMSE'))),...
    sprintf('Reduced order: %2.4g%%',100*(1-goodnessOfFit(y7Hmr,yH,'NRMSE'))))
```
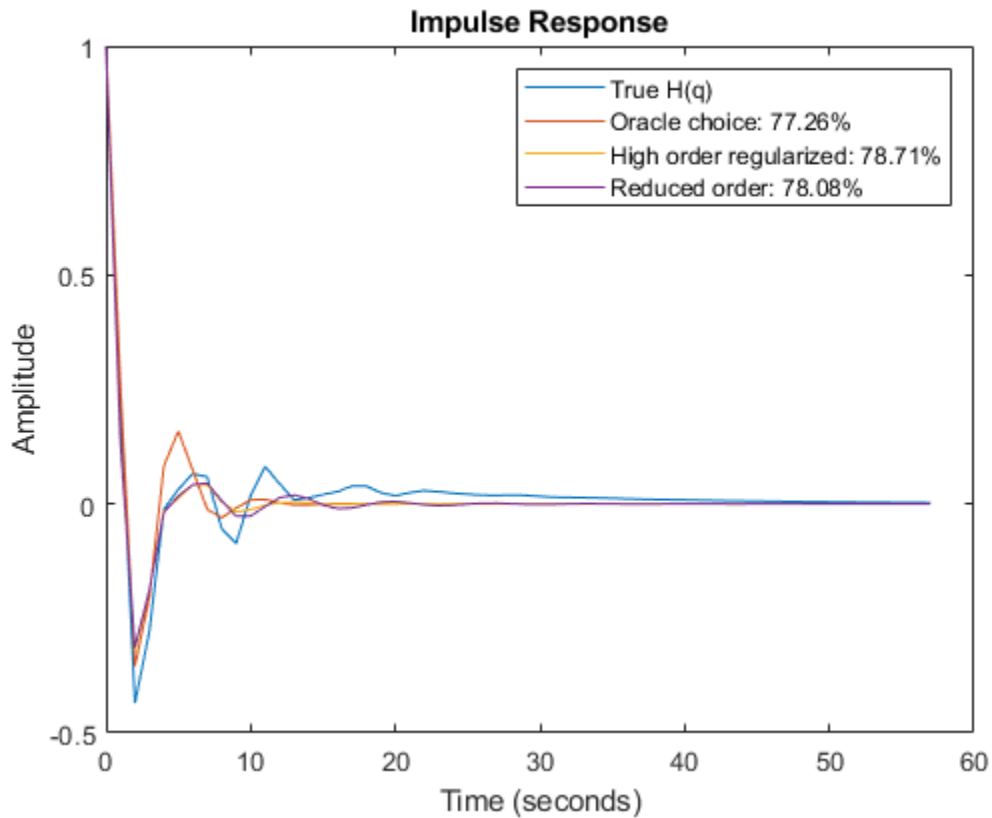

Figure 12: The regularized models compared to the Oracle choice for `H(q)`.

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

State Space Model Estimation by Regularized Reduction Technique

The above steps of estimating a high-order ARX model, followed by a conversion to state-space and reduction to the desired order can be automated using the `ssregest` command. `ssregest` greatly simplifies this procedure while also facilitating other useful options such as search for optimal order and fine tuning of model structure by specification of feedthrough and delay values. Here we simply reestimate the reduced model similar to `mred7` using `ssregest`:

```
opt = ssregestOptions('ARXOrder',[5 60 0]);
mred7_direct = ssregest(m0simdata, 7, 'Feedthrough', true, opt);
compare(m0simdata, mred7, mred7_direct)
```
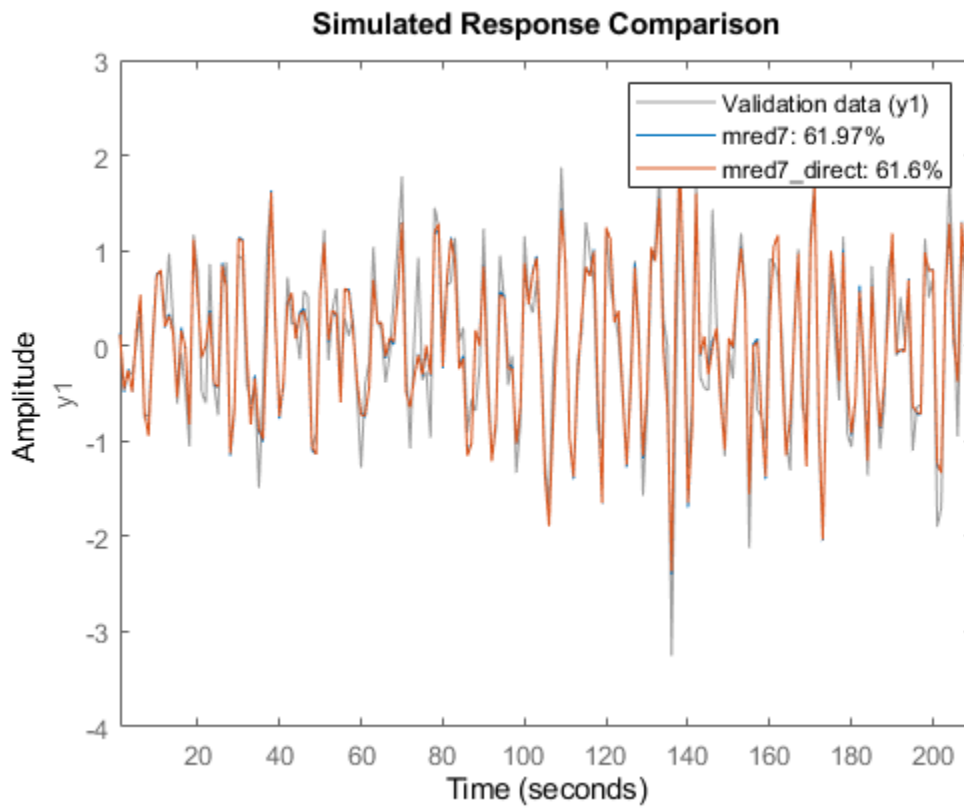
Figure 13: Comparing responses of state space models to estimation data.

```
h = impulseplot(mred7, mred7_direct, 40);
showConfidence(h,1) %  1 s.d. "zero interval"
hold on
s = stem(t,yG,'r');
s.DisplayName = 'True G(q)';
legend('show')
```
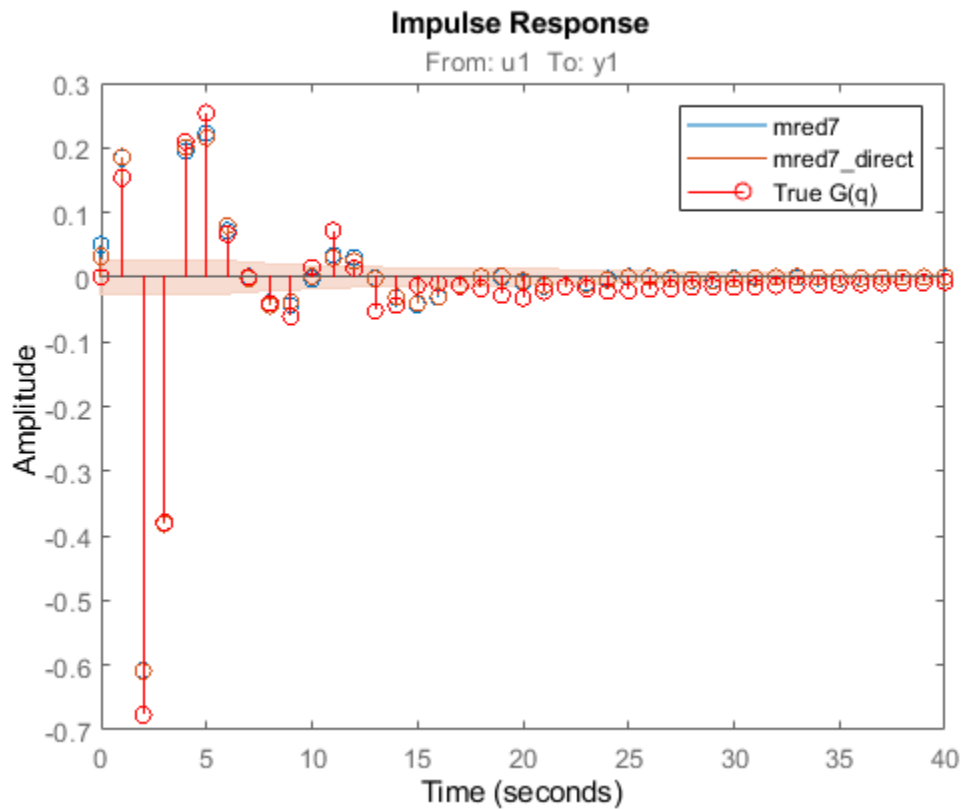
Figure 14: Comparing impulse responses of state space models.

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

Basic Bias - Variance Tradeoff in Grey Box Models

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

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

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

In state-space form we have:

$$\dot{x}_1 = x_2$$

$$\dot{x}_2 = -1/\tau \cdot x_2$$

$$y = x + e$$

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

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

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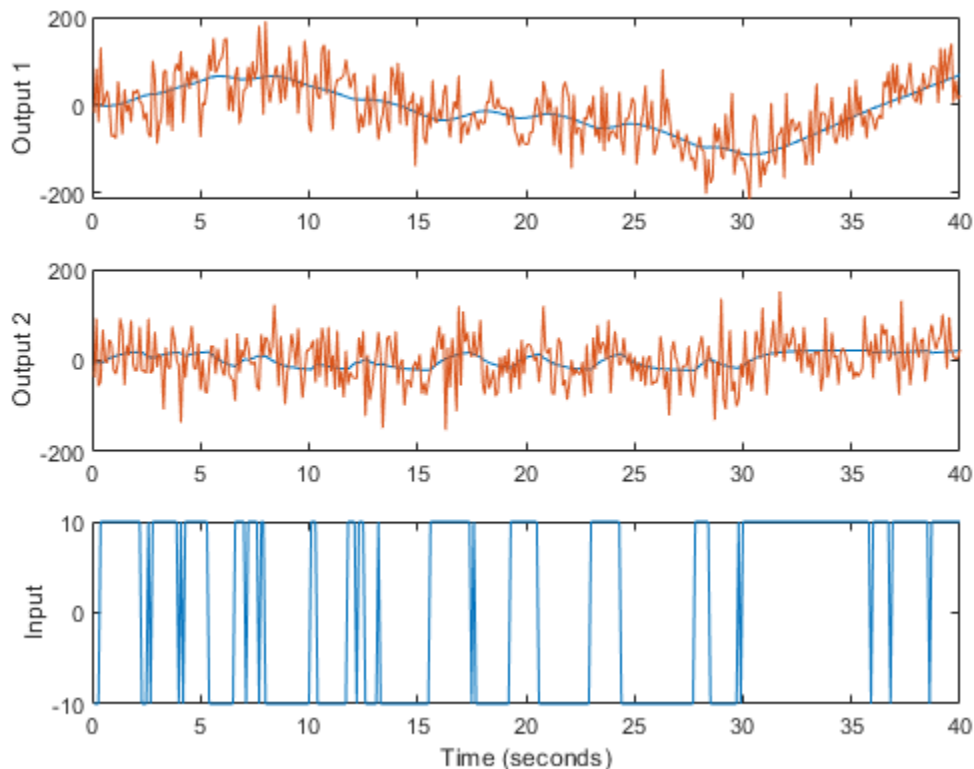

Figure 15: The noisy data to be used for grey box estimation superimposed over noise-free simulation data to be used for qualifications. From top to bottom: Angle, Angular Velocity, Input voltage.

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

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

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

%   Copyright 2013 The MathWorks, Inc.

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

An `idgrey` object is then created as:

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

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

```
m = greyest(motorData, mi)


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

  A =
           x1        x2
    x1        0         1
    x2        0   -1.741

  B =
           u1
    x1        0
    x2   3.721

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

```
 D =
        u1
   y1    0
   y2    0

 K =
        y1   y2
   x1    0    0
   x2    0    0

 Model parameters:
   G = 2.138
   Tau = 0.5745

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

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

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

```
copt = compareOptions('InitialCondition', 'z');
[ymi, fiti] = compare(motorData, mi, copt);
[ym, fit] = compare(motorData, m, copt);
t = motorData.SamplingInstants;
subplot(211)
plot(t, [motorData.y(:,1), ymi.y(:,1), ym.y(:,1)])
axis tight
ylabel('Output 1')
legend({'Measured output',...
    sprintf('Initial: %2.4g%%',fiti(1)),...
    sprintf('Estimated: %2.4g%%',fit(1))},...
    'Location','BestOutside')
subplot(212)
plot(t, [motorData.y(:,2), ymi.y(:,2), ym.y(:,2)])
ylabel('Output 2')
axis tight
legend({'Measured output',...
    sprintf('Initial: %2.4g%%',fiti(2)),...
    sprintf('Estimated: %2.4g%%',fit(2))},...
    'Location','BestOutside')
```
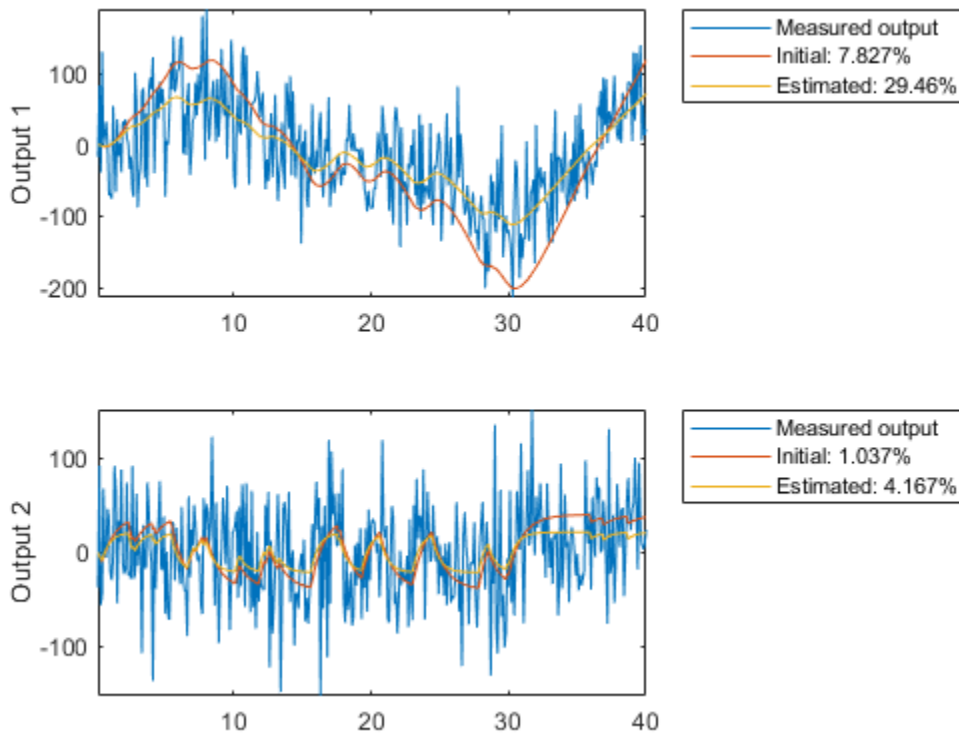
Figure 16: Measured output and model outputs for initial and estimated models.

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

```
[ymi, fiti] = compare(motorData_NoiseFree, mi, copt);
[ym, fit] = compare(motorData_NoiseFree, m, copt);
subplot(211)
plot(t, [motorData_NoiseFree.y(:,1), ymi.y(:,1), ym.y(:,1)])
axis tight
ylabel('Output 1')
legend({'Noise-free output',...
   sprintf('Initial: %2.4g%%',fiti(1)),...
   sprintf('Estimated: %2.4g%%',fit(1))},...
   'Location','BestOutside')
subplot(212)
plot(t, [motorData_NoiseFree.y(:,2), ymi.y(:,2), ym.y(:,2)])
ylabel('Output 2')
axis tight
legend({'Noise-free output',...
   sprintf('Initial: %2.4g%%',fiti(2)),...
   sprintf('Estimated: %2.4g%%',fit(2))},...
   'Location','BestOutside')
```
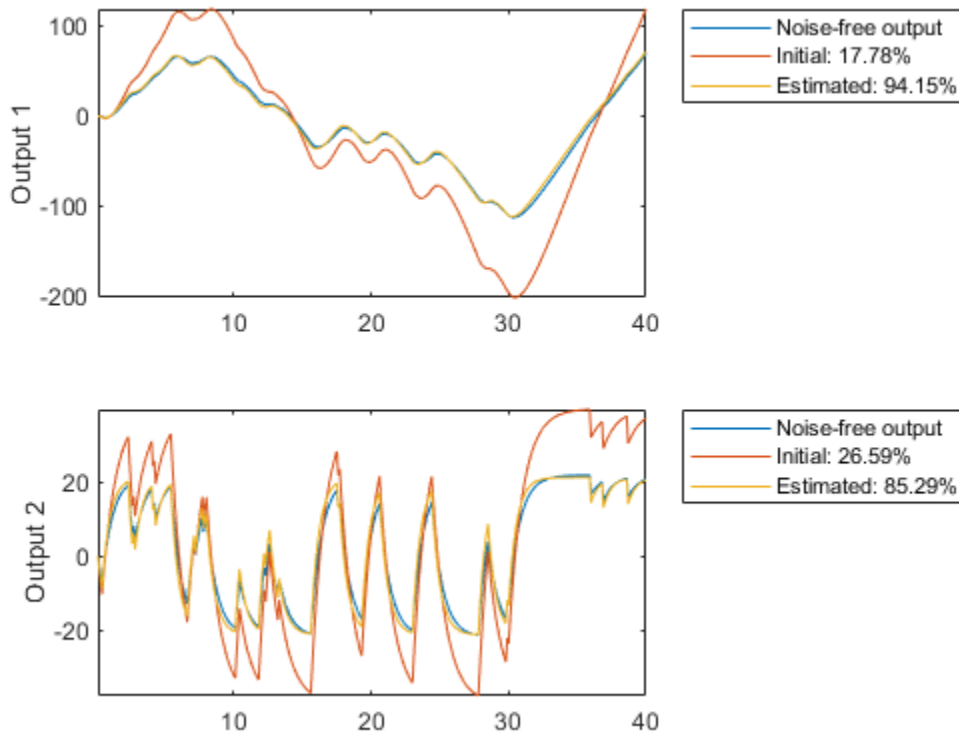
Figure 17: Noise-free output and model outputs for initial and estimated models.

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

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

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

  A =
           x1       x2
     x1      0        1
     x2      0   -1.119

  B =
           u1
     x1      0
     x2  2.447
```

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

   D =
        u1
    y1   0
    y2   0

   K =
        y1   y2
    x1   0    0
    x2   0    0

  Model parameters:
   G = 2.187
   Tau = 0.8938

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

Status:
Estimated using GREYEST on time domain data "motorData".
Fit to estimation data: [29.34;3.848]%
FPE: 6.135e+06, MSE: 4933
```

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

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

```
sprintf('Estimated: %2.4g%%',fit(2)),...
sprintf('Regularized: %2.4g%%',fitr(2))},...
'Location','BestOutside')
```
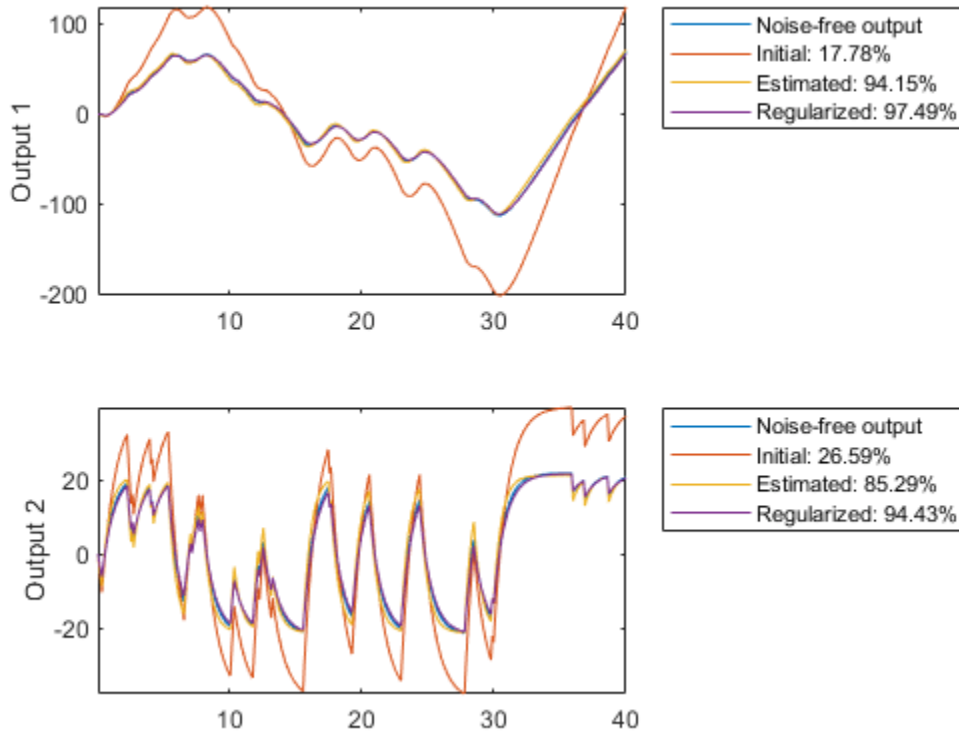

Figure 18: Noise-Free measured output and model outputs for initial, estimated and regularized models.

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

```
clf
showConfidence(bodeplot(m,mr,logspace(-1,1.4,100)),3) % 3 s.d. region
legend('show')
```
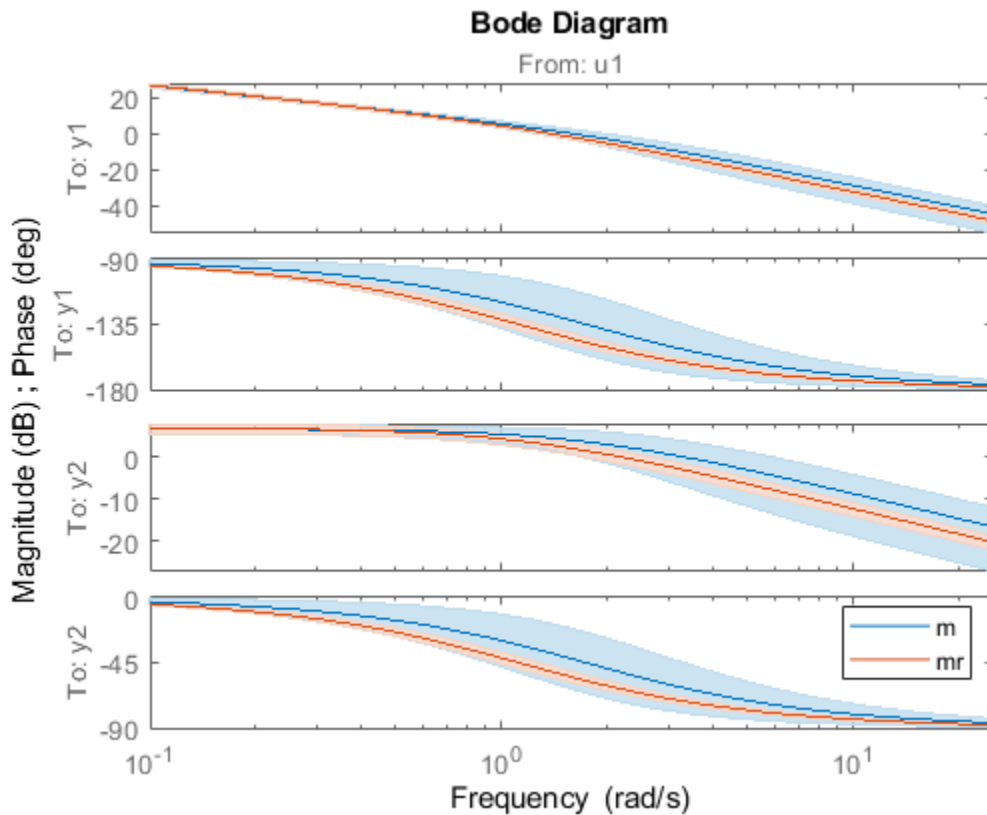
Figure 19: Bode plot of `m` and `mr` with confidence bounds

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

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

Use of Regularization to Robustify Large Nonlinear Models

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

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

compare(nldata, m10, m20) % compare responses of m10, m20 to measured response
```
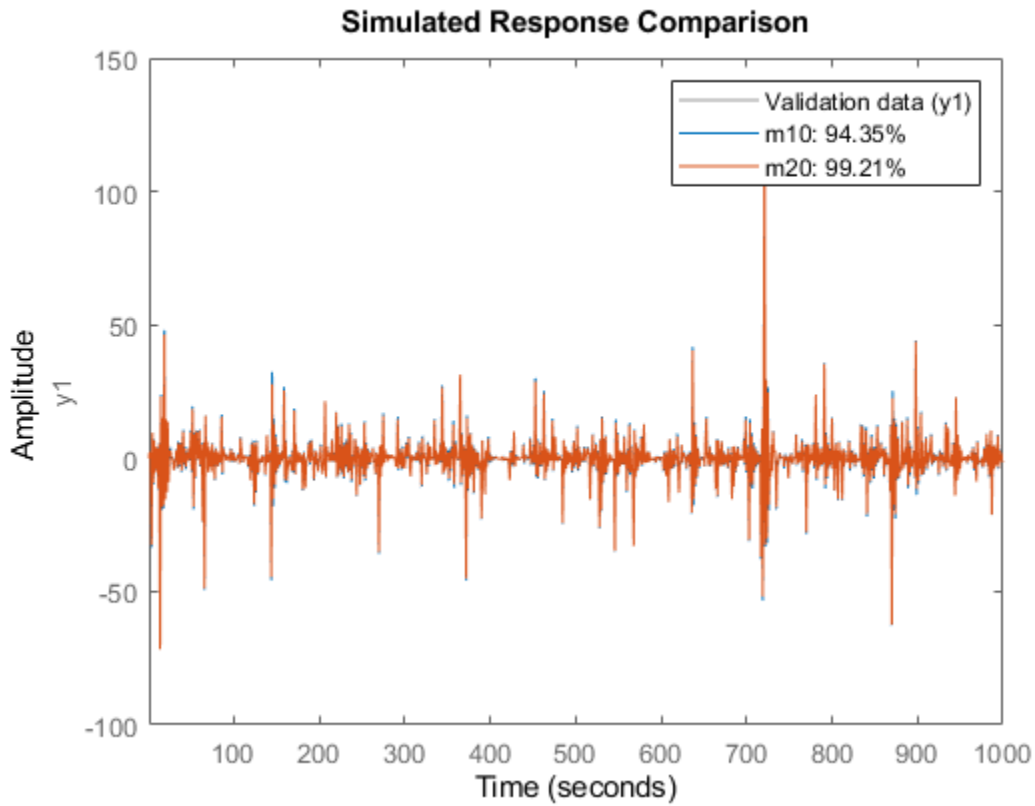
Figure 20: Comparison plot for models `m10` and `m20`.

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

```
Number of parameters (m10, m20, m30): [54 104 154]
```
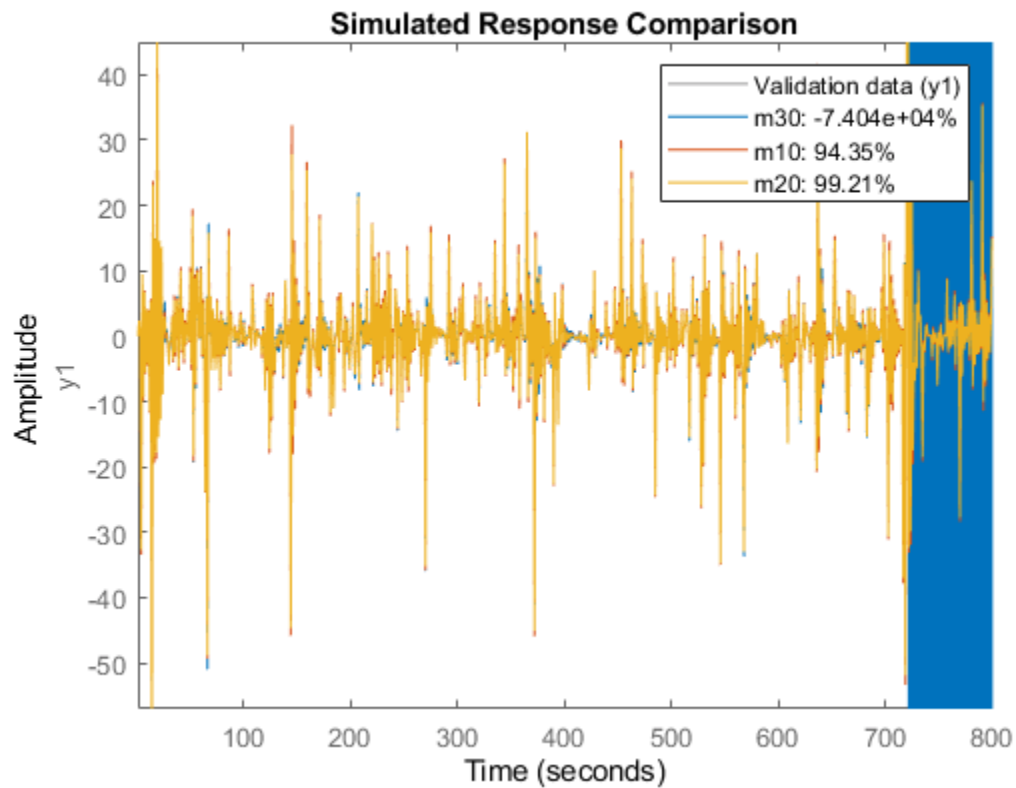
Figure 21: Comparison plot for models `m10`, `m20` and `m30`.

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

```
opt.Regularization.Lambda = 1e-8;
m30r = nlarx(nldata, [1 2 1], idSigmoidNetwork(30), opt);
compare(nldata, m30r, m10, m20)
```
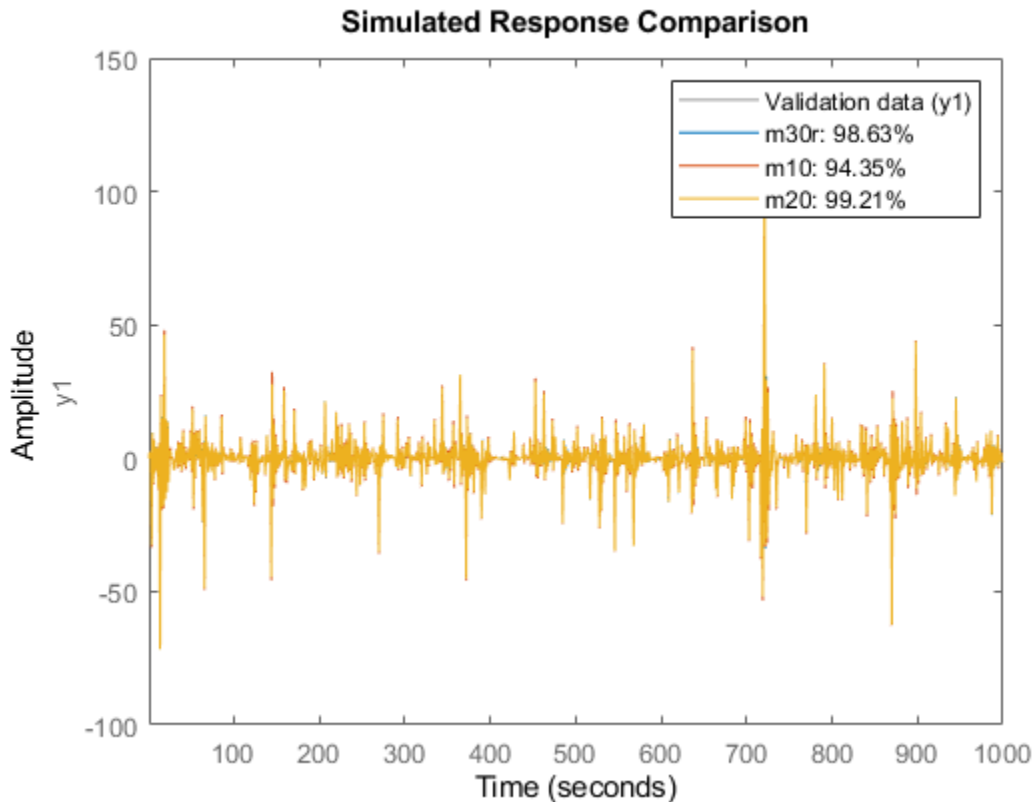
Figure 22: Comparison plot for models `m10`, `m20` and regularized model `m30r`.

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

Conclusions

We discussed the benefit of regularization for estimation of FIR models, linear grey-box models and Nonlinear ARX models. Regularization can have significant impact on the quality of the identified model provided the regularization constants `Lambda` and `R` are chosen appropriately. For ARX models, this can be done very easily using the `arxRegul` function. These automatic choices also feed into the dedicated state-space estimation algorithm `ssregest`.

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

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

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 1-27.

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

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

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

and a corresponding output sequence

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

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

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

Supported Data	Description
Time-domain I/O data	One or more input variables $u(t)$ and one or more output variables $y(t)$, sampled as a function of time. Time-domain data can be either real or complex
Time-series data	Contains one or more outputs $y(t)$ and no measured input. Can be time-domain or frequency-domain data.
Frequency-domain data	Fourier transform of the input and output time-domain signals. The data is the set of input and output signals in frequency domain; the frequency grid need not be uniform.
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 2-102 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 2-8. Then, import it in the System Identification app or represent it as a data object for estimating models at the command line.

- Generating an input signal with desired characteristics, such as a random Gaussian or binary signal or a sinusoid, using `idinput`. Then, generate an output signal using this input to simulate a model with known coefficients. For more information, see "Generate Data Using Simulation" on page 2-98.

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

- Logging signals from Simulink models.

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

Ways to Prepare Data for System Identification

Before you can perform any task in this toolbox, your data must be in the MATLAB workspace. You can import the data from external data files or manually create data arrays at the command line. For more information about importing data, see "Representing Data in MATLAB Workspace" on page 2-8.

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

Represent data for system identification

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

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

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

  For time-domain or frequency-domain data, see "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-34.

  For frequency-response data, see "Representing Frequency-Response Data Using idfrd Objects" on page 2-61.
- To simulate data with and without noise, see "Generate Data Using Simulation" on page 2-98.

Analyze data quality

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

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

  See "How to Plot Data in the App" on page 2-67 and "How to Plot Data at the Command Line" on page 2-71.
- 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 2-73.

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 2-77.
- Offsets and drifts in signal levels (low-frequency disturbances).

  See "Handling Offsets and Trends in Data" on page 2-81 for information about subtracting means and linear trends, and "Filtering Data" on page 2-92 for information about filtering.
- High-frequency disturbances above the frequency interval of interest for the system dynamics.

See "Resampling Data" on page 2-86 for information about decimating and interpolating values, and "Filtering Data" on page 2-92 for information about filtering.

Select a subset of your data

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

To learn more about selecting data, see "Select Subsets of Data" on page 2-74.

Combine data from multiple experiments

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

To learn more about creating multiple-experiment data sets, see "Create Multiexperiment Data Sets in the App" on page 2-26 or "Create Multiexperiment Data at the Command Line" on page 2-42.

Requirements on Data Sampling

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

The System Identification app only supports uniformly sampled data.

The System Identification Toolbox product provides limited support for nonuniformly sampled data. For more information about specifying uniform and nonuniform time vectors, see "Constructing an iddata Object for Time-Domain Data" on page 2-34.

Representing Data in MATLAB Workspace

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 2-9.

You can organize time-domain input/output data in one of three data types:

1 A `timetable` that contains a time variable and any number of input and output channel variables

2 A pair of numeric matrices that contain the input and output data separately

3 An `iddata` object that contains properties such as input and output data, sample time, and intersample behavior for continuous models

For all these data types, the general data format requirements, as implemented in the three data types, are as follows:

- For single-input/single-output (SISO) data, the sampled data values must be double column vectors with lengths equal to the number of data samples (measurements).

- For multi-input/multi-output (MIMO) data with N_u inputs and N_y outputs, and N_s number of data samples:

  - The input data must be an N_s-by-N_u matrix

  - The output data must be an N_s-by-N_y matrix

To use time-domain data for identification, you must know the sample time. If you are working with uniformly sampled data, use the actual sample time from your experiment. Each data value is assigned a time instant, which is calculated from the start time and sample time. You can work with nonuniformly sampled data only at the command line by specifying a vector of time instants using the `SamplingInstants` property of `iddata`, as described in "Constructing an iddata Object for Time-Domain Data" on page 2-34 or by using the `time` variable in a `timetable`. Typically, estimation algorithms require uniformly sampled data.

If the data is contained in numeric matrices, the matrices provide no sample-time knowledge. The data must be uniformly sampled in this case. The software assumes the sample time is 1 second unless you specify a different sample time explicitly in the function syntax. Estimating continuous models using matrix-based data is not recommended.

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 app or create a data object for working at the command line. For more information, see "Import Time-Domain Data into the App" on page 2-13, "Data Types in System Identification Toolbox" on page 2-106, and "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-34.

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 2-4.

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

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

To use time-series data for identification, you also need the sample time. If you are working with uniformly sampled data, use the actual sample time from your experiment. Each data value is assigned a sample time, which is calculated from the start time and the sample time. If you are working with nonuniformly sampled data at the command line, you can specify a vector of time instants using the `iddata SamplingInstants` property, as described in "Constructing an iddata Object for Time-Domain Data" on page 2-34, or by using the `time` variable in a `timetable`. 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 app or create a data object for working at the command line. For more information, see "Import Time-Domain Data into the App" on page 2-13 and "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-34.

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

Frequency-Domain Data Representation

Frequency-domain data consists of either transformed input and output time-domain signals on page 2-9 or system frequency response on page 2-11 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 sample time *T*, the frequency-domain output $Y(e^{iw})$ and input $U(e^{iw})$ is the time-discrete Fourier transform (TDFT):

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

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

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

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

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

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

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 2-4.

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 app or create a data object for working at the command line. For more information, see "Importing Frequency-Domain Input/Output Signals into the App" on page 2-15 and "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-34.

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 sample time *T* to make the frequency function periodic with the sampling frequency $2\pi/T$.

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

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

For more information on how to obtain identification data, see "Ways to Obtain Identification Data" on page 2-4.

How to Represent Frequency-Response Data in MATLAB

You can represent frequency-response data in two ways:

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

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

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

Frequency-Response Data Representation	For Single-Input Single-Output (SISO) Data	For Multi-Input Multi-Output (MIMO) Data
Complex Values	• Frequency function must be a column vector. • Frequency values must be a column vector.	• Frequency function must be an N_y-by-N_u-by-N_f array, where N_u is the number of inputs, N_y is the number of outputs, and N_f is the number of frequency measurements. • Frequency values must be a column vector.
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 app or create a data object for working at the command line. For more information about importing data into the app, see "Importing Frequency-Response Data into the App" on page 2-16. To learn more about creating a data object, see "Representing Frequency-Response Data Using idfrd Objects" on page 2-61.

Import Time-Domain Data into the App

Before you can import time-domain data into the **System Identification** app, you must import the data into the MATLAB workspace, as described in "Time-Domain Data Representation" on page 2-8.

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

To import data into the app:

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

```
systemIdentification
```

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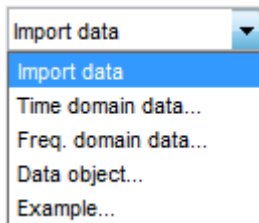
3 Specify the following options:

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

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

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

- **Data name** — Enter the name of the data set, which appears in the System Identification app window after the import operation is completed.

- **Starting time** — Enter the starting value of the time axis for time plots.

- **Sample time** — Enter the actual sample time in the experiment. For more information about this setting, see "Specifying the Data Sample Time" on page 2-21.

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

---

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

Input Properties

- **InterSample** — This options specifies the behavior of the input signals between samples during data acquisition. It is used when transforming models from discrete-time to continuous-time and when resampling the data.

  - zoh (zero-order hold) indicates that the input was piecewise-constant during data acquisition.
  - foh (first-order hold) indicates that the input was piecewise-linear during data acquisition.
  - bl (bandwidth-limited behavior) specifies that the input signal has zero power above the Nyquist frequency.

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

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

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

Channel Names

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

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

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

Physical Units of Variables

- **Input** — Enter the input units.

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

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

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

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

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

Import Frequency-Domain Data into the App

Importing Frequency-Domain Input/Output Signals into the App

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

Before you can import frequency-domain data into the **System Identification** app, you must import the data into the MATLAB workspace, as described in "Frequency-Domain Input/Output Signal Representation" on page 2-9.

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

To import data into the app:

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

`systemIdentification`

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

3 Specify the following options:

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

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

- **Frequency** — Enter the MATLAB variable name of a vector or a MATLAB expression that represents the frequencies. The expression must evaluate to a column vector.

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

- **Data name** — Enter the name of the data set, which appears in the System Identification app window after the import operation is completed.

- **Frequency unit** — Enter `Hz` for Hertz or keep the `rad/s` default value.

- **Sample time** — Enter the actual sample time in the experiment. For continuous-time data, enter `0`. For more information about this setting, see "Specifying the Data Sample Time" on page 2-21.

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

Input Properties

- **InterSample** — This options specifies the behavior of the input signals between samples during data acquisition. It is used when transforming models from discrete-time to continuous-time and when resampling the data.

  - `zoh` (zero-order hold) indicates that the input was piecewise-constant during data acquisition.

  - `foh` (first-order hold) indicates that the output was piecewise-linear during data acquisition.

- `bl` (bandwidth-limited behavior) specifies that the continuous-time input signal has zero power above the Nyquist frequency (equal to the inverse of the sample time).

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

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

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

Channel Names

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

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

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

Physical Units of Variables

- **Input** — Enter the input units.

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

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

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

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

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

Importing Frequency-Response Data into the App

Prerequisite

Before you can import frequency-response data into the System Identification app, you must import the data into the MATLAB workspace, as described in "Frequency-Response Data Representation" on page 2-11.

Importing Complex-Valued Frequency-Response Data

To import frequency-response data consisting of complex-valued frequency values at specified frequencies:

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

```
systemIdentification
```

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

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

4 Specify the following options:

- **Response** — Enter the MATLAB variable name or a MATLAB expression that represents the complex frequency-response data $G(e^{iw})$.

- **Frequency** — Enter the MATLAB variable name of a vector or a MATLAB expression that represents the frequencies. The expression must evaluate to a column vector.

- **Data name** — Enter the name of the data set, which appears in the System Identification app window after the import operation is completed.

- **Frequency unit** — Enter `Hz` for Hertz or keep the `rad/s` default value.

- **Sample time** — Enter the actual sample time in the experiment. For continuous-time data, enter `0`. For more information about this setting, see "Specifying the Data Sample Time" on page 2-21.

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

Channel Names

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

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

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

Physical Units of Variables

- **Input** — Enter the input units.

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

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

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

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

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

Importing Amplitude and Phase Frequency-Response Data

To import frequency-response data consisting of amplitude and phase values at specified frequencies:

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

```
systemIdentification
```

2-17

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

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

4   Specify the following options:

- **Amplitude** — Enter the MATLAB variable name or a MATLAB expression that represents the amplitude $|G|$.

- **Phase (deg)** — Enter the MATLAB variable name or a MATLAB expression that represents the phase $\varphi = \arg G$.

- **Frequency** — Enter the MATLAB variable name of a vector or a MATLAB expression that represents the frequencies. The expression must evaluate to a column vector.

- **Data name** — Enter the name of the data set, which appears in the System Identification app window after the import operation is completed.

- **Frequency unit** — Enter `Hz` for Hertz or keep the `rad/s` default value.

- **Sample time** — Enter the actual sample time in the experiment. For continuous-time data, enter `0`. For more information about this setting, see "Specifying the Data Sample Time" on page 2-21.

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

Channel Names

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

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

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

Physical Units of Variables

- **Input** — Enter the input units.

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

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

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

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

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

Import Data Objects into the App

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

Before you can import a data object into the System Identification app, you must create the data object in the MATLAB workspace, as described in "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-34 or "Representing Frequency-Response Data Using idfrd Objects" on page 2-61.

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

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

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

To import a data object into the app:

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

`systemIdentification`

2   In the System Identification app window, select **Import data > Data object**.
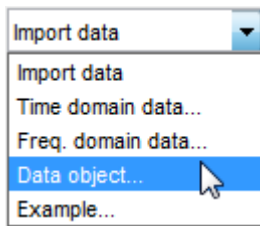
This action opens the Import Data dialog box. **IDDATA or IDFRD/FRD** is already selected in the **Data Format for Signals** list.

3   Specify the following options:

- **Object** — Enter the name of the MATLAB variable that represents the data object in the MATLAB workspace. Press **Enter**.

- **Data name** — Enter the name of the data set, which appears in the System Identification app window after the import operation is completed.

- (Only for time-domain `iddata` object) **Starting time** — Enter the starting value of the time axis for time plots.

- (Only for frequency domain `iddata` or `idfrd` object) **Frequency unit** — Enter the frequency unit for response plots.

- **Sample time** — Enter the actual sample time in the experiment. For more information about this setting, see "Specifying the Data Sample Time" on page 2-21.

Tip The System Identification Toolbox product uses the sample time during model estimation and to set the horizontal axis on time plots. If you transform a time-domain signal to a

---

frequency-domain signal, the Fourier transforms are computed as discrete Fourier transforms (DFTs) using this sample time.

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

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

- **InterSample** — This options specifies the behavior of the input signals between samples during data acquisition. It is used when transforming models from discrete-time to continuous-time and when resampling the data.

  - `zoh` (zero-order hold) indicates that the input was piecewise-constant during data acquisition.
  - `foh` (first-order hold) indicates that the input was piecewise-linear during data acquisition.
  - `bl` (bandwidth-limited behavior) specifies that the continuous-time input signal has zero power above the Nyquist frequency (equal to the inverse of the sample time).

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

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

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

Channel Names

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

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

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

Physical Units of Variables

- **Input** — Enter the input units.

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

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

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

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

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

Specifying the Data Sample Time

When you import data into the app, you must specify the data sample time.

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

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

The sample time is used during model estimation. For time-domain data, the sample time is used together with the start time to calculate the sampling time instants. When you transform time-domain signals to frequency-domain signals (see the `fft` reference page), the Fourier transforms are computed as discrete Fourier transforms (DFTs) for this sample time. In addition, the sampling instants are used to set the horizontal axis on time plots.
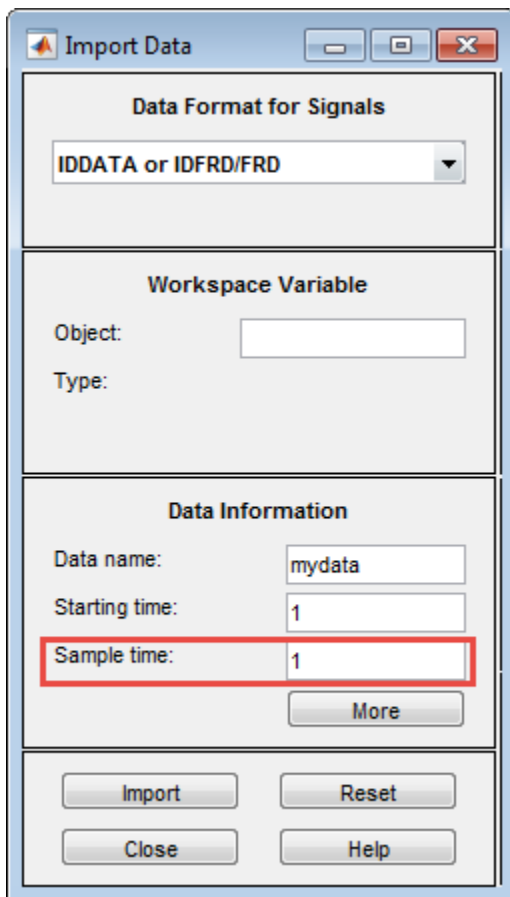
Sample Time in the Import Data dialog box

Specify Estimation and Validation Data in the App

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

In the System Identification app, **Working Data** refers to estimation data. Similarly, **Validation Data** refers to the data set you use to validate a model. For example, when you plot the model output, the input to the model is the input signal from the validation data set. This plot compares model output to the measured output in the validation data set. Selecting **Model resids** performs residual analysis using the validation data.

To specify **Working Data**, drag and drop the corresponding data icon into the **Working Data** rectangle, as shown in the following figure. Similarly, to specify **Validation Data**, drag and drop the corresponding data icon into the **Validation Data** rectangle. Alternatively, right-click the icon to open the Data/model Info dialog box. Select the **Use as Working Data** or **Use as Validation Data** and click **Apply** to specify estimation and validation data, respectively.

See Also

More About

- "Select Subsets of Data" on page 2-74

Preprocess Data Using Quick Start

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

- Subtract the mean value from each channel.

Note For information about when to subtract mean values from the data, see "Handling Offsets and Trends in Data" on page 2-81.

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

Create Data Sets from a Subset of Signal Channels

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

To create a new data set from selected channels:

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

2 Select **Preprocess > Select channels** to open the Select Channels dialog box.
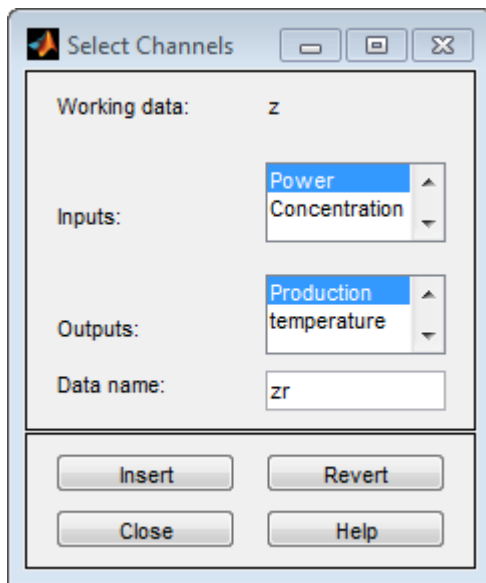
The **Inputs** list displays the input channels and the **Outputs** list displays the output channels in the selected data set.

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

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

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

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

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

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

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

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

7   Click **Close**.

Create Multiexperiment Data Sets in the App

Why Create Multiexperiment Data?

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

Experiments can mean data that was collected during different sessions, or portions of the data collected during a single session. In the latter situation, you can create multiexperiment data by splitting a single data set into multiple segments that exclude corrupt data, and then merge the good data segments.

Limitations on Data Sets

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

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

Merging Data Sets

You can merge data sets using the System Identification app.

For example, suppose that you want to combine the data sets `tdata`, `tdata2`, `tdata3`, `tdata4` shown in the following figure.
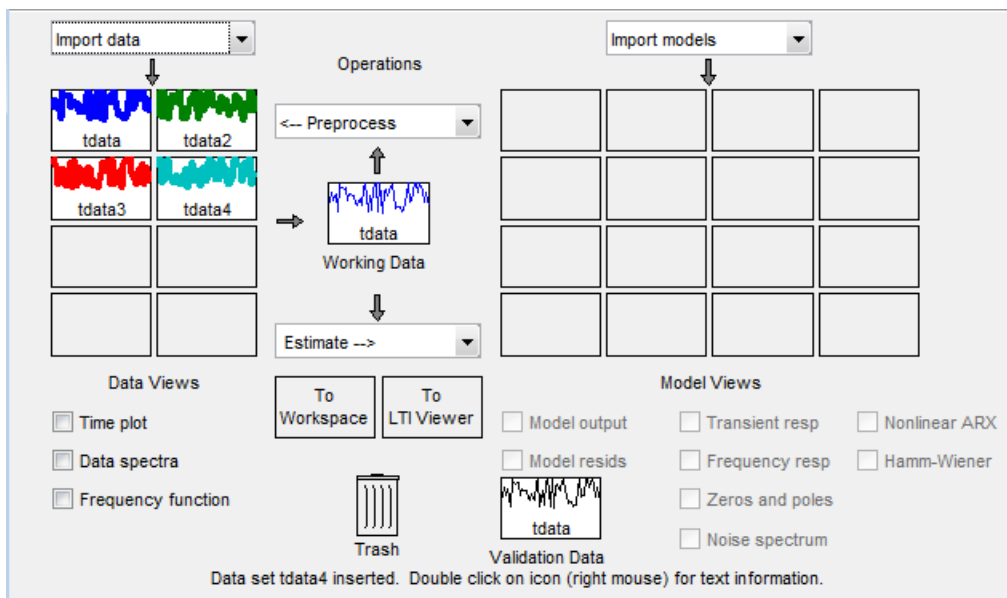
App Contains Four Data Sets to Merge

To merge data sets in the app:

1 In the **Operations** area, select **<--Preprocess > Merge experiments** from the drop-down menu to open the Merge Experiments dialog box.
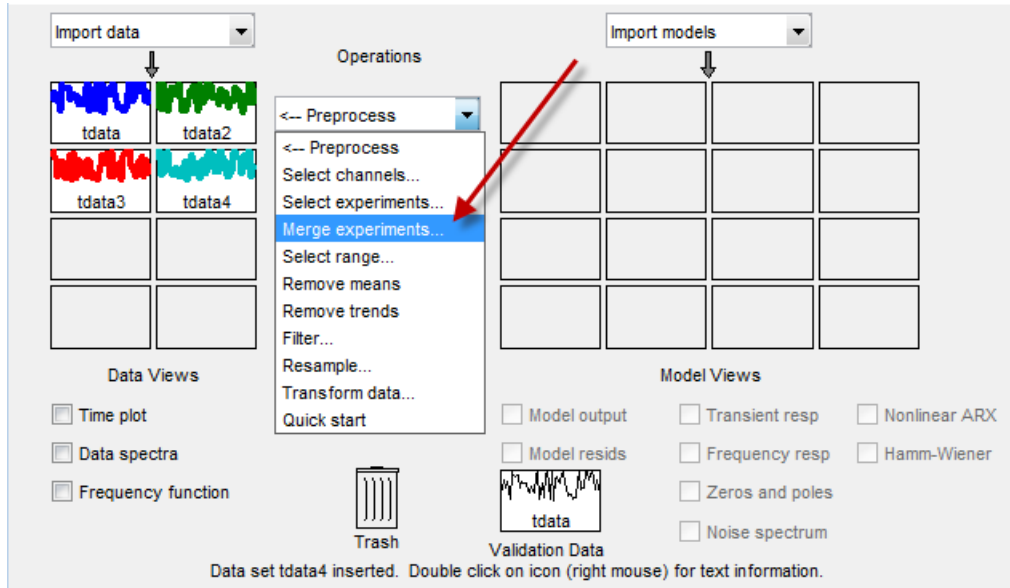
2 In the System Identification app window, drag a data set icon to the Merge Experiments dialog box, to the **drop them here to be merged** rectangle.

The name of the data set is added to the **List of sets**. Repeat for each data set you want to merge.
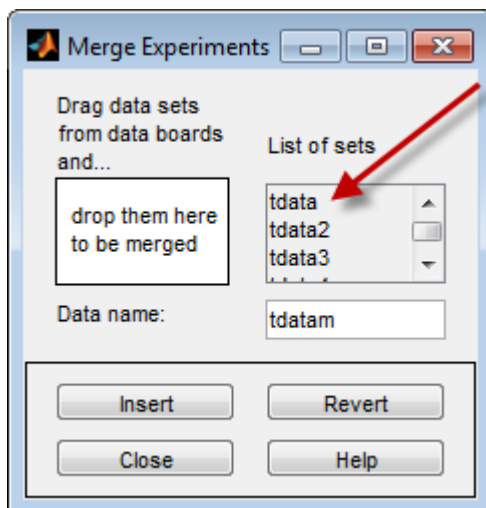
tdata and tdata2 to Be Merged

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

---

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

4   Click **Insert** to add the new data set to the Data Board in the System Identification app window.
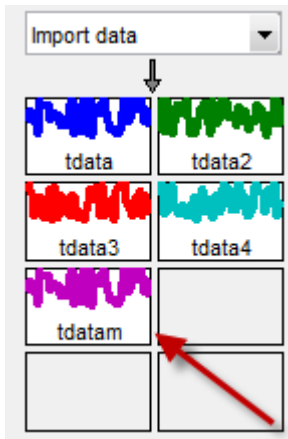
Data Board Now Contains tdatam with Merged Experiments

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

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

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

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

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

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

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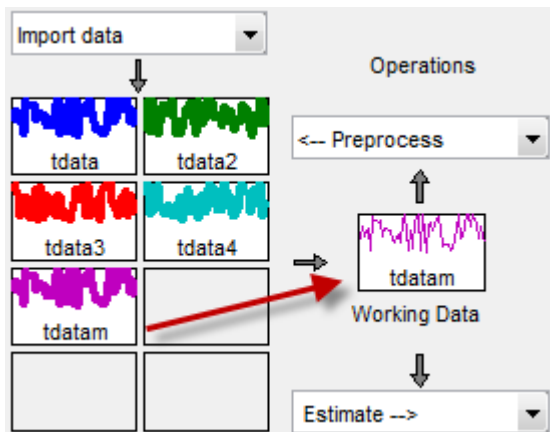
tdatam Is Set to Working Data

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

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

- Select one data set by clicking its name.
- Select adjacent data sets by pressing the **Shift** key while clicking the first and last names.
- Select nonadjacent data sets by pressing the **Ctrl** key while clicking each name.
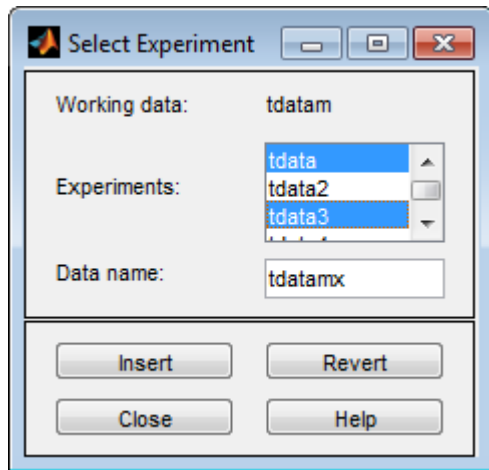
4 In the **Data name** field, type the name of the new data set. This name must be unique in the Data Board.

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

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

See Also

More About

- "Select Subsets of Data" on page 2-74
- "Create Multiexperiment Data at the Command Line" on page 2-42

Managing Data in the App

Viewing Data Properties

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

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

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

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

Renaming Data and Changing Display Color

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

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

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

You can also specify a new display color using three RGB values in the **Color** field. Each value is between 0 to 1 and indicates the relative presence of red, green, and blue, respectively. For more information about specifying default data color, see "Customizing the System Identification App" on page 21-11.

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

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

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

Information About the Data

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

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

Distinguishing Data Types

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

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

Colors Representing Type of Data

Organizing Data Icons

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

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

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

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

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

Type comments in the **Notes** field to describe the data sets. When you save a session, as described in "Saving, Merging, and Closing Sessions" on page 21-4, all additional windows and notes are also saved.

Deleting Data Sets

To delete data sets in the System Identification app, drag and drop the corresponding icon into **Trash**. You can also use the **Delete** key on your keyboard to move items to the **Trash**. Moving items to **Trash** does not permanently delete these items.

Note You cannot delete a data set that is currently designated as **Working Data** or **Validation Data**. You must first specify a different data set in the System Identification app to be **Working Data** or **Validation Data**, as described in "Specify Estimation and Validation Data in the App" on page 2-22.

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

Note You must restore data to the Data Board; you cannot drag data icons to the Model Board.
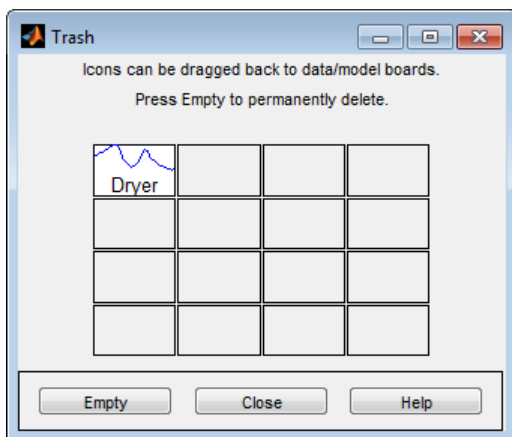
To permanently delete all items in **Trash**, select **Options > Empty trash**.

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

Exporting Data to the MATLAB Workspace

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

To export a data set to the MATLAB workspace, do one of the following:

- Drag and drop the corresponding icon to the **To Workspace** rectangle.
- Right-click the icon to open the Data/model Info dialog box. Click **Export**.

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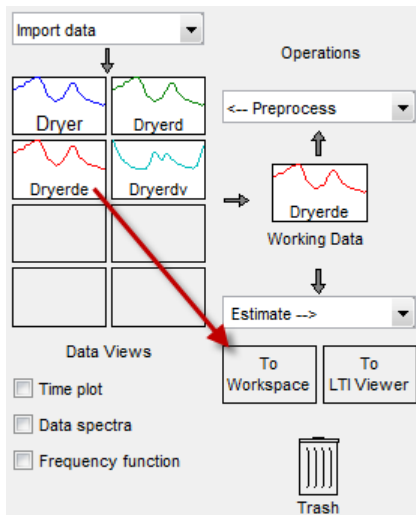
Exporting Data to the MATLAB Workspace

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

Representing Time- and Frequency-Domain Data Using iddata Objects

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 2-8.

Constructing an iddata Object for Time-Domain Data

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

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

You can also specify additional properties, as follows:

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

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

In this example, `Ts` is the sample time, or the time interval, between successive data samples. For uniformly sampled data, `Ts` is a scalar value equal to the sample time of your experiment. The default time unit is seconds, but you can set it to a new value using the `TimeUnit` property. For more information about `iddata` time properties, see "Modifying Time and Frequency Vectors" on page 2-57.

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

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

Where `TimeVector` represents a vector of time values.

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

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

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

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

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

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

```
data =

Time domain data set with 1000 samples.
Sample time: 0.08 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

The default channel name `'y1'` is assigned to the first and only output channel. When y2 contains several channels, the channels are assigned default names `'y1'`,`'y2'`,`'y2'`,...,`'yn'`. Similarly, the default channel name `'u1'` is assigned to the first and only input channel. For more information about naming channels, see "Naming, Adding, and Removing Data Channels" on page 2-59.

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 2-57.

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

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

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

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

iddata Properties

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

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

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

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

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

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

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

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

or equivalently:

```
data.ts = 0.05
```

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

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

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

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

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

```
data =

Frequency domain data set with responses at 501 frequencies.
Frequency range: 0 to 39.27 rad/seconds
Sample time: 0.08 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

```
% Get property values of data.
  get(data)
```

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

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

Subreferencing Input and Output Data

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

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

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

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

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

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

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

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

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

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

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

or

```
data.InputData
```

or

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

Similarly, you can retrieve the output data using

```
data.OutputData
```

or

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

Subreferencing Data Channels

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

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

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

To specify several channel names, you must use a cell array of character vectors of names.

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

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

or equivalently

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

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

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

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

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

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

Subreferencing Experiments

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

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

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

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

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

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

or

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

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

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

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

or

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

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

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

Increasing Number of Channels or Data Points of iddata Objects

iddata Properties Storing Input and Output Data

The `InputData iddata` property stores column-wise input data, and the `OutputData` property stores column-wise output data. For more information about accessing `iddata` properties, see "iddata Properties" on page 2-35.

Horizontal Concatenation

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

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

This syntax is equivalent to the following longer syntax:

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

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

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

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

Vertical Concatenation

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

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

This syntax is equivalent to the following longer syntax:

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

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

See Also
`iddata`

More About
- "Representing Data in MATLAB Workspace" on page 2-8
- "Managing iddata Objects" on page 2-57

Create 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

This example shows how to create a multiexperiment `iddata` object by merging `iddata` objects, where each contains data from a single experiment or is a multiexperiment data set.

Load iddata objects `z1` and `z3`.

```
load iddata1
load iddata3
```

Merge experiments `z1` and `z3` into the iddata object `z`.

```
z = merge(z1,z3)

z =
Time domain data set containing 2 experiments.

Experiment   Samples      Sample Time
```

```
   Exp1          300              0.1
   Exp2          300              1

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

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

Adding Experiments to an Existing iddata Object

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

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

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

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

For more information about subreferencing experiments in a multiexperiment data set, see "Subreferencing Experiments" on page 2-39.

See Also

More About

- "Select Subsets of Data" on page 2-74
- "Dealing with Multi-Experiment Data and Merging Models" on page 2-44
- "Create Multiexperiment Data Sets in the App" on page 2-26

Dealing with Multi-Experiment Data and Merging Models

This example shows how to deal with multiple experiments and merging models when working with System Identification Toolbox™ for estimating and refining models.

Introduction

The analysis and estimation functions in System Identification Toolbox let you work with multiple batches of data. Essentially, if you have performed multiple experiments and recorded several input-output datasets, you can group them up into a single IDDATA object and use them with any estimation routine.

In some cases, you may want to "split up" your (single) measurement dataset to remove portions where the data quality is not good. For example, portion of data may be unusable due to external disturbance or a sensor failure. In those cases, each good portion of data may be separated out and then combined into a single multi-experiment IDDATA object.

For example, let us look at the dataset iddemo8.mat:

```
load iddemo8
```

The name of the data object is `dat`, and let us view it.

```
dat
```

```
dat =

Time domain data set with 1000 samples.
Sample time: 1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1

```

```
plot(dat)
```
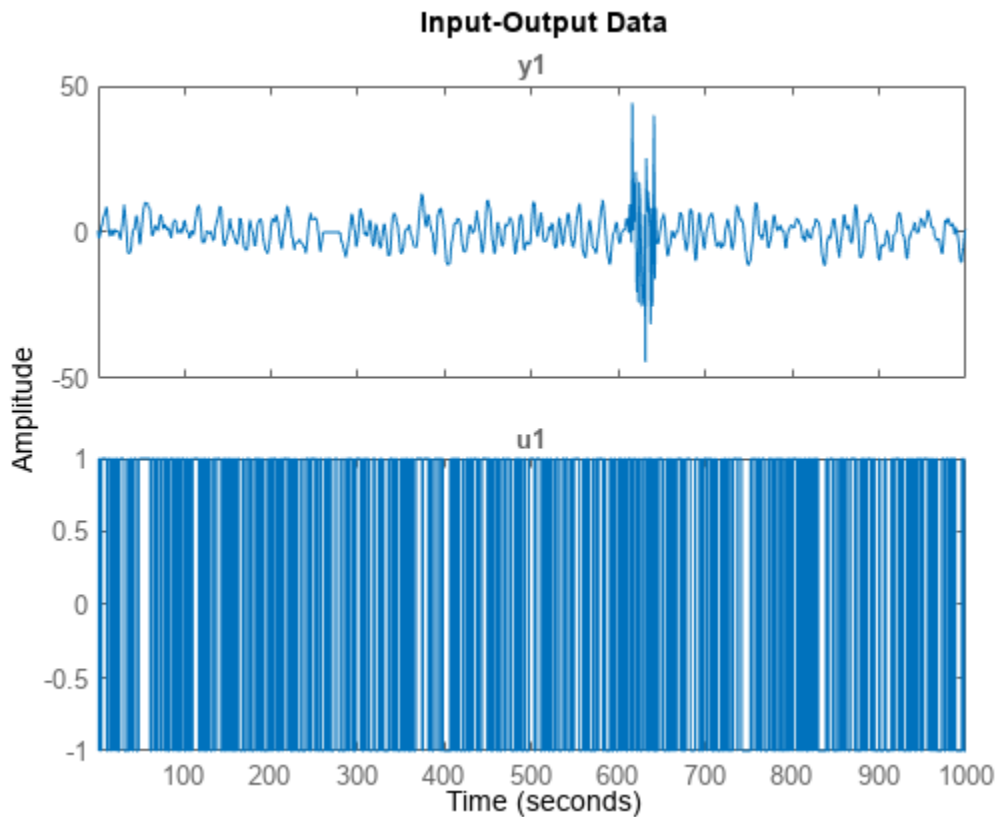
We see that there are some problems with the output around sample 250-280 and around samples 600 to 650. These might have been sensor failures.

Therefore split the data into three separate experiments and put then into a multi-experiment data object:

```
d1 = dat(1:250);
d2 = dat(281:600);
d3 = dat(651:1000);
d = merge(d1,d2,d3) % merge lets you create multi-exp IDDATA object

d =
Time domain data set containing 3 experiments.

Experiment    Samples       Sample Time
   Exp1         250             1
   Exp2         320             1
   Exp3         350             1

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

The different experiments can be given other names, for example:

```
d.exp = {'Period 1';'Day 2';'Phase 3'}

d =
Time domain data set containing 3 experiments.

Experiment       Samples        Sample Time
   Period 1         250             1
   Day 2            320             1
   Phase 3          350             1

Outputs         Unit (if specified)
   y1

Inputs          Unit (if specified)
   u1
```

To examine it, use plot, as in `plot(d)`.

Performing Estimation Using Multi-Experiment Data

As mentioned before, all model estimation routines accept multi-experiment data and take into account that they are recorded at different periods. Let us use the two first experiments for estimation and the third one for validation:

```
de = getexp(d,[1,2]);     % subselection is done using  the command GETEXP
dv = getexp(d,'Phase 3'); % using numbers or names.
m1 = arx(de,[2 2 1]);
m2 = n4sid(de,2);
m3 = armax(de,[2 2 2 1]);
compare(dv,m1,m2,m3)
```
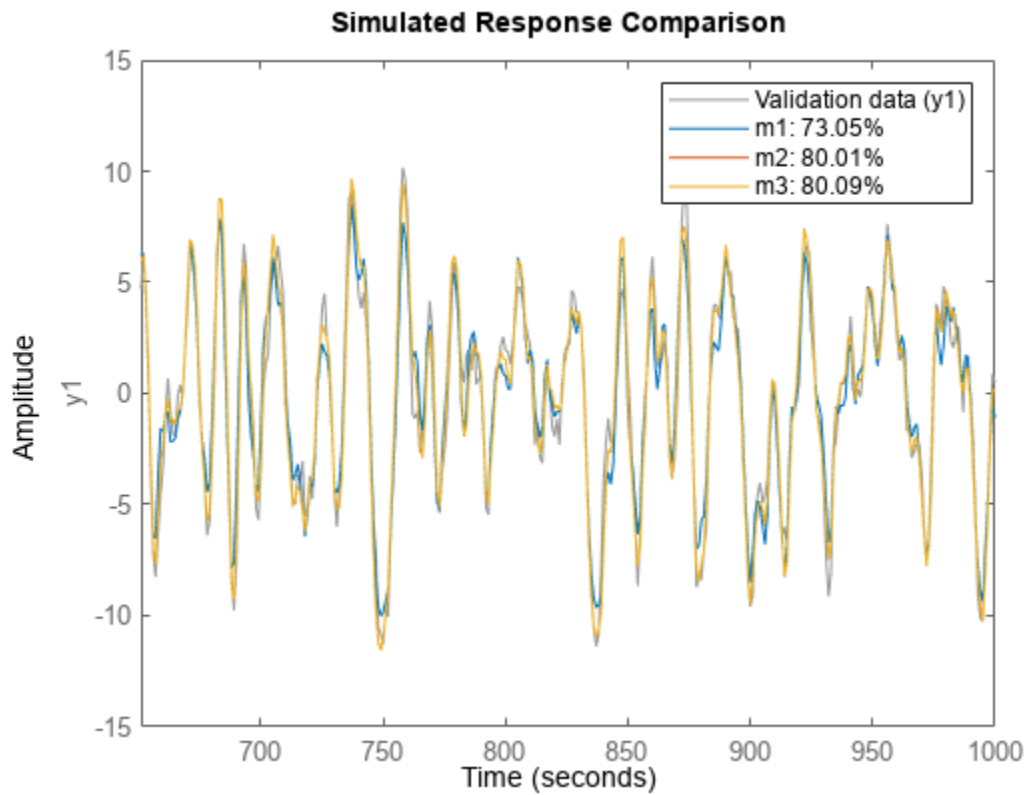
The `compare` command also accepts multiple experiments. Use the right click menu to pick the experiment to use, one at a time.

```
compare(d,m1,m2,m3)
```

Simulated Response Comparison
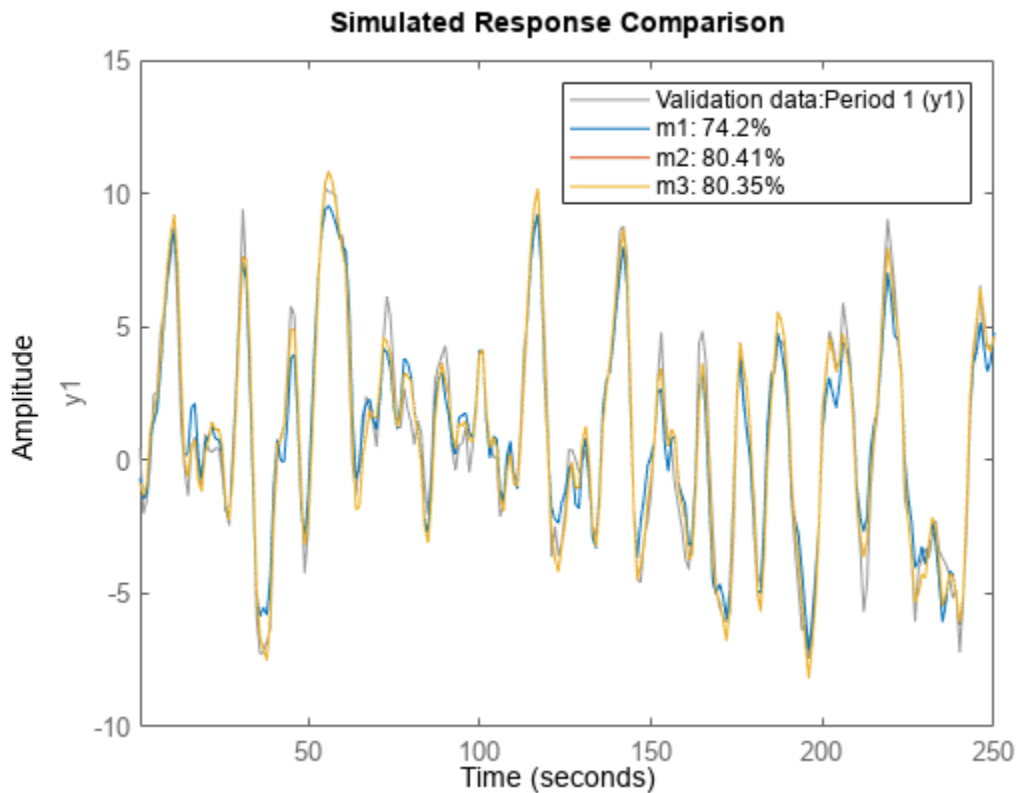
Also, `spa`, `etfe`, `resid`, `predict`, `sim` operate in the same way for multi-experiment data, as they do for single experiment data.

Merging Models After Estimation

There is another way to deal with separate data sets: a model can be computed for each set, and then the models can be merged:

```
m4 = armax(getexp(de,1),[2 2 2 1]);
m5 = armax(getexp(de,2),[2 2 2 1]);
m6 = merge(m4,m5); % m4 and m5 are merged into m6
```

This is conceptually the same as computing m from the merged set `de`, but it is not numerically the same. Working on `de` assumes that the signal-to-noise ratios are (about) the same in the different experiments, while merging separate models makes independent estimates of the noise levels. If the conditions are about the same for the different experiments, it is more efficient to estimate directly on the multi-experiment data.

We can check the models `m3` and `m6` that are both ARMAX models obtained on the same data in two different ways:

```
[m3.a;m6.a]
```

ans = *2×3*

```
    1.0000   -1.5034    0.7008
    1.0000   -1.5022    0.7000
```

```
[m3.b;m6.b]
```

ans = *2×3*

```
     0    1.0023    0.5029
     0    1.0035    0.5028
```

```
[m3.c;m6.c]
```

ans = *2×3*

```
  1.0000   -0.9744    0.1578
  1.0000   -0.9751    0.1584
```

```
compare(dv,m3,m6)
```
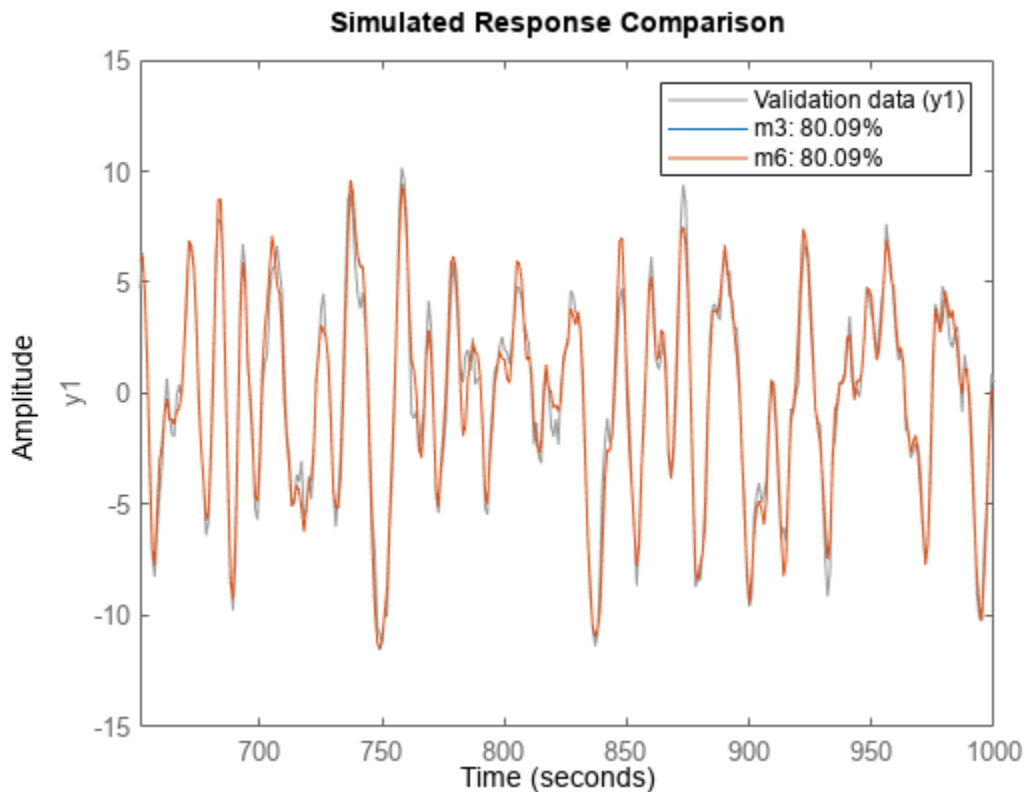

Case Study: Concatenating Vs. Merging Independent Datasets

We now turn to another situation. Let us consider two data sets generated by the system m0. The system is given by:

```
m0
```

```
m0 =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + B u(t) + K e(t)
        y(t) = C x(t) + D u(t) + e(t)
```

```
A =
              x1        x2        x3
    x1     0.5296    -0.476    0.1238
    x2     -0.476   -0.09743    0.1354
    x3     0.1238    0.1354   -0.8233

B =
              u1        u2
    x1     -1.146   -0.03763
    x2      1.191     0.3273
    x3          0          0

C =
              x1        x2        x3
    y1    -0.1867   -0.5883   -0.1364
    y2     0.7258         0    0.1139

D =
              u1        u2
    y1     1.067         0
    y2         0         0

K =
        y1   y2
    x1   0    0
    x2   0    0
    x3   0    0

Sample time: 1 seconds

Parameterization:
   STRUCTURED form (some fixed coefficients in  A, B, C).
   Feedthrough: on some input channels
   Disturbance component: none
   Number of free coefficients: 23
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

The data sets that have been collected are z1 and z2, obtained from m0 with different inputs, noise and initial conditions. These datasets are obtained from iddemo8.mat that was loaded earlier.

```
pause off
```

First data set:

```
plot(z1) %generates a separate plot for each I/O pair if pause is on; showing only the last one l
```

Input-Output Data
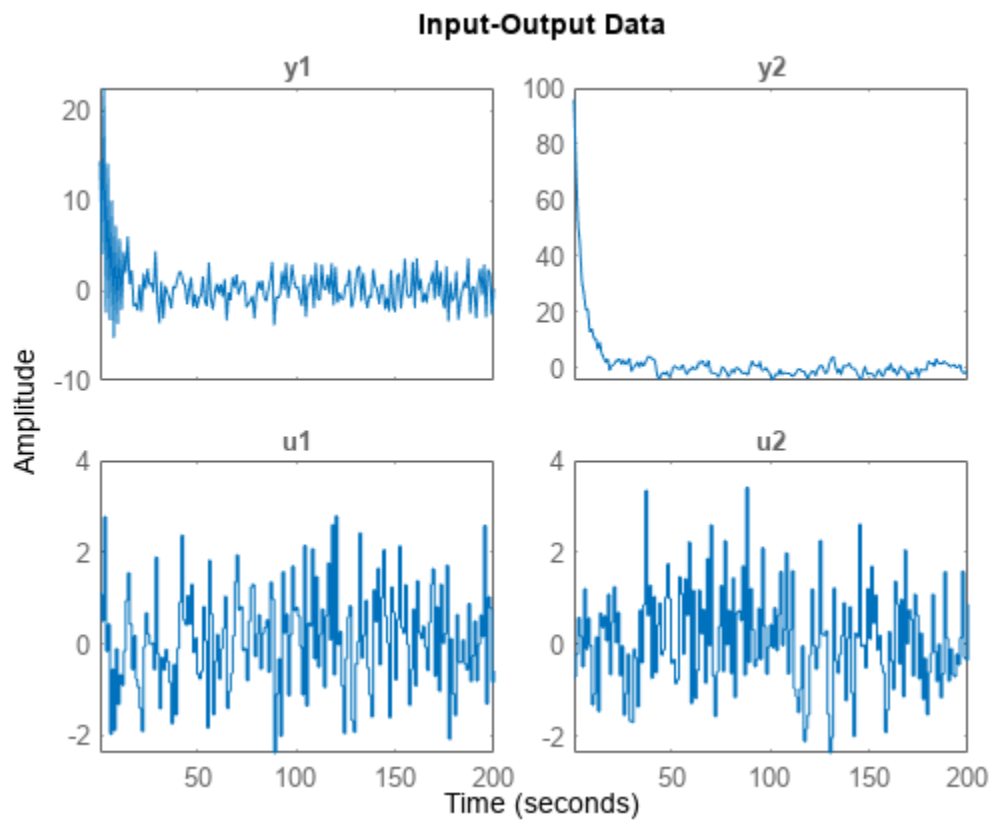
The second set:

```
plot(z2) %generates a separate plot for each I/O pair if pause is on; showing only the last one
```

Input-Output Data
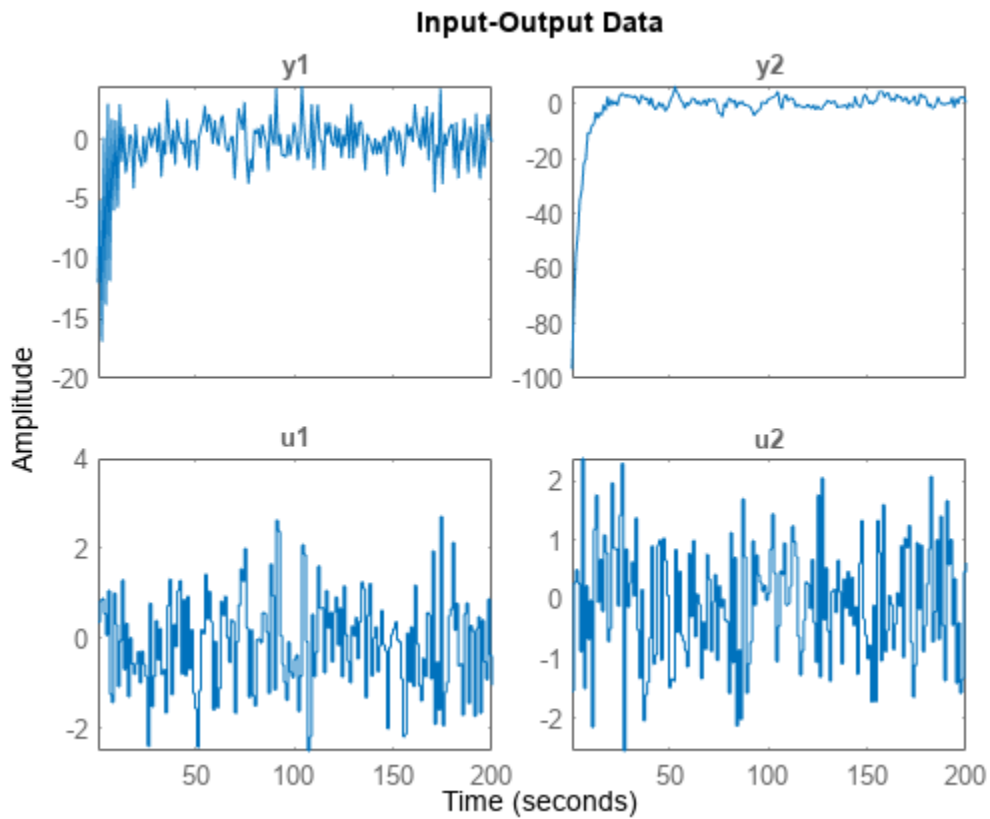
If we just concatenate the data we obtained:

```
zzl = [z1;z2]

zzl =

Time domain data set with 400 samples.
Sample time: 1 seconds

Outputs      Unit (if specified)
   y1
   y2

Inputs       Unit (if specified)
   u1
   u2


plot(zzl)
```
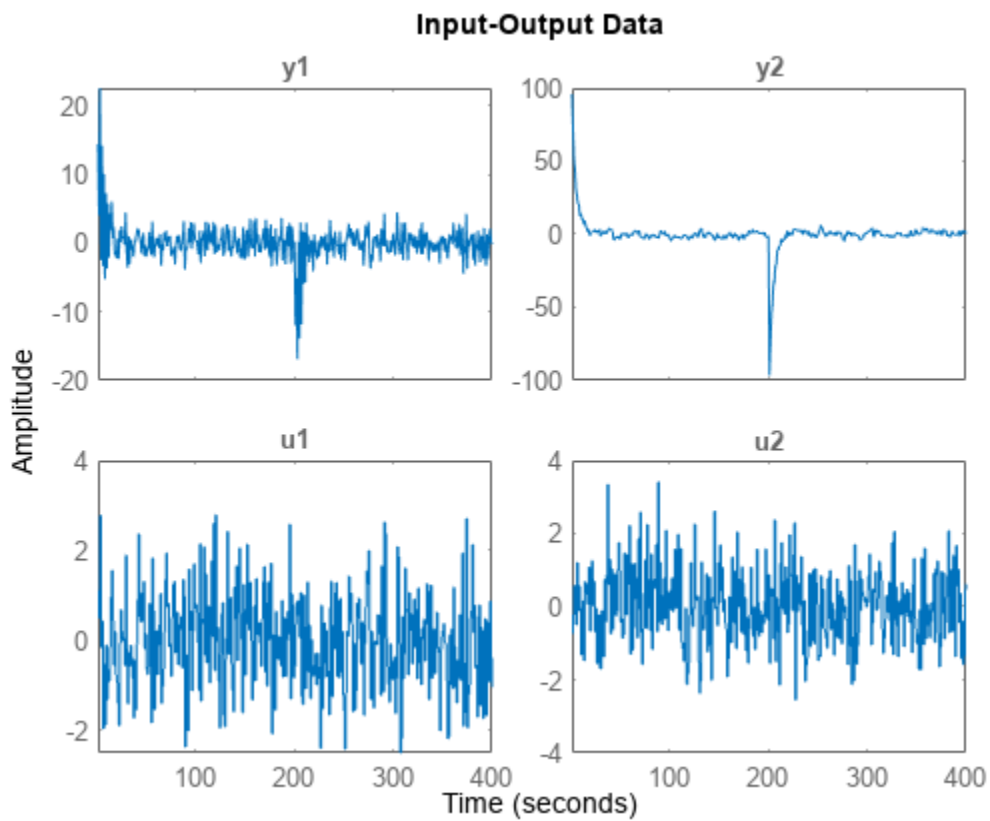
pause on

A discrete-time state-space model can be obtained by using `ssest`:

```
ml = ssest(zzl,3,'Ts',1, 'Feedthrough', [true, false]);
```

Compare the bode response for models m0 and ml:

```
clf
bode(m0,ml)
legend('show')
```
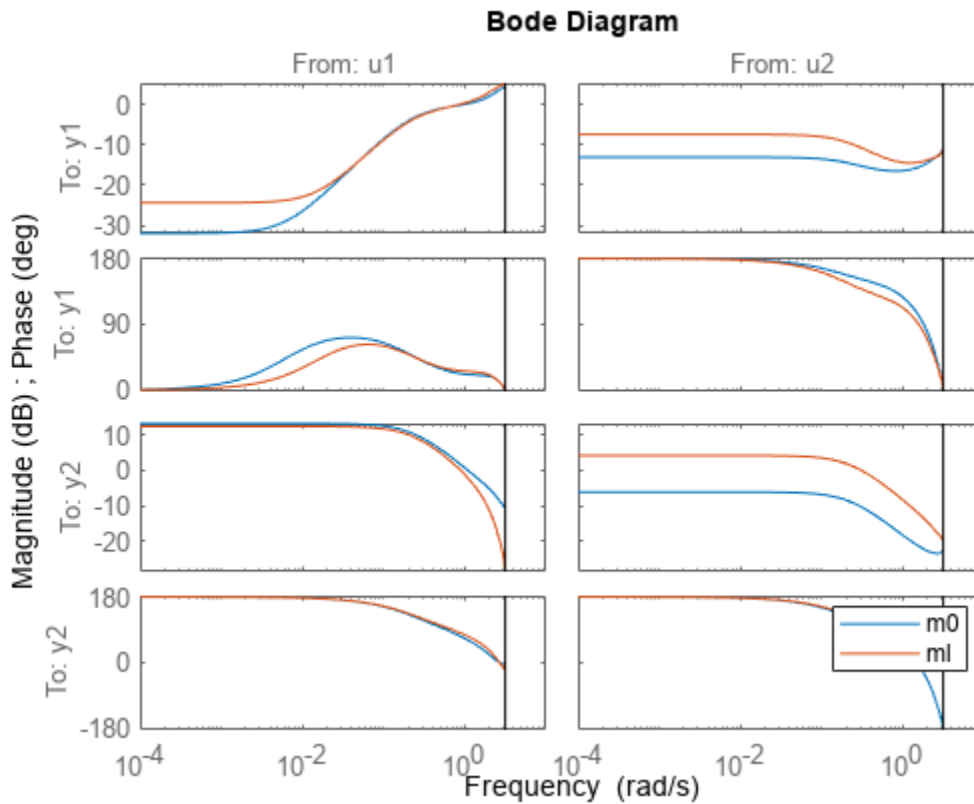
This is not a very good model, as observed from the four Bode plots above.

Now, instead treat the two data sets as different experiments:

```
zzm = merge(z1,z2)

zzm =
Time domain data set containing 2 experiments.

Experiment   Samples      Sample Time
   Exp1         200            1
   Exp2         200            1

Outputs      Unit (if specified)
   y1
   y2

Inputs       Unit (if specified)
   u1
   u2
```

```
% The model for this data can be estimated as before (watching progress this time)
mm = ssest(zzm,3,'Ts',1,'Feedthrough',[true, false], ssestOptions('Display', 'on'));
```

Let us compare the Bode plots of the true system (blue)

the model from concatenated data (green) and the model from the

merged data set (red):

```
clf
bode(m0,'b',ml,'g',mm,'r')
legend('show')
```
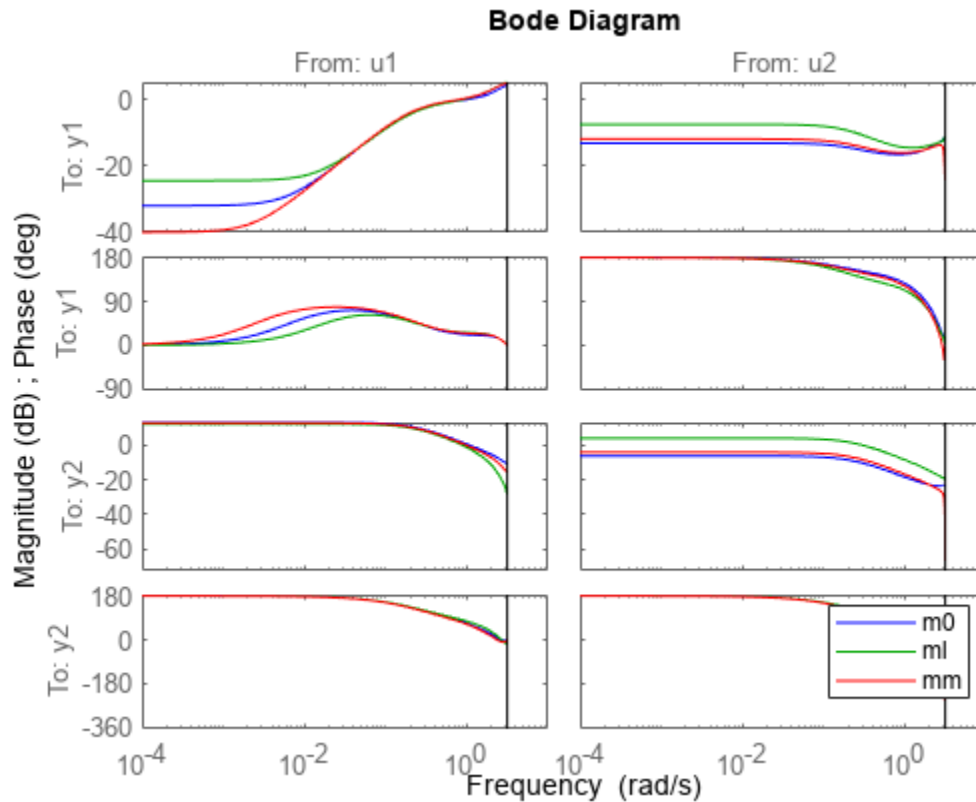

The merged data give a better model, as observed from the plot above.

Conclusions

In this example we analyzed how to use multiple data sets together for estimation of one model. This technique is useful when you have multiple datasets from independent experiment runs or when you segment data into multiple sets to remove bad segments. Multiple experiments can be packaged into a single IDDATA object, which is then usable for all estimation and analysis requirements. This technique works for both time and frequency domain iddata.

It is also possible to merge models after estimation. This technique can be used to "average out" independently estimated models. If the noise characteristics on multiple datasets are different,

merging models after estimation works better than merging the datasets themselves before estimation.

See Also

More About

- "Select Subsets of Data" on page 2-74
- "Create Multiexperiment Data Sets in the App" on page 2-26
- "Create Multiexperiment Data at the Command Line" on page 2-42

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 2-34.

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

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

iddata Time-Vector Properties

Property	Description	Syntax Example

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

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

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

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

iddata Frequency-Vector Properties

Property	Description	Syntax Example

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

`FrequencyUnit`	Unit of Frequency. Specify as one of the following: be one of the following: `'rad/TimeUnit'`, `'cycles/TimeUnit'`, `'rad/s'`, `'Hz'`, `'kHz'`, `'MHz'`, `'GHz'`, and, `'rpm'`. Default: `'rad/TimeUnit'` For multi-experiment data with `Ne` experiments, `Units` is a 1-by-`Ne` cell array, and each cell contains the frequency unit for each experiment.	Set the frequency unit to Hz: `data.FrequencyUnit = 'Hz'` Note that changing the frequency unit does not scale the frequency vector. For a proper translation of units, use `chgFreqUnit`.

Naming, Adding, and Removing Data Channels

What Are Input and Output Channels?

A multivariate system might contain several input variables or several output variables, or both. When an input or output signal includes several measured variables, these variables are called *channels*.

Naming Channels

The `iddata` properties `InputName` and `OutputName` store the channel names for the input and output signals. When you plot the data, you use channel names to select the variable displayed on the plot. If you have multivariate data, it is helpful to assign a name to each channel that describes the measured variable. For more information about selecting channels on a plot, see "Selecting Measured and Noise Channels in Plots" on page 21-10.

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 2-40.

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 2-37.

Concatenating iddata Objects

See "Increasing Number of Channels or Data Points of iddata Objects" on page 2-40.

Representing Frequency-Response Data Using idfrd Objects

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 2-11.

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

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

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

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

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

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

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

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

idfrd Properties

To view the properties of the `idfrd` object, you can use the `get` command. The following example shows how to create an `idfrd` object that contains 100 frequency-response values with a sample time of 0.1 s and get its properties:

```
f = logspace(-1,1,100);
[mag, phase] = bode(idtf([1 .2],[1 2 1 1]),f);
response = mag.*exp(1j*phase*pi/180);
fr_data = idfrd(response,f,0.1);
get(fr_data)

    FrequencyUnit: 'rad/TimeUnit'
           Report: [1x1 idresults.frdest]
     SpectrumData: []
   CovarianceData: []
  NoiseCovariance: []
      InterSample: {'zoh'}
     ResponseData: [1x1x100 double]
           IODelay: 0
```

```
       InputDelay: 0
      OutputDelay: 0
        InputName: {''}
        InputUnit: {''}
       InputGroup: [1x1 struct]
       OutputName: {''}
       OutputUnit: {''}
      OutputGroup: [1x1 struct]
            Notes: [0x1 string]
         UserData: []
             Name: ''
               Ts: 0.1000
         TimeUnit: 'seconds'
     SamplingGrid: [1x1 struct]
        Frequency: [100x1 double]
```

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

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

```
fr_data.Name = 'DC_Converter';
```

Select I/O Channels and Data in idfrd Objects

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

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

Reference specific channels by name or by channel index.

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

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

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

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

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

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

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

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

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

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

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

Adding Input or Output Channels in idfrd Objects

About Concatenating idfrd Objects

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

Horizontal Concatenation of idfrd Objects

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

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

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

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

Vertical Concatenation of idfrd Objects

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

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

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

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

Concatenating Noise Spectrum Data of idfrd Objects

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

In case of horizontal concatenation, there is no meaningful way to combine the `SpectrumData` of individual `idfrd` objects and the resulting `SpectrumData` property is empty. An empty property results because each `idfrd` object has its own set of noise channels, where the number of noise channels equals the number of outputs. When the resulting `idfrd` object contains the same output channels as each of the individual `idfrd` objects, it cannot accommodate the noise data from all the `idfrd` objects.

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

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

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

Managing idfrd Objects

Subreferencing idfrd Objects

See "Select I/O Channels and Data in idfrd Objects" on page 2-62.

Concatenating idfrd Objects

See "Adding Input or Output Channels in idfrd Objects" on page 2-63.

Operations that Create idfrd Objects

The following operations create `idfrd` objects:

- Constructing `idfrd` objects.
- Estimating nonparametric models using `etfe`, `spa`, and `spafdr`. For more information, see "Frequency-Response Models".
- Converting the Control System Toolbox `frd` object. For more information, see "Using Identified Models for Control Design Applications" on page 19-2.
- Converting any linear dynamic system using the `idfrd` command.

  For example:

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

  G =

  IDFRD model.
  Contains Frequency Response Data for 1 output(s) and 1 input(s), and the spectra for disturban
  Response data and disturbance spectra are available at 128 frequency points, ranging from 0 ra
  ```

```
Sample time: 1 seconds
Status:
Created by direct construction or transformation. Not estimated.
```

Is Your Data Ready for Modeling?

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

The following data plots are available in the toolbox:

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

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

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

See Also

Related Examples

- "How to Analyze Data Using the advice Command" on page 2-73
- "How to Plot Data in the App" on page 2-67
- "How to Plot Data at the Command Line" on page 2-71

More About

- "Ways to Prepare Data for System Identification" on page 2-5

How to Plot Data in the App

How to Plot Data in the App

After importing data into the System Identification app, as described in "Represent Data", you can plot the data.

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

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

When you have several data sets, you can view different input-output channel pair by selecting that pair from the **Channel** menu. For more information about selecting different input and output pairs, see "Selecting Measured and Noise Channels in Plots" on page 21-10.

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

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

***

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

***

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

Manipulating a Time Plot

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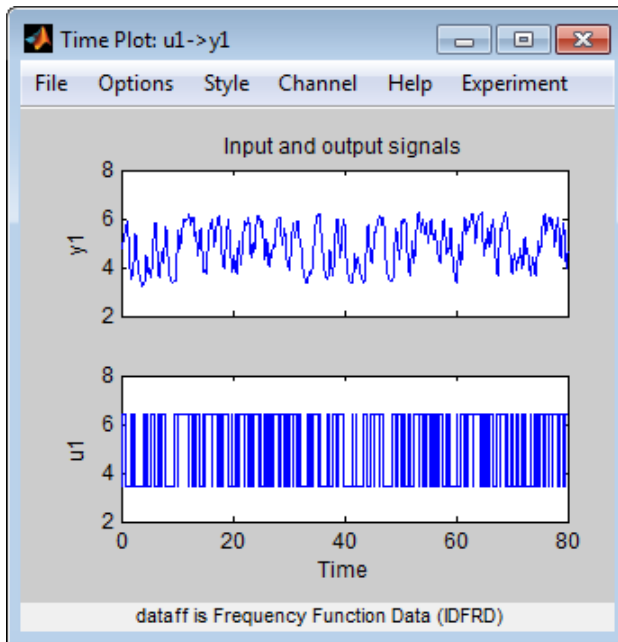
Time Plot of data1

The following table summarizes options that are specific to time plots, which you can select from the plot window menus. For general information about working with System Identification Toolbox plots, see "Working with Plots" on page 21-8.

Time Plot Options

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

Manipulating Data Spectra Plot

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

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

The top axes show the input and the bottom axes show the output. The vertical axis of each plot is labeled with the corresponding channel name.
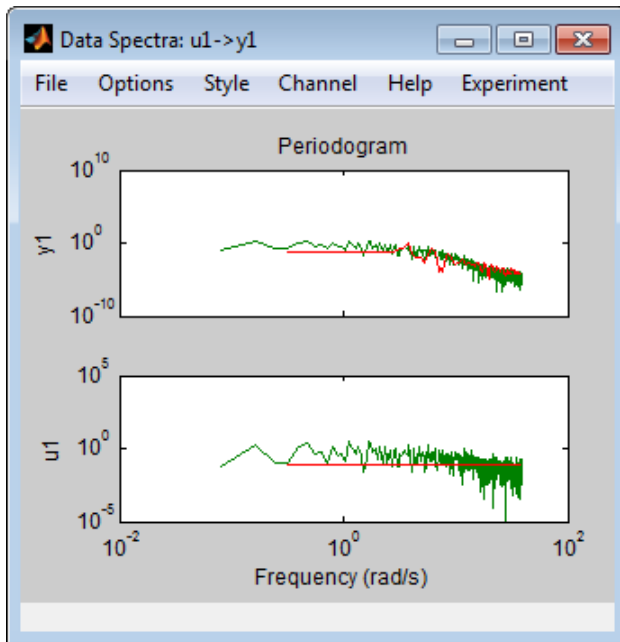
Periodograms of data1 and data3fd

Data Spectra Plot Options

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

Manipulating a Frequency Function Plot

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

The frequency-response plot shows the amplitude and phase plots of the corresponding frequency response. For more information about frequency-response data, see "Frequency-Response Data Representation" on page 2-11.
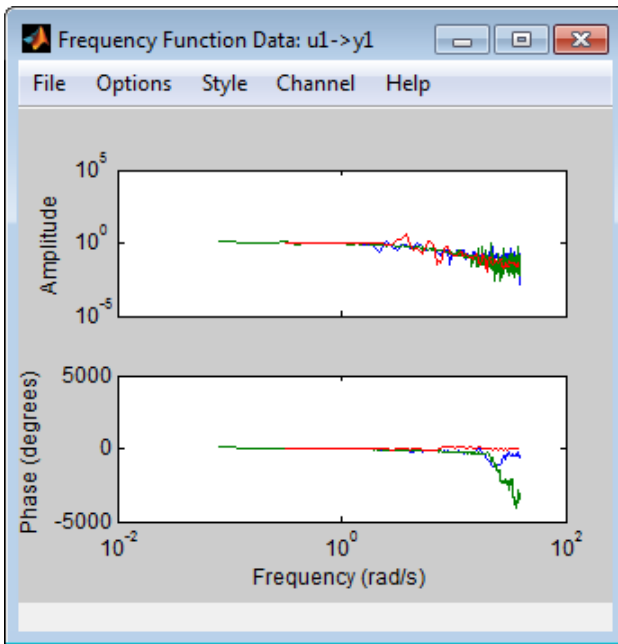
Frequency Functions of data1 and data3fd

Frequency Function Plot Options

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

See Also

Related Examples

- "How to Plot Data at the Command Line" on page 2-71

How to Plot Data at the Command Line

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

Commands for Plotting Data

Command	Description	Example

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

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

All plot commands display the data in the standard MATLAB Figure window, which provides options for formatting, saving, printing, and exporting plots to a variety of file formats.

To plot portions of the data, you can subreference specific samples (see "Select Data Channels, I/O Data and Experiments in iddata Objects" on page 2-37 and "Select I/O Channels and Data in idfrd Objects" on page 2-62. 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 2-38.

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

See Also

Related Examples

*   "How to Plot Data in the App" on page 2-67

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 2-34.

If you are using the System Identification app, you must export your data to the MATLAB workspace before you can use the `advice` command on this data. For more information about exporting data, see "Exporting Models from the App to the MATLAB Workspace" on page 21-8.

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
`delayest` | `advice` | `detrend` | `pexcit` | `feedback`

Related Examples
- "How to Plot Data in the App" on page 2-67
- "How to Plot Data at the Command Line" on page 2-71

Select Subsets of Data

Why Select Subsets of Data?

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

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

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

You can merge several data segments into a single multiexperiment data set and identify an average model. For more information, see "Create Data Sets from a Subset of Signal Channels" on page 2-24 or "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-34.

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 portions of frequency-domain data is equivalent to filtering the data. For more information about filtering, see "Filtering Data" on page 2-92.

Extract Subsets of Data Using the App

Ways to Select Data in the App

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

After you select portions of the data, you can specify to use one data segment for estimating models and use the other data segment for validating models. For more information, see "Specify Estimation and Validation Data in the App" on page 2-22.

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

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

Selecting a Range for Time-Domain Data

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

Note Selecting data does not extract experiments from a data set containing multiple experiments. For more information about multiexperiment data, see "Create Multiexperiment Data Sets in the App" on page 2-26.

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

1 Import time-domain data into the System Identification app, as described in "Create Data Sets from a Subset of Signal Channels" on page 2-24.

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

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

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

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

- Graphically — Draw a rectangle on either the input-signal or the output-signal plot with the mouse to select the desired time interval. Your selection appears on both plots regardless of the plot on which you draw the rectangle. The **Time span** and **Samples** fields are updated to match the selected region.

- By specifying the **Time span** — Edit the beginning and the end times in seconds. The **Samples** field is updated to match the selected region. For example:

  ```
  28.5 56.8
  ```

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

  ```
  342 654
  ```

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

---

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

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

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

Selecting a Range of Frequency-Domain Data

Selecting a range of values in frequency domain is equivalent to filtering the data. For more information about data filtering, see "Filtering Frequency-Domain or Frequency-Response Data in the App" on page 2-94.

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 2-37.

For more information about subreferencing frequency-response data, see "Select I/O Channels and Data in idfrd Objects" on page 2-62.

See Also

More About

- "Create Multiexperiment Data Sets in the App" on page 2-26
- "Create Multiexperiment Data at the Command Line" on page 2-42

Handling Missing Data and Outliers

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 2-34.

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

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

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

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

Handling Outliers

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

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

- Before estimating a model, plot the data on a time plot and identify values that appear out of range.
- After estimating a model, plot the residuals and identify unusually large values. For more information about plotting residuals, see topics on the "Residual Analysis" page. Evaluate the original data that is responsible for large residuals. For example, for the model Model and validation data Data, you can use the following commands to plot the residuals:

```
% Compute the residuals
  E = resid(Data,Model)
% Plot the residuals
  plot(E)
```

Next, try these techniques for removing or minimizing the effects of outliers:

- Extract the informative data portions into segments and merge them into one multiexperiment data set (see "Extract and Model Specific Data Segments" on page 2-79). For more information about selecting and extracting data segments, see "Select Subsets of Data" on page 2-74.

> **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 NaNs 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 2-77. 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 2-92.

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

See Also

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

Extract and Model Specific Data Segments

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

Load and plot the data.

```
load iddemo8;
plot(dat);
```
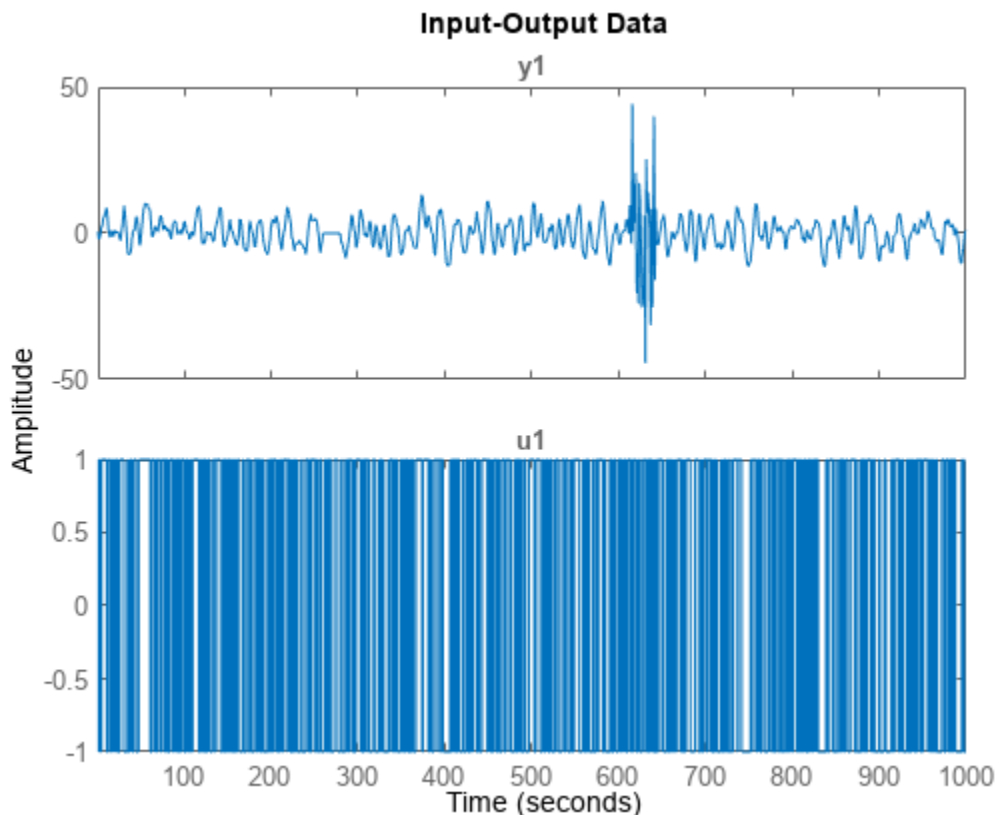

The data has poor or no measurements from samples 251 to 280 and 601 to 650. You cannot simply concatenate the good data segments because the transients at the connection points compromise the model. Instead, you must create a multiexperiment `iddata` object, where each experiment corresponds to a good segment of data.

Create multiexperiment data set by merging data segments.

```
datam = merge(dat(1:250),dat(281:600),dat(651:1000));
```

Estimate a state-space model using the multiexperiment data set using experiments 1 and 2.

```
data_est = getexp(datam,[1,2]);
m = ssest(data_est,2);
```

Validate the model by comparing its output to the output data of experiment 3.

```
data_val = getexp(datam,3);
compare(data_val,m)
```
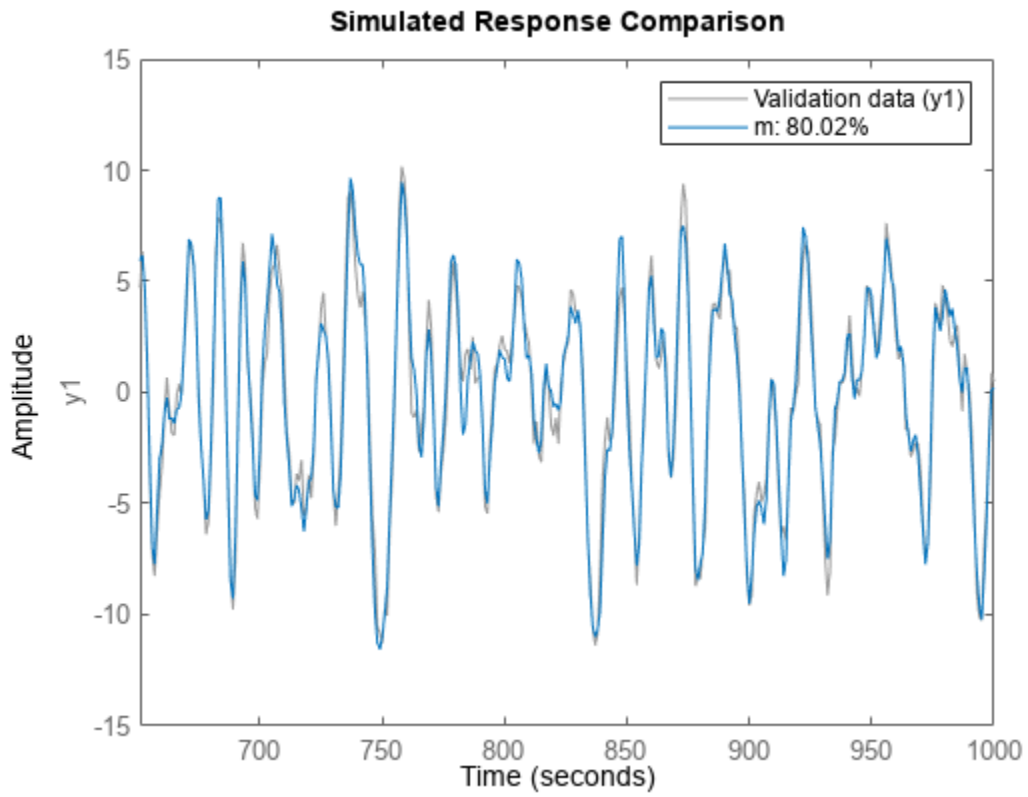
Handling Offsets and Trends in Data

When to Detrend Data

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

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

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

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

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

- Linear models that capture offsets essential for describing important system dynamics. For example, when a model contains integration behavior, you could estimate a low-order transfer function (process model) from nondetrended data. For more information, see "Process Models".
- Nonlinear black-box models, such as nonlinear ARX or Hammerstein-Wiener models. For more information, see "Nonlinear Model Identification".

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

- Nonlinear ODE parameters (nonlinear grey-box models). For more information, see "Estimate Nonlinear Grey-Box Models" on page 13-25.

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

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

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

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

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

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

- Subtracting piecewise linear trends at specified breakpoints. A *breakpoint* is a time value that defines the discontinuities between successive linear trends.

- Subtracting arbitrary offsets and linear trends from transient data signals.
- Saving trend information to a variable so that you can apply it to multiple data sets.

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

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

- Using `detrend`:

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

  model = tfest(datad, np);
  ```
- Specify offsets as estimation options:

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

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

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

Next Steps After Detrending

After detrending your data, you might do the following:

- Perform other data preprocessing operations. See "Ways to Prepare Data for System Identification" on page 2-5.
- Estimate a linear model. See "Linear Model Identification".

See Also

Related Examples

How to Detrend Data Using the App

Before you can perform this task, you must have regularly-sampled, steady-state time-domain data imported into the System Identification app. See "Import Time-Domain Data into the App" on page 2-13). For transient data, see "How to Detrend Data at the Command Line" on page 2-84.

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

---

1   In the System Identification app, drag the data set you want to detrend to the **Working Data** rectangle.

2   Detrend the data.

- To remove linear trends, select **Preprocess > Remove trends**.

- To remove mean values from each input and output data signal, select **Preprocess > Remove means**.

See Also

More About

- "Handling Offsets and Trends in Data" on page 2-81

How to Detrend Data at the Command Line

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 2-34.

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

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

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

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

Detrending Transient Data

Before you can perform this task, you must have

- Time-domain data as an `iddata` object. See "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-34.

   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 2-71.

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

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

   where `T` is a `TrendInfo` object.

2   Assign offset values to `T`.

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

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

3   Remove the specified offsets from `data`.

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

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

See Also
`detrend` | `TrendInfo`

More About

- "Handling Offsets and Trends in Data" on page 2-81

Resampling Data

What Is Resampling?

Resampling data signals in the System Identification Toolbox product applies an antialiasing (lowpass) FIR filter to the data and changes the sampling rate of the signal by decimation or interpolation.

If your data is sampled faster than needed during the experiment, you can decimate it without information loss. If your data is sampled more slowly than needed, there is a possibility that you miss important information about the dynamics at higher frequencies. Although you can resample the data at a higher rate, the resampled values occurring between measured samples do not represent new measured information about your system. Instead of resampling, repeat the experiment using a higher sampling rate.

Tip You should decimate your data when it contains high-frequency noise outside the frequency range of the system dynamics.

Resampling takes into account how the data behaves between samples, which you specify when you import the data into the System Identification app (zero-order or first-order hold). For more information about the data properties you specify before importing the data, see "Represent Data".

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

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

Resampling Data Without Aliasing Effects

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

The command `resample` performs the decimation without aliasing effects. This command includes a factor of *T* to normalize the spectrum and preserve the energy density after decimation. For more information about spectrum normalization, see "Spectrum Normalization" on page 9-10.

If you use manual decimation instead of `resample`—by picking every fourth sample from the signal, for example—the energy contributions from higher frequencies are folded back into the lower frequencies("aliasing"). Because the total signal energy is preserved by this operation and this energy must now be squeezed into a smaller frequency range, the amplitude of the spectrum at each frequency increases. Thus, the energy density of the decimated signal is not constant.

This example shows how `resample` avoids folding effects.

Construct a fourth-order moving-average process.

```
m0 = idpoly(1,[ ],[1 1 1 1]);
```

m0 is a time-series model with no inputs.

Generate error signal.

```
e = idinput(2000,'rgs');
```

Simulate the output using the error signal.

```
sim_opt = simOptions('AddNoise',true,'NoiseData',e);
y = sim(m0,zeros(2000,0),sim_opt);
y = iddata(y,[],1);
```

Estimate the signal spectrum.

```
g1 = spa(y);
```

Estimate the spectrum of the modified signal including every fourth sample of the original signal. This command automatically sets Ts to 4.

```
g2 = spa(y(1:4:2000));
```

Plot the frequency response to view folding effects.

```
h = spectrumplot(g1,g2,g1.Frequency);
opt = getoptions(h);
opt.FreqScale = 'linear';
opt.FreqUnits = 'Hz';
setoptions(h,opt);
```


Power Spectrum

From: e@y1 To: y1
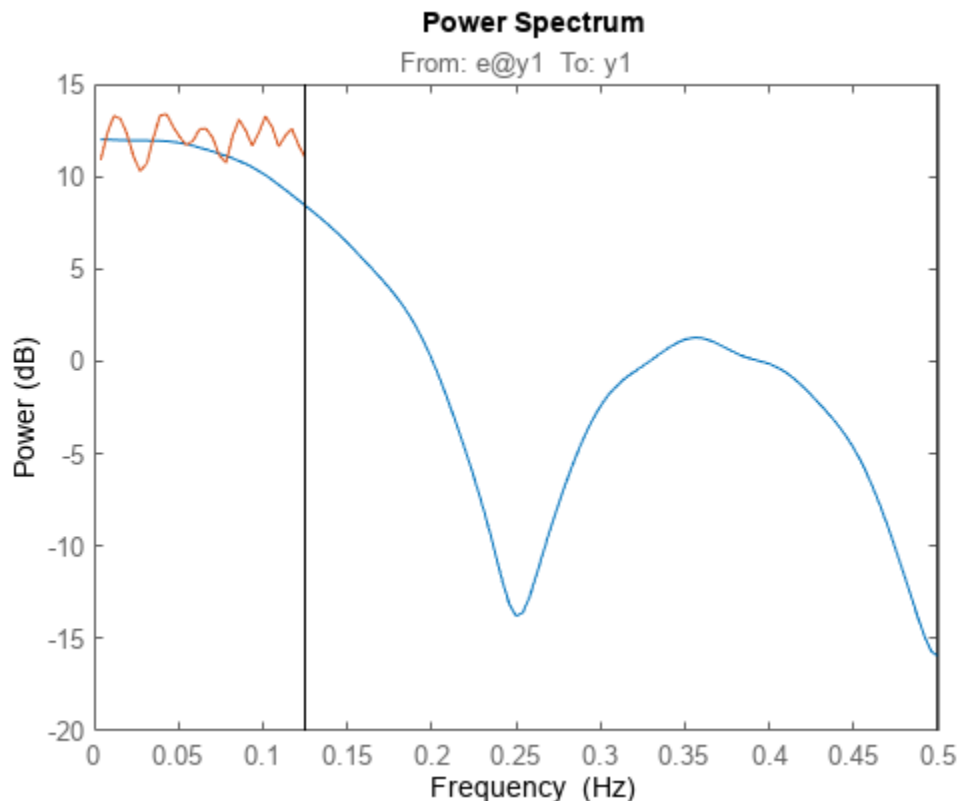
Estimate the spectrum after prefiltering that does not introduce folding effects.

```
g3 = spa(resample(y,1,4));
figure
spectrumplot(g1,g3,g1.Frequency,opt)
```

Power Spectrum

From: e@y1 To: y1
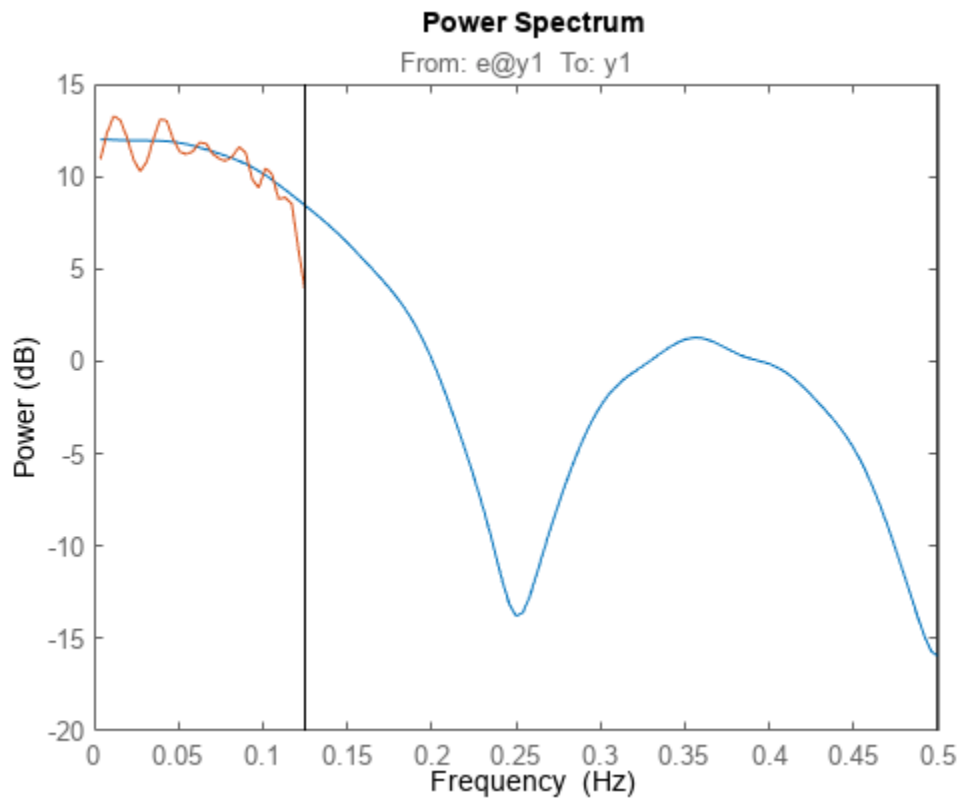
Use `resample` to decimate the signal before estimating the spectrum and plot the frequency response.

```
g3 = spa(resample(y,1,4));
figure
spectrumplot(g1,g3,g1.Frequency,opt)
```

Power Spectrum

From: e@y1 To: y1
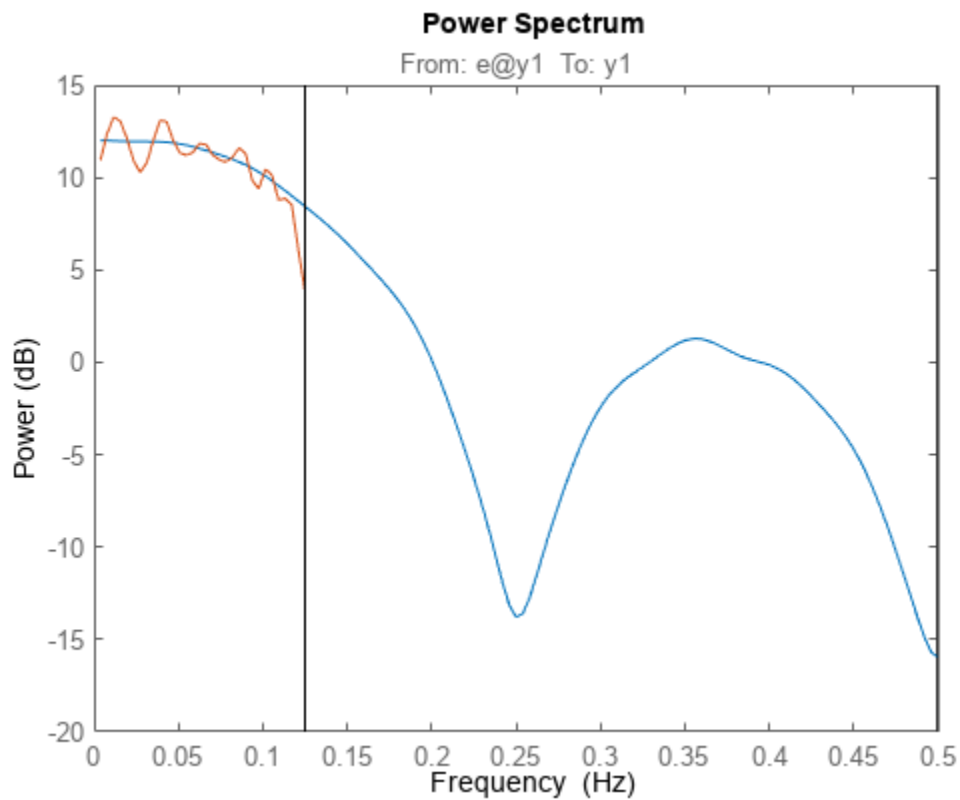
The plot shows that the estimated spectrum of the resampled signal has the same amplitude as the original spectrum. Thus, there is no indication of folding effects when you use `resample` to eliminate aliasing.

See Also

Related Examples

- "Resampling Data Using the App" on page 2-90
- "Resampling Data at the Command Line" on page 2-91

Resampling Data Using the App

Use the System Identification app to resample time-domain data. To specify additional options, such as the prefilter order, see "Resampling Data at the Command Line" on page 2-91.

The System Identification app uses `idresamp` to interpolate or decimate the data. For more information about this command, type `help idresamp` at the prompt.

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

1   Import time-domain data into the System Identification app, as described in "Create Data Sets from a Subset of Signal Channels" on page 2-24.

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

3   In the **Resampling factor** field, enter the factor by which to multiply the current sample time:

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

-   For interpolation (more samples), enter a factor less than 1 to decrease the sample time by this factor.

    Default = 1.

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

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

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

See Also

Related Examples

-   "Resampling Data at the Command Line" on page 2-91

More About

-   "Resampling Data" on page 2-86

Resampling Data at the Command Line

Use `resample` to decimate and interpolate time-domain `iddata` objects. You can specify the order of the antialiasing filter as an argument.

Note `resample` uses the Signal Processing Toolbox™ command, when this toolbox is installed on your computer. If this toolbox is not installed, use `idresamp` instead. `idresamp` only lets you specify the filter order, whereas `resample` also lets you specify filter coefficients and the design parameters of the Kaiser window.

To create a new `iddata` object `datar` by resampling `data`, use the following syntax:

```
datar = resample(data,P,Q,filter_order)
```

In this case, `P` and `Q` are integers that specify the new sample time: the new sample time is `Q/P` times the original one. You can also specify the order of the resampling filter as a fourth argument `filter_order`, which is an integer (default is `10`). For detailed information about `resample`, see the corresponding reference page.

For example, `resample(data,1,Q)` results in decimation with the sample time modified by a factor `Q`.

The next example shows how you can increase the sampling rate by a factor of 1.5 and compare the signals:

```
plot(u)
ur = resample(u,3,2);
plot(u,ur)
```

When the Signal Processing Toolbox product is not installed, using `resample` calls `idresamp` instead.

`idresamp` uses the following syntax:

```
datar = idresamp(data,R,filter_order)
```

In this case, `R=Q/P`, which means that data is interpolated by a factor `P` and then decimated by a factor `Q`. To learn more about `idresamp`, type `help idresamp`.

The `data.InterSample` property of the `iddata` object is taken into account during resampling (for example, first-order hold or zero-order hold). For more information, see "iddata Properties" on page 2-35.

See Also

Related Examples

- "Resampling Data Using the App" on page 2-90

More About

- "Resampling Data" on page 2-86

Filtering Data

Supported Filters

You can filter the input and output signals through a linear filter before estimating a model in the System Identification app or at the command line. How you want to handle the noise in the system determines whether it is appropriate to prefilter the data.

The filter available in the System Identification app is a fifth-order (passband) Butterworth filter. If you need to specify a custom filter, use the `idfilt` command.

Choosing to Prefilter Your Data

Prefiltering data can help remove high-frequency noise or low-frequency disturbances (drift). The latter application is an alternative to subtracting linear trends from the data, as described in "Handling Offsets and Trends in Data" on page 2-81.

In addition to minimizing noise, prefiltering lets you focus your model on specific frequency bands. The frequency range of interest often corresponds to a passband over the breakpoints on a Bode plot. For example, if you are modeling a plant for control-design applications, you might prefilter the data to specifically enhance frequencies around the desired closed-loop bandwidth.

Prefiltering the input and output data through the same filter does not change the input-output relationship for a linear system. However, prefiltering does change the noise characteristics and affects the estimated noise model of the system.

To get a reliable noise model in the app, instead of prefiltering the data, set **Focus** to `Filter`, and specify the filter. To get a reliable noise model at the command line, instead of prefiltering the data, specify the filter in the `WeightingFilter` estimation option of the estimation command. If the `Focus` option is available, specify it as `'simulation'`.

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

For practical examples of prefiltering data, see the section on posttreatment of data in *Modeling of Dynamic Systems*, by Lennart Ljung and Torkel Glad, Prentice Hall PTR, 1994.

See Also

Related Examples

How to Filter Data Using the App

Filtering Time-Domain Data in the App

The System Identification app lets you filter time-domain data using a fifth-order Butterworth filter by enhancing or selecting specific passbands.

To create a filtered data set:

1 Import time-domain data into the System Identification app, as described in "Represent Data".

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

3 Select **<--Preprocess > Filter**. By default, this selection shows a periodogram of the input and output spectra (see the `etfe` reference page).

> **Note** To display smoothed spectral estimates instead of the periodogram, select **Options > Spectral analysis**. This spectral estimate is computed using `spa` and your previous settings in the Spectral Model dialog box. To change these settings, select **<--Estimate > Spectral model** in the System Identification app, and specify new model settings.

4 If your data contains multiple input/output channels, in the **Channel** menu, select the channel pair you want to view. Although you view only one channel pair at a time, the filter applies to all input/output channels in this data set.

5 Select the data of interest using one of the following ways:

- Graphically — Draw a rectangle with the mouse on either the input-signal or the output-signal plot to select the desired frequency interval. Your selection is displayed on both plots regardless of the plot on which you draw the rectangle. The **Range** field is updated to match the selected region. If you need to clear your selection, right-click the plot.

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

  For example:

  `8.5 20.0` (rad/s).

  > **Tip** To change the frequency units from `rad/s` to `Hz`, select **Style > Frequency (Hz)**. To change the frequency units from `Hz` to `rad/s`, select **Style > Frequency (rad/s)**.

6 In the **Range is** list, select one of the following:

- `Pass band` — Allows data in the selected frequency range.
- `Stop band` — Excludes data in the selected frequency range.

7 Click **Filter** to preview the filtered results. If you are satisfied, go to step 8. Otherwise, return to step 5.

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

9 Click **Insert** to save the selection as a new data set and add it to the Data Board.

10 To select another range, repeat steps 5 to 9.

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

For frequency-domain and frequency-response data, *filtering* is equivalent to selecting specific data ranges.

To select a range of data in frequency-domain or frequency-response data:

1 Import data into the System Identification app, as described in "Represent Data".

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

3 Select **<--Preprocess > Select range**. This selection displays one of the following plots:

   - Frequency-domain data — Plot shows the absolute of the squares of the input and output spectra.

   - Frequency-response data — Top axes show the frequency response magnitude equivalent to the ratio of the output to the input, and the bottom axes show the ratio of the input signal to itself, which has the value of 1 at all frequencies.

4 If your data contains multiple input/output channels, in the **Channel** menu, select the channel pair you want to view. Although you view only one channel pair at a time, the filter applies to all input/output channels in this data set.

5 Select the data of interest using one of the following ways:

   - Graphically — Draw a rectangle with the mouse on either the input-signal or the output-signal plot to select the desired frequency interval. Your selection is displayed on both plots regardless of the plot on which you draw the rectangle. The **Range** field is updated to match the selected region.

     If you need to clear your selection, right-click the plot.

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

     For example:

     8.5 20.0 (rad/s).

     Tip If you need to change the frequency units from rad/s to Hz, select **Style > Frequency (Hz)**. To change the frequency units from Hz to rad/s, select **Style > Frequency (rad/s)**.

6 In the **Range is** list, select one of the following:

   - Pass band — Allows data in the selected frequency range.

   - Stop band — Excludes data in the selected frequency range.

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

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

9 To select another range, repeat steps 5 to 8.

See Also

Related Examples

   - "How to Filter Data at the Command Line" on page 2-96

More About

- "Filtering Data" on page 2-92

How to Filter Data at the Command Line

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, `wl` and `wh` represent the low and high frequencies of the passband, respectively.

You can specify several passbands, as follows:

```
filter=[[w1l,w1h];[ w2l,w2h]; ....;[wnl,wnh]]
```

The filter is an `n`-by-2 matrix, where each row defines a passband in radians per second.

To define a stopband between `ws1` and `ws2`, use

```
filter = [0 ws1; ws2 Nyqf]
```

where, `Nyqf` is the Nyquist frequency.

For time-domain data, the passband filtering is cascaded Butterworth filters of specified order. The default filter order is 5. The Butterworth filter is the same as `butter` in the Signal Processing Toolbox product. For frequency-domain data, select the indicated portions of the data to perform passband filtering.

Defining a Custom Filter

Use `idfilt` to apply passband and other custom filters to a time-domain or a frequency-domain `iddata` object.

In general, you can specify any custom filter. Use this syntax to filter an `iddata` object `data` using the filter called `filter`:

```
fdata = idfilt(data,filter)
```

You can define a general single-input/single-output (SISO) system for filtering time-domain or frequency-domain data. For frequency-domain only, you can specify the (nonparametric) frequency response of the filter.

You use this syntax to filter an `iddata` object `data` using a custom filter specified by `filter`:

```
fdata = idfilt(data,filter)
```

`filter` can be also any of the following:

```
filter = idm
filter = {num,den}
filter = {A,B,C,D}
```

`idm` is a SISO identified linear model on page 1-10 or LTI object. For more information about LTI objects, see the Control System Toolbox documentation.

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

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

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

```
filter = Wf
```

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

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

Causal and Noncausal Filters

For time-domain data, the filtering is causal by default. Causal filters typically introduce a phase shift in the results. To use a noncausal zero-phase filter (corresponding to `filtfilt` in the Signal Processing Toolbox product), specify a third argument in `idfilt`:

```
fdata = idfilt(data,filter,'noncausal')
```

For frequency-domain data, the signals are multiplied by the frequency response of the filter. With the filters defined as passband filters, this calculation gives ideal, zero-phase filtering ("brick wall filters"). Frequencies that have been assigned zero weight by the filter (outside the passband or via frequency response) are removed.

When you apply `idfilt` to an `idfrd` data object, the data is first converted to a frequency-domain `iddata` object (see "Transforming Between Frequency-Domain and Frequency-Response Data" on page 3-14). The result is an `iddata` object.

See Also

More About

*   "Filtering Data" on page 2-92

Generate Data Using Simulation

Commands for Generating Data Using Simulation

You can generate input data and then use it with a model to create output data.

Simulating output data requires that you have a model with known coefficients. For more information about commands for constructing models, see "Commands for Constructing Linear Model Structures" on page 1-16.

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

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

Commands for Generating Data

Command	Description	Example
`idinput`	Constructs a signal with the desired characteristics, such as a random Gaussian or binary signal or a sinusoid, and returns a matrix of input values.	`u = iddata([],...` `    idinput(400,'rbs',[0 0.3]));`
`sim`	Simulates response data based on existing linear or nonlinear parametric model in the MATLAB workspace.	To simulate the model output `y` for a given input, use the following command: `y = sim(m,data)` `m` is the model object name, and `data` is input data matrix or `iddata` object.

Create Periodic Input Data

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

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

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

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

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

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

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

Power Spectrum

From: e@u1 To: u1
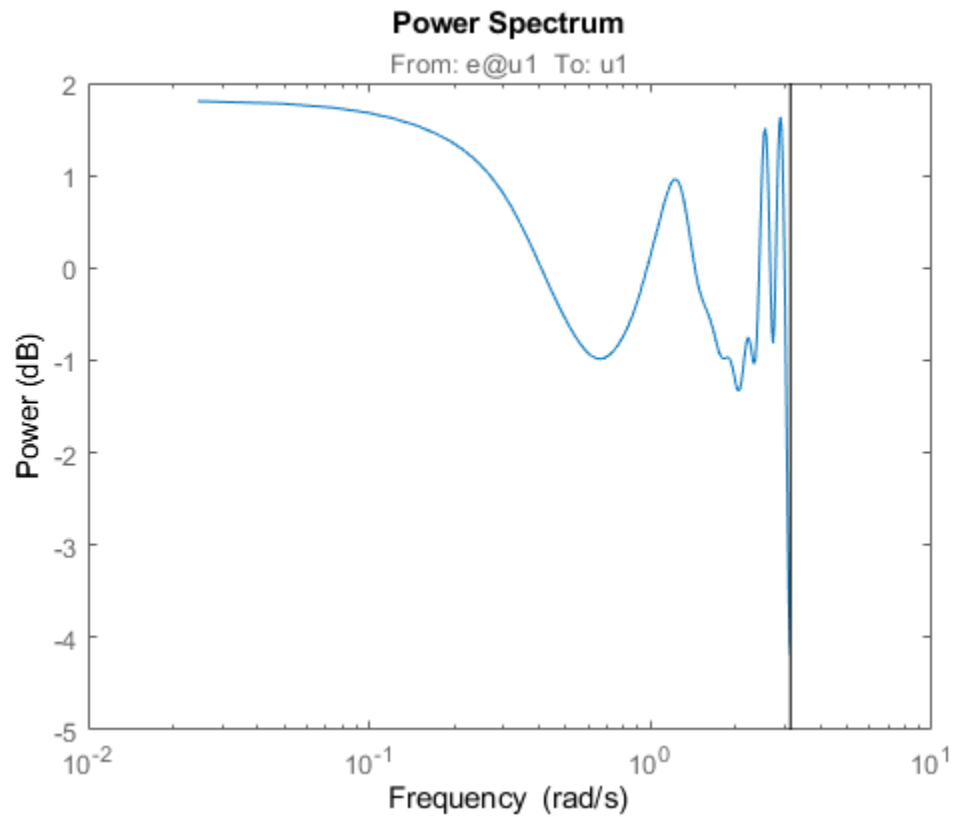
(Optional) Simulate model output using the data.

```matlab
% 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)
```

Simulated response
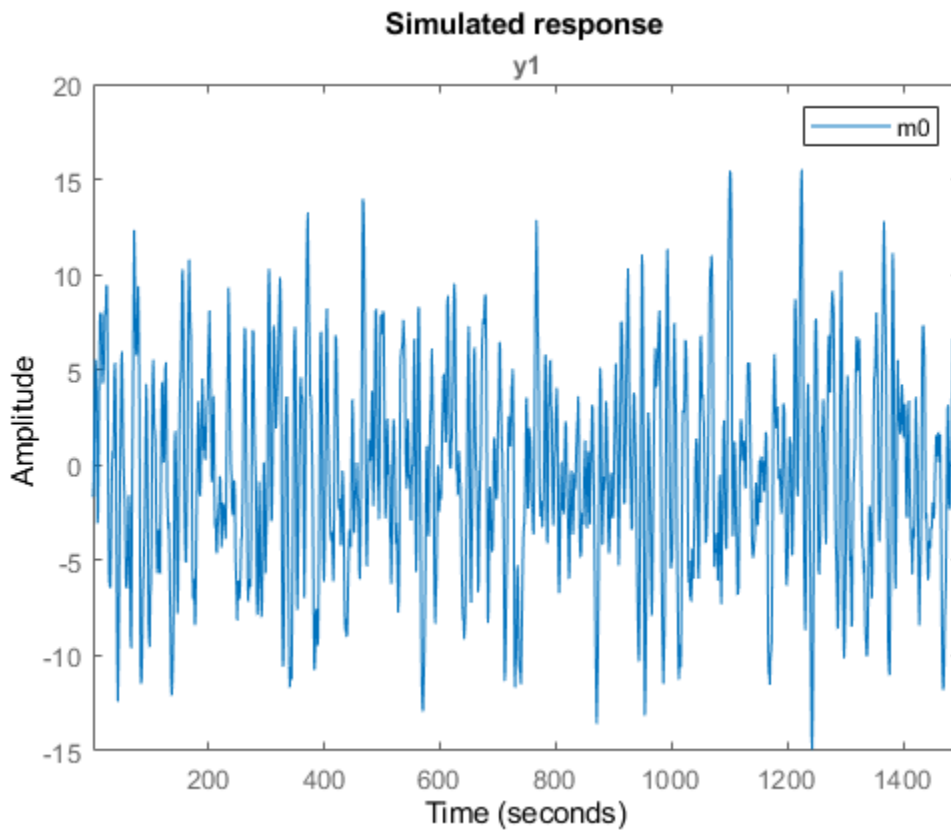
Generate Output Data Using Simulation

This example shows how to generate output data by simulating a model using an input signal created using `idinput`.

You use the generated data to estimate a model of the same order as the model used to generate the data. Then, you check how closely both models match to understand the effects of input data characteristics and noise on the estimation.

Create an ARMAX model with known coefficients.

```
A = [1 -1.2 0.7];
B = {[0 1 0.5 0.1],[0 1.5 -0.5],[0 -0.1 0.5 -0.1]};
C = [1 0 0 0 0];
Ts = 1;
m0 = idpoly(A,B,C,'Ts',1);
```

The leading zeros in the B matrix indicate the input delay (nk), which is 1 for each input channel.

Construct a pseudorandom binary input data.

```
u = idinput([255,3],'prbs');
```

Simulate model output with noise using the input data.

```
y = sim(m0,u,simOptions('AddNoise',true));
```

Represent the simulation data as an `iddata` object.

```
iodata = iddata(y,u,m0.Ts);
```

(Optional) Estimate a model of the same order as `m0` using `iodata`.

```
na = 2;
nb = [3 2 3];
nc = 4;
nk = [1 1 1];
me = armax(iodata,[na,nb,nc,nk]);
```

Use `bode(m0,me)` and `compare(iodata,me)` to check how closely `me` and `m0` match.

```
compare(iodata,me);
```
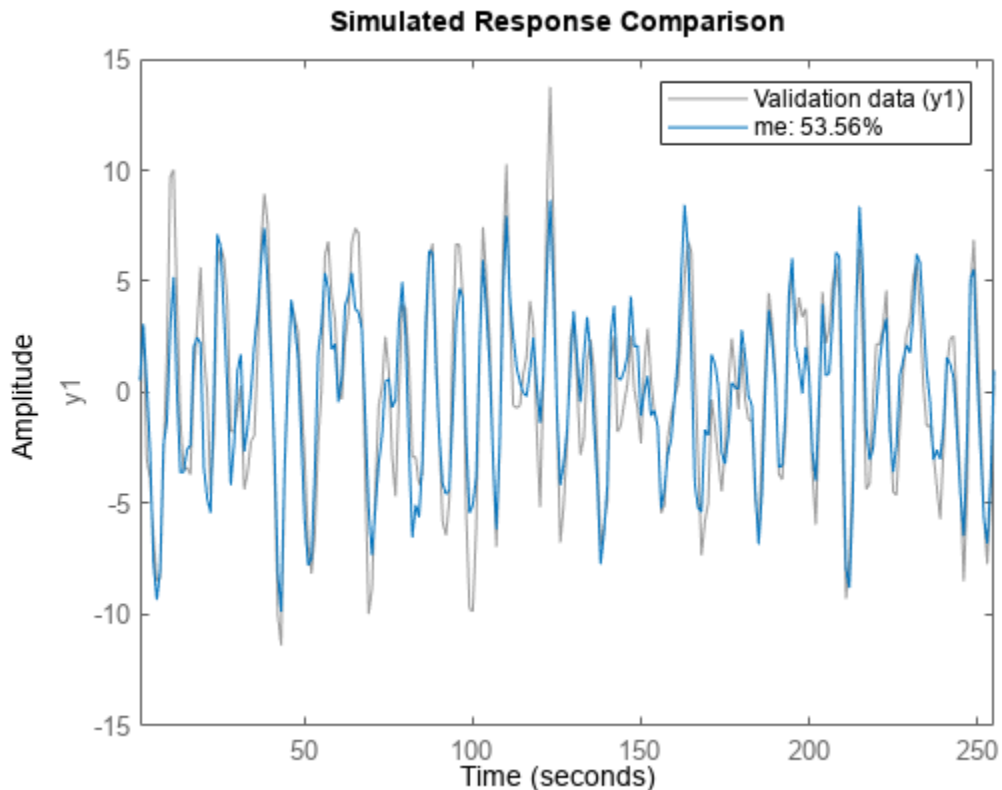

Simulating Data Using Other MathWorks Products

You can also simulate data using the Simulink and Signal Processing Toolbox software. Data simulated outside the System Identification Toolbox product must be in the MATLAB workspace as double matrices. For more information about simulating models using the Simulink software, see "Simulate Identified Model in Simulink" on page 20-5.

Manipulating Complex-Valued Data

Supported Operations for Complex Data

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

Model transformation routines, such as `freqresp` and `zpkdata`, work for complex-valued models. However, they do not provide pole-zero confidence regions. For complex models, the parameter variance-covariance information refers to the complex-valued parameters and the accuracy of the real and imaginary is not computed separately.

The display commands `compare` and `plot` also work with complex-valued data and models. To plot the real and imaginary parts of the data separately, use `plot(real(data))` and `plot(imag(data))`, respectively.

Processing Complex iddata Signals at the Command Line

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

Command	Description
`abs(data)`	Absolute value of complex signals in `iddata` object.
`angle(data)`	Phase angle (in radians) of each complex signals in `iddata` object.
`complex(data)`	For time-domain data, this command makes the `iddata` object complex— even when the imaginary parts are zero. For frequency-domain data that only stores the values for nonnegative frequencies, such that `realdata(data)=1`, it adds signal values for negative frequencies using complex conjugation.
`imag(data)`	Selects the imaginary parts of each signal in `iddata` object.
`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
```

Convert SISO Matrix Data to Timetable

Load the data, which consists of the input vector `umat1`, the output vector `ymat1`, and the sample time `Ts`.

```
load sdata1 umat1 ymat1 Ts
```

Combine the data into the single timetable `tt`. View the first two rows of `tt`.

```
tt = timetable(umat1,ymat1,'rowtimes',seconds(Ts*(1:size(umat1,1))));
head(tt,2)
```

```
   Time        umat1     ymat1
  _____     _____    _____

  0.1 sec        1      -0.58724
  0.2 sec       -1       1.1082
```

Use `tt` to estimate a continuous-time state-space model.

```
sys = ssest(tt,2);
```

Compare the model output with the estimation data.

```
compare(tt,sys)
```


2-103
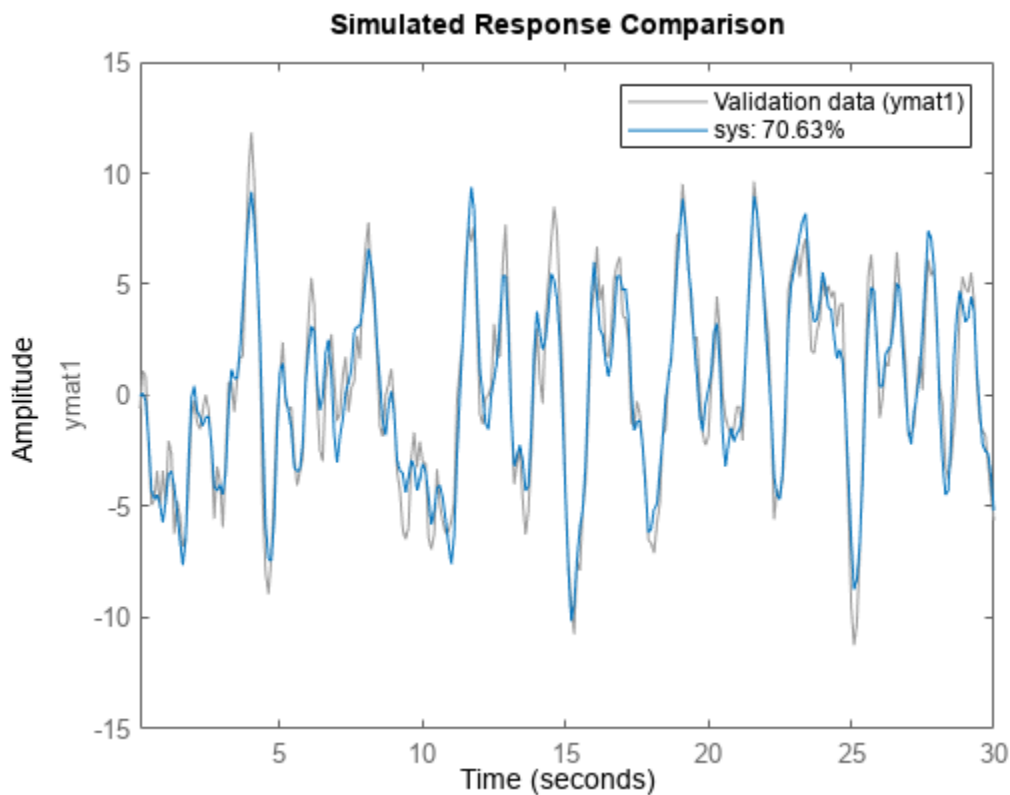
Convert MIMO Matrix Data to Timetable for Continuous-Time Model Estimation

Estimate a continuous-time model function by first converting matrix data to a timetable.

Load the data, which includes input matrix `usteam`, output matrix `ysteam`, and sample time `Ts`.

```
load sdatasteam.mat usteam ysteam Ts
```

Combine `usteam` and `ysteam` into the single timetable `ttsteam`. In order to create a variable for each channel, you must specify each matrix column explicitly.

```
tts = timetable(usteam(:,1),usteam(:,2),ysteam(:,1),ysteam(:,2),...
    'rowtimes',seconds(Ts*(1:size(usteam,1))));
head(tts,4)
```

Time	Var1	Var2	Var3	Var4
0.05 sec	-1.5283	2.0584	0.57733	-0.12274
0.1 sec	1.4412	-2.005	0.75804	-0.086114
0.15 sec	1.4314	2.0584	-0.76577	-0.19845
0.2 sec	1.4412	-1.9806	0.47721	-0.20577

Estimate a continuous-time state-space model.

```
nx = 3;
sysc = ssest(tts,nx,'InputName',["Var1" "Var2"],'OutputName',["Var3" "Var4"]);
```

Compare the model to the data.

```
compare(tts,sysc)
```
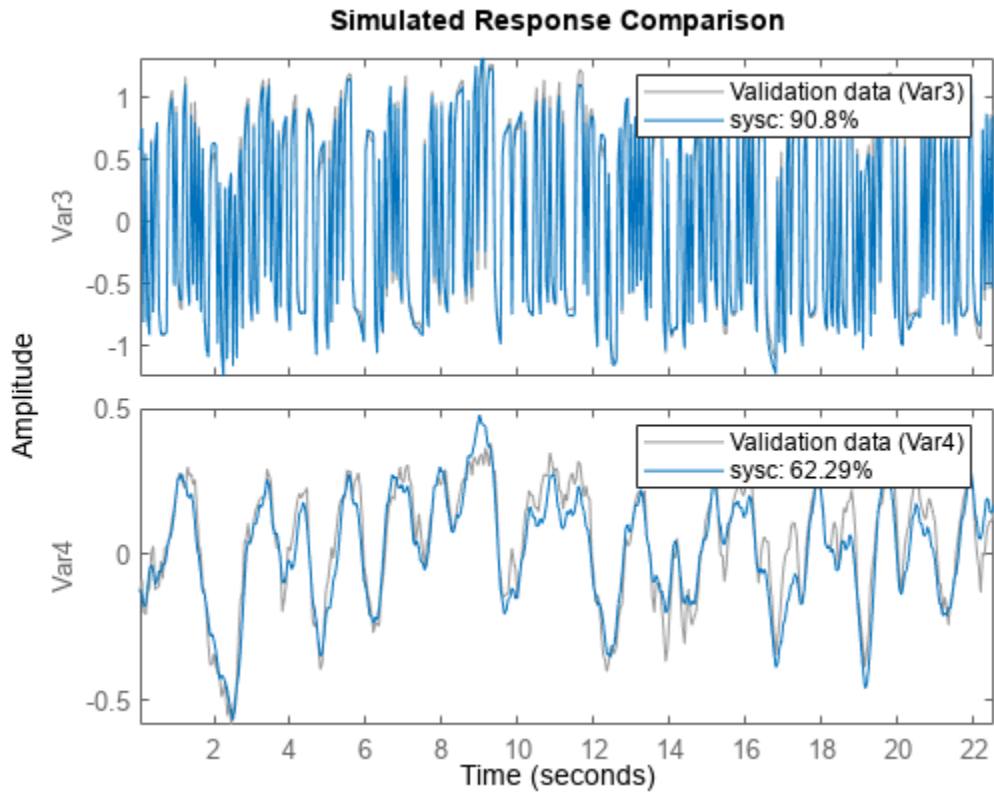
Data Types in System Identification Toolbox

This example describes different data types and formats supported by System Identification Toolbox™.

The toolbox supports model identification using time-domain signals, frequency-domain signals and frequency-response data (also called *Frequency Response Function*, or FRF). Typically there is no restriction on the number of input and output signals; most estimation commands support an arbitrary number of I/O signals. In particular, there is full support for time-series modeling, or signal modeling, where the data is typically described as having an arbitrary number of outputs, but no inputs.

How Data of Different Types are Generated

The origin of signals is in time-domain where the system or process to be modeled is excited by an external stimulus (the "input" **u(t)**) and the corresponding response of the system (the "output" **y(t)**) recorded. These signals are recorded over a finite span of time **t** resulting in vectors or matrices storing the measured values. The triplet of recorded values $\{y, u, t\}$ denoted by D_1 in the schematic diagram below forms the **time-domain data**. Since the data can only be recorded at discrete time instants, sometimes an extra piece of information is also required for modeling - the intersample behavior.
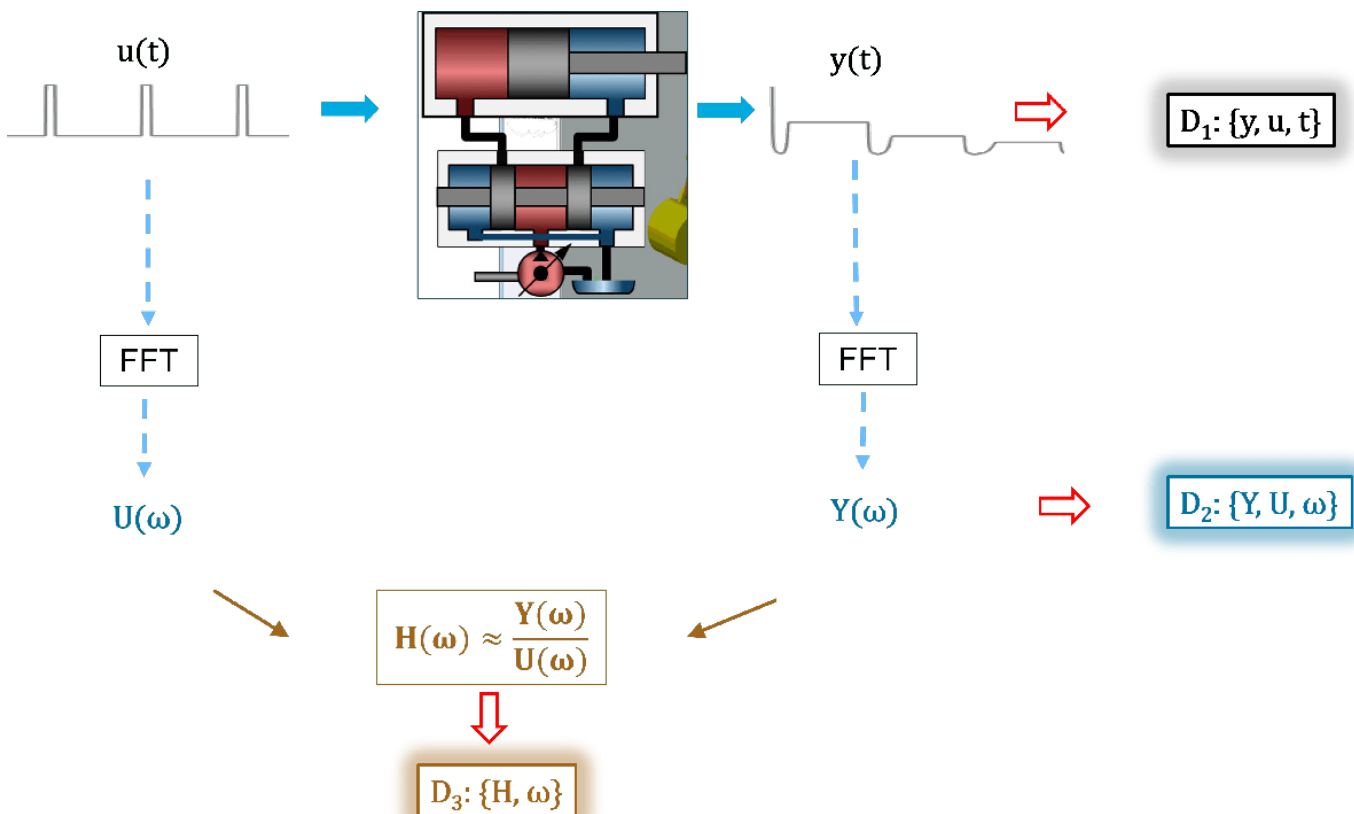
The time-domain signals can be converted into their frequency-domain counterparts by using Discrete Fourier Transform, typically carried out using the FFT operation. This operation requires the

time domain signals to be collected on (or interpolated to) a uniform time-grid, that is, made available at a constant sampling interval T_s. Unless zero-padding is used, the resulting signals correspond to a uniform frequency grid of as many samples as in the original (time-domain) signals and spaced over 0 to Nyquist frequency (= π/T_s, where T_s is the time-domain data sampling interval). The triplet $\{Y, U, \omega\}$ is called **frequency-domain data**. It is shown by D_2 in the diagram above. To make the frequency-domain data independent of any correspondence to another time-domain dataset, the sample time T_s is also stored.

The dynamic behavior of systems is often described by its frequency response, which is a collection of the gain and relative phase of the output of the system to a sinusoidal input, over a range of input frequencies. Typically spectrum analyzers compute these values. They can also be computed by applying spectral analysis techniques to the time- or frequency-domain data. The result is a complex frequency response vector $H(\omega)$ computed over a range of ω values. The pair $\{H, \omega\}$ is called the **frequency response data** (FRD) or **frequency response function** (FRF), shown as D_3 in the diagram above.

To summarize, System Identification Toolbox supports these 3 data types:

1 Time-domain data: composed of input, output signal values for a given time-vector. The input signal is not aways present, such as in case of time-series modeling. Both linear and nonlinear models can be identified using time-domain data.

2 Frequency-domain data: FFT of time-domain signals. Typically used for linear model identification. They offer the benefit of data compression and fast estimation algorithms.

3 Frequency response data (FRD): gain and phase of the system output relative to its input, recorded over a range sinusoidal inputs of varying frequencies. Like frequency-domain data, the FRDs provide the benefit of data compression and fast estimation algorithms. They are used for identifying linear models.

Data Formats: Representation of Various Data Types for Identification Purposes

System Identification Toolbox supports 3 ways of representing time-domain data - timetables, numeric matrices, and iddata objects.

1. Timetables

Timetables are a built-in MATLAB datatype represented by the timetable objects. They are used to represent observations that are ordered by time and hence are suitable for time-domain identification. You can create timetables by using the timetable constructor, or by using the `array2timetable` command. Timetables can be created automatically when importing data into MATLAB from data files (spreadsheets, csv files etc).

The time vector associated with the data in a timetable TT can be accessed using its `TT.Properties.RowTimes` property. Identification typically requires uniformly sampled data (has a finite value for its "`TT.Properties.SampleRate`" and "`TT.Properties.TimeStep`" properties).Timetables are supported for model identification, release R2022b onwards. For earlier releases, you must use iddata object to represent data. For the identification of Neural State-Space models (represented by the `idNeuralStateSpace` object) in the continuous-time form, the timetable sample points need not be uniformly separated.

Timetables can store non-numeric data, as well as samples of arbitrary sizes for each time value. However, only the numeric, scalar-valued variables (that is, the variables whose values are scalars for a given time sample) can be used for identification of models. Suppose you have data for two signals

u1 and u2 that share a common time vector T. There are two ways of storing these signals in a timetable:

- As a single variable whose value is a 2-element vector [u1(t), u2(t)] for a given time t.
- As two distinct variables, one for each signal.

In System Identification Toolbox, the second option - using distinct variables for each signal, is the only supported representation. Here is an example to see the difference more clearly.

```
% Make up some data to illustrate the difference between the two ways of
% creating the timetable
T = seconds(1:5)'; % 5 samples
u1 = rand(5,1);
u2 = rand(5,1);

% Option 1: Joint representation of u1, u2 into a single variable
TT1 = timetable(T, [u1, u2], 'VariableNames', {'U'});
head(TT1)

      T                  U

    _____      _____

    1 sec     0.81472      0.09754
    2 sec     0.90579       0.2785
    3 sec     0.12699      0.54688
    4 sec     0.91338      0.95751
    5 sec     0.63236      0.96489


% Option 2: Separate variable for each signal
TT2 = timetable(T, u1, u2, 'VariableNames', {'u1','u2'});
head(TT2)

      T          u1          u2

    _____      _____      _____

    1 sec     0.81472      0.09754
    2 sec     0.90579       0.2785
    3 sec     0.12699      0.54688
    4 sec     0.91338      0.95751
    5 sec     0.63236      0.96489
```

The timetable TT1 cannot be used for identification/analysis purposes. TT2 is the only valid representation. Let us now see how a timetable can be used for identifying a model.

```
% Example: Estimate a continuous-time transfer function
% using data represented by a timetable
load sdata8 tt8
head(tt8) % view the first few rows of the variables in timetable tt8

      t       u1    u2      u3            y

    _____     __    __    _____    _____

    1 sec      1     1    -0.62559      1.3563
    2 sec     -1    -1     0.38628      1.2895
    3 sec      1    -1    -0.61745     -0.61873
    4 sec      1     1    -0.72216     -0.10768
    5 sec      1    -1    -0.41638      2.9138
```

```
      6 sec      1     -1    -0.002612      0.58144
      7 sec     -1     -1      0.38949      0.20034
      8 sec     -1     -1      -1.1337     -1.0462
```

```
np = 2; % number of poles
nz = 2; % number of zeros

% Estimate a model using only one input ('u2') and output ('y')
% (this is just to illustrate a calling syntax; the model quality is not good)
sysTF_a = tfest(tt8, np, nz, 'InputName','u2','OutputName','y');

% Estimate a model with 3 inputs ('u1', 'u2', 'u3') and 1 output ('y')
sysTF_b = tfest(tt8, np, nz, 'InputName',{'u1','u2','u3'},'OutputName','y')
```

```
sysTF_b =
  From input "u1" to output "y":
  -0.06172 s^2 + 0.1762 s + 1.836
  -------------------------------
      s^2 + 0.3634 s + 0.2453

  From input "u2" to output "y":
  1.025 s^2 + 1.679 s + 1.211
  ---------------------------
    s^2 + 0.3707 s + 0.2436

  From input "u3" to output "y":
  -0.06868 s^2 - 0.3401 s + 0.5967
  -------------------------------
      s^2 + 0.3603 s + 0.2496

Continuous-time identified transfer function.

Parameterization:
   Number of poles: [2 2 2]   Number of zeros: [2 2 2]
   Number of free coefficients: 15
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using TFEST on time domain data.
Fit to estimation data: 80.91%
FPE: 1.068, MSE: 0.9817

Model Properties
```

Timetables can similarly be in all the model validation and analysis operations that require input/output data.

```
% Compare the responses of models sysTF_a and sysTF_b to the measured output in tt8
compare(tt8, sysTF_a, sysTF_b)
```

Simulated Response Comparison
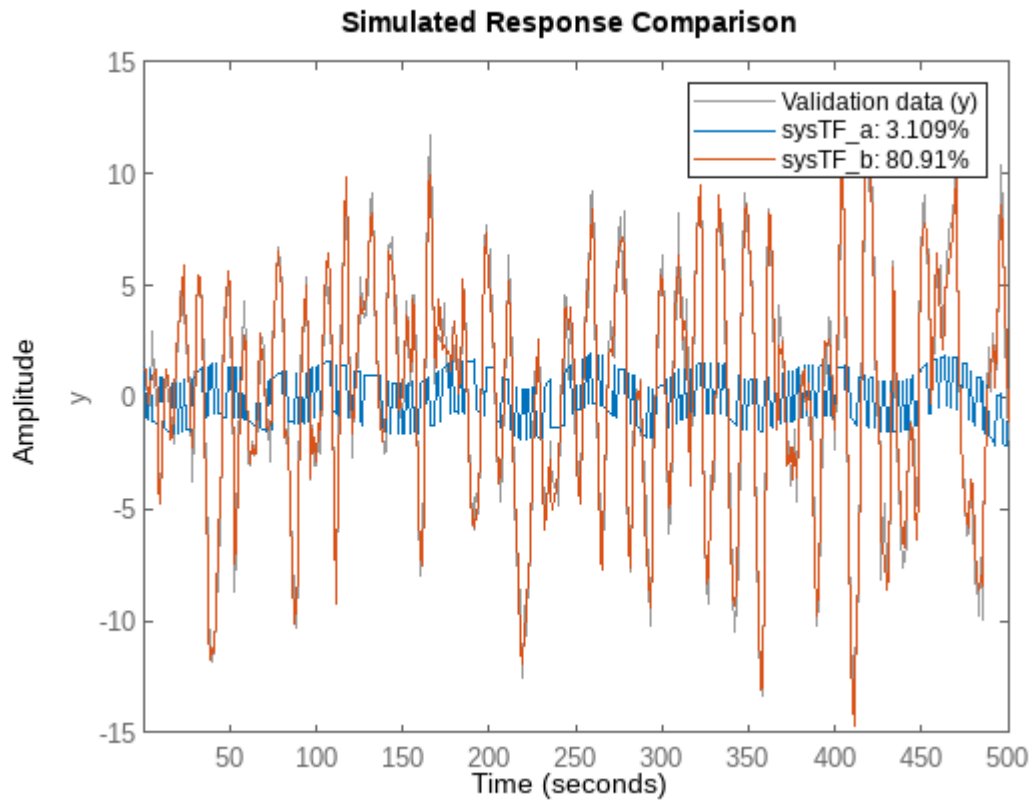
Note that the input intersample behavior has no effect on discrete-time estimations.

```
% Perform residual analysis using the timetable tt8 to represent the data.
resid(tt8, sysTF_b)
```

Residue Correlation
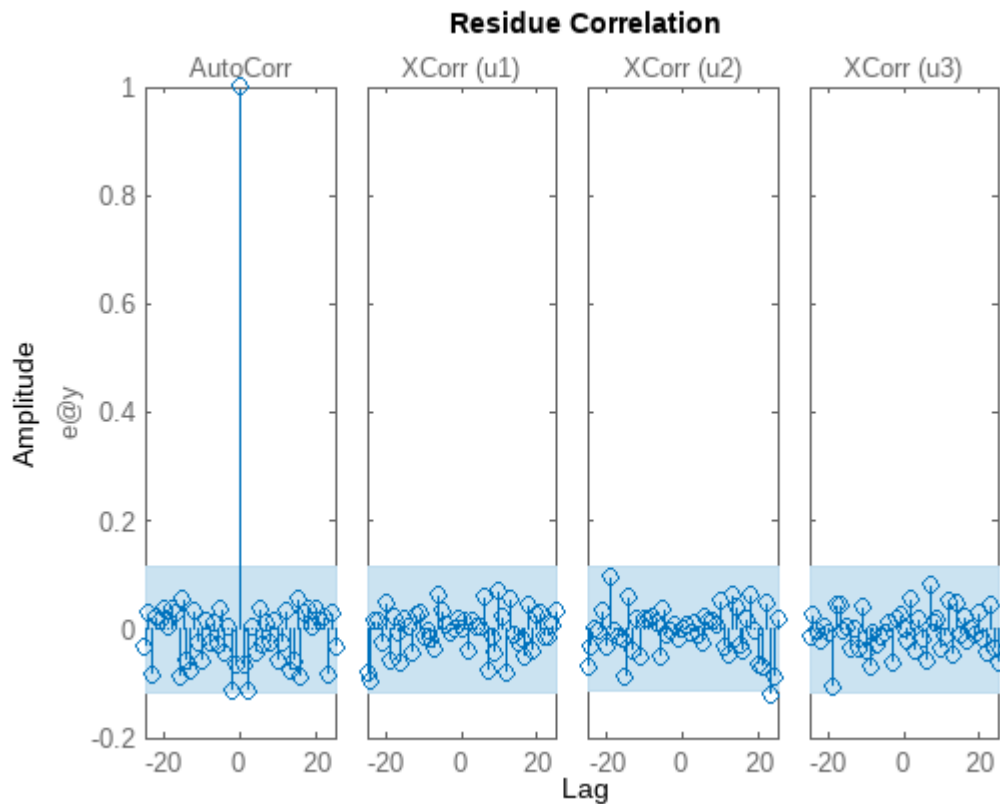
```
% Simulate the model sysTF_a using the input signal variables in timetable tt8
InputData = tt8(:,{'u2'}); % separate out the variables that are inputs for the model
head(InputData)
```

```
      t        u2

    _____      __

    1 sec       1
    2 sec      -1
    3 sec      -1
    4 sec       1
    5 sec      -1
    6 sec      -1
    7 sec      -1
    8 sec      -1
```

```
sim(sysTF_a, InputData(1:20,:)) % simulate the model response for 20 seconds
```

Simulated response
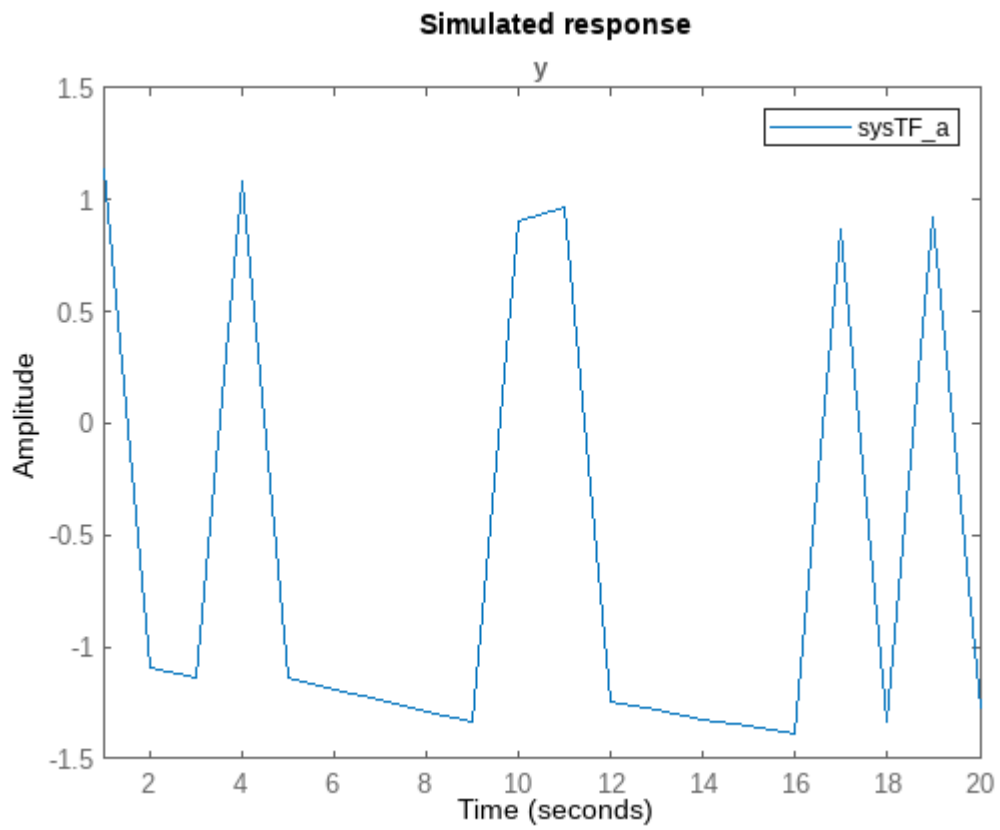
It is not necessary to separate out the variables to use. In the simulation example above, the same result would be obtained if the original timetable tt8 was used in the `sim` command.

```
sim(sysTF_a, tt8(1:20,:))
```

Simulated response
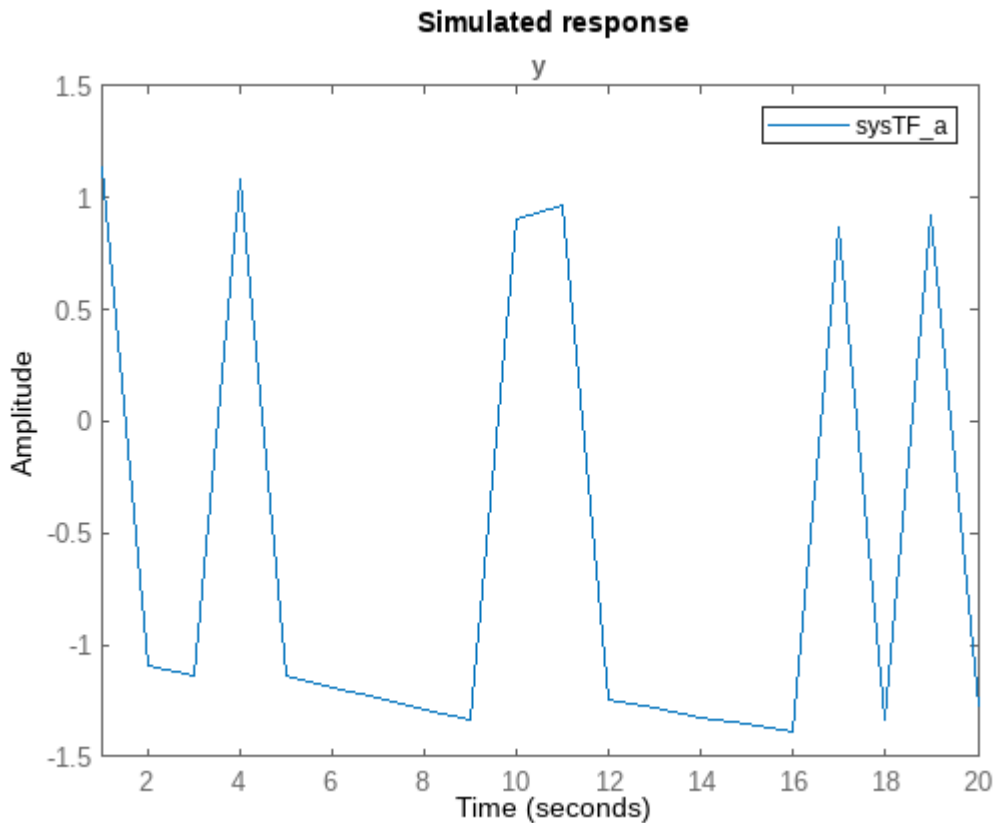
Timetables: Representing Multi-Experiment Data

You can use multiple datasets together for model identification. If the data is in timetables, you pass a cell array of the timetables to all the estimation and analysis commands. The individual timetables in the cell array are referred to as "experiments". Thus, the variable X = {TT1, TT2, TT3} represents a multi-experiment dataset consisting of 3 experiments TT1, TT2, and TT3. Each experiment must have the same datatype (timetable here), the same active numeric variables, and the same sample time and time units. The start time, and the number of observations in the individual experiments may be different.

```
% Load two timetables - tt1 and tt2.
% For the purpose of this example only, assume that they both correspond to the same system
% The datasets have different lengths, but are sample at the same rate (10 Hz)
load sdata1 tt1
load sdata2 tt2

% Create a multi-experiment dataset
MultiExpData = {tt1, tt2};

% Estimate a nonlinear ARX model using MultiExpData
Orders = [5 5 1];
NL = idGaussianProcess;
nlsys = nlarx(MultiExpData, Orders, NL)

nlsys =

Nonlinear ARX model with 1 output and 1 input
```

```
    Inputs: u
    Outputs: y

  Regressors:
    Linear regressors in variables y, u
    List of all regressors

  Output function: Gaussian process function using a SquaredExponential kernel
  Sample time: 0.1 seconds

  Status:
  Estimated using NLARX on time domain data "MultiExpData".
  Fit to estimation data: [79.04 87.04]% (prediction focus)
  FPE: 0.9208, MSE: [0.8527 0.8901]
  More information in model's "Report" property.

  Model Properties
```

```matlab
% Compare the model response to the estimation data
compare(MultiExpData, nlsys)
```


Simulated Response Comparison
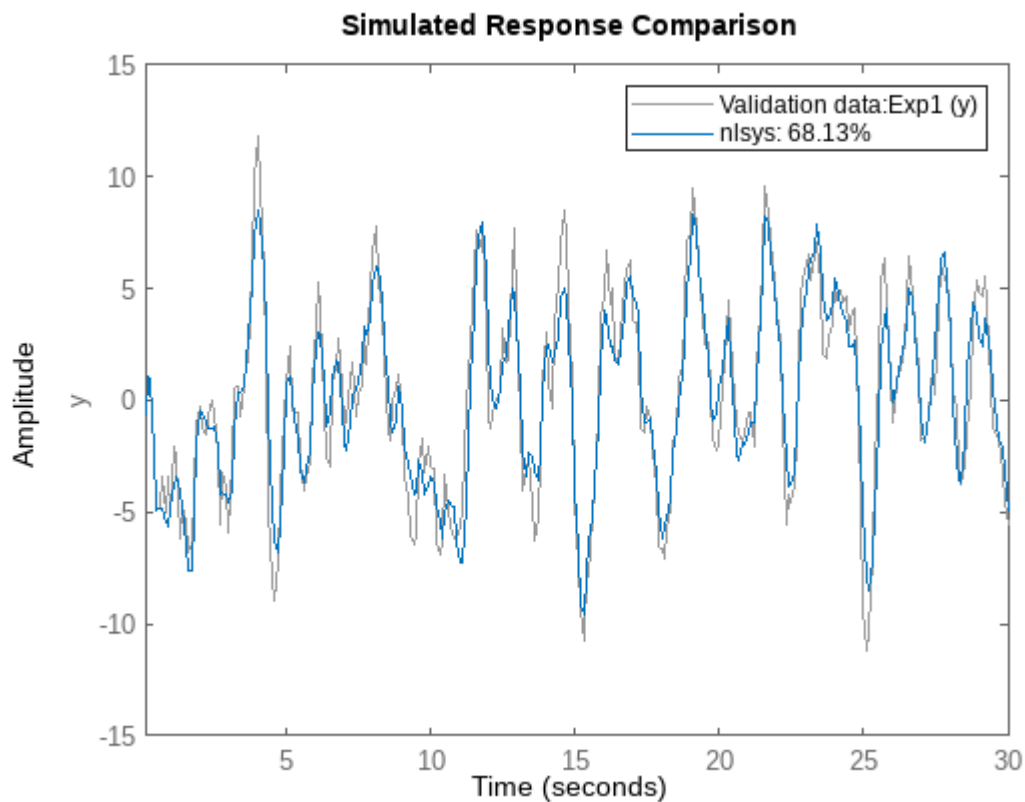
The plot shows the fit to the first dataset. Right-click on the plot and use the context menu to pick the second data experiment to view.

Similarly, you can perform model simulation, response prediction, and residual analysis using multi-experiment data. Here is an example of using multi-experiment timetable data for model simulation.

```matlab
sim(nlsys,MultiExpData)
```
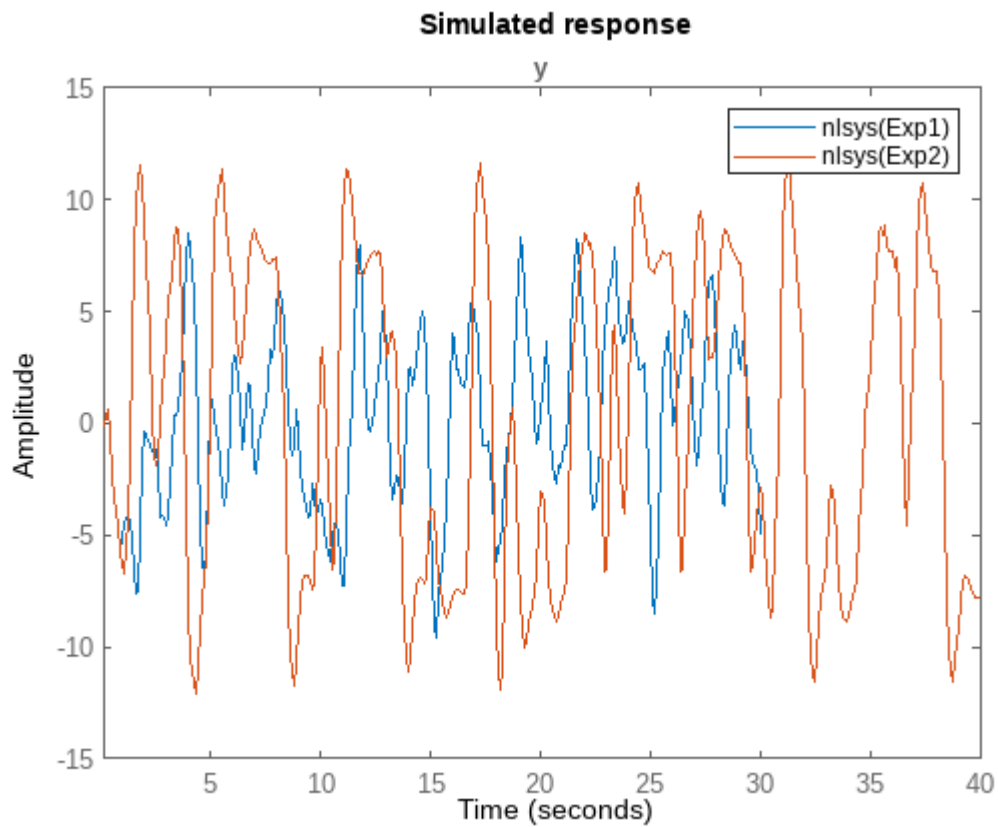
Here is another example of using multi-experiment data for spectral analysis, that is, generating a non-parametric estimate of the model frequency response.

```
% Example: Using SPA for spectral analysis.
G = spa(MultiExpData);
% view result
bode(G)
```

Bode Diagram

From: u To: y
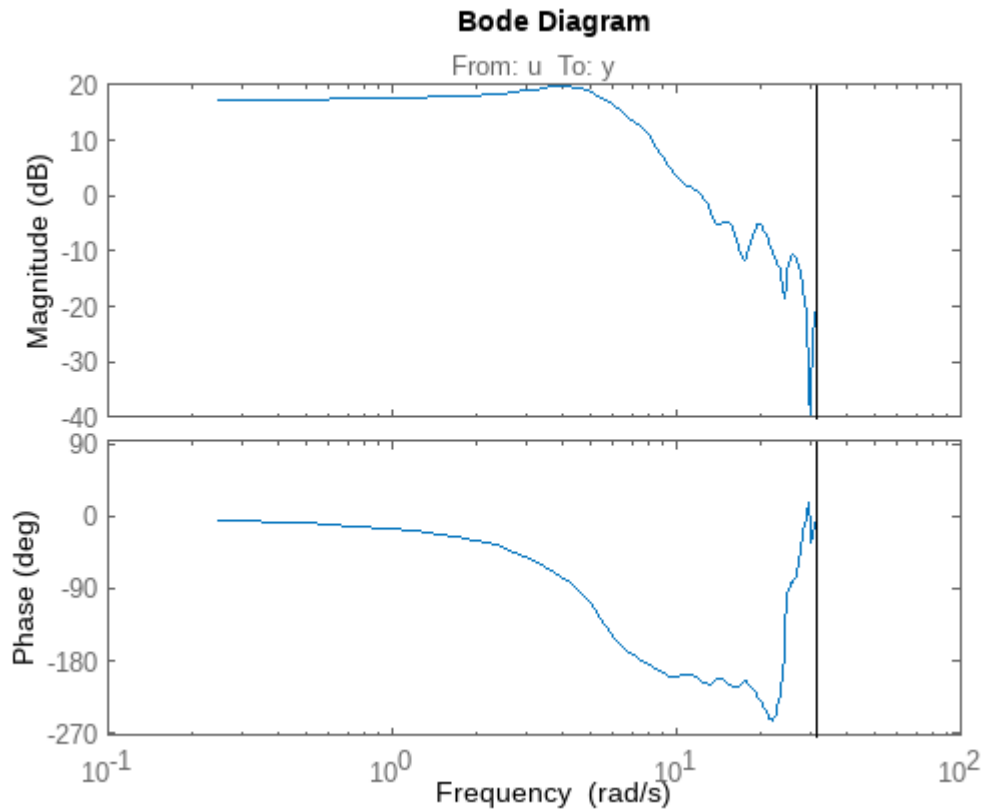
Relationship of Timetables to the IDDATA object

The timetable object is a basic datatype available in MATLAB and is the recommended way of representing time-domain signals. The iddata object is specific to System Identification Toolbox. It can be used to store both time-domain signals as well frequency-domain signals. Given a timetable with variables `'u1'`, `'u2'`, `'u3'` for inputs, and the variables `'y1'`, `'y2'` for the outputs, you can convert the data into an equivalent `iddata` object as follows:

1 Extract the time vector T from the timetable and convert it into the numeric form.
2 Extract all the input variables (`'u1'`, `'u2'`, `'u3'`) and combine them into a single input matrix U.
3 Extract all the output variables (`'y1'`, `'y2'`) and combine them into a single output matrix Y.
4 Optionally, also fetch the names and units of these variables from the timetable's "Properties".
5 Call the iddata constructor, using Y, U and T as the primary input arguments. You can optionally specify the names and/or units of the I/O channels using name-value pairs in the call to the constructor (or by dot-assignment post construction)

Here is an example.

```
% Generate some data
u1 = rand(100,1);
u2 = cumsum(randn(100,1));
u3 = -rand(100,1);
y1 = sin(u1+u2);
y2 = rand(100,1);
T = hours(0:99)';
```

```
% Pack the data into a timetable
TT = timetable(T,u1,u2,u3,y1,y2,'VariableNames',{'u1','u2','u3','y1','y2'});
TT.Properties.VariableUnits = {'a','b','c','d','e'};

% Fetch the inputs from the timetable TT
U = TT{:,1:3}; % fetch by index here (could also fetch by names)
Y = TT{:,4:5};
T = TT.Properties.RowTimes; % T is a duration vector
T_numeric = hours(T); % convert the duration vector into doubles

% Compose the iddata object
z = iddata(Y, U, 'SamplingInstants', T_numeric, 'TimeUnit','hours');
z.InputName = TT.Properties.VariableNames(1:3)';
z.OutputName = TT.Properties.VariableNames(4:5)';

% do this only if TT has units available (non-empty)
z.InputUnit = TT.Properties.VariableUnits(1:3)';
z.OutputUnit = TT.Properties.VariableUnits(4:5)';

% optional, if you want to carry over any UserData content
z.UserData = TT.Properties.UserData;
display(z)

z =

Time domain data set with 100 samples.
Sample time: 1 hours

Outputs      Unit (if specified)
   y1             d
   y2             e

Inputs       Unit (if specified)
   u1             a
   u2             b
   u3             c


plot(z)
```
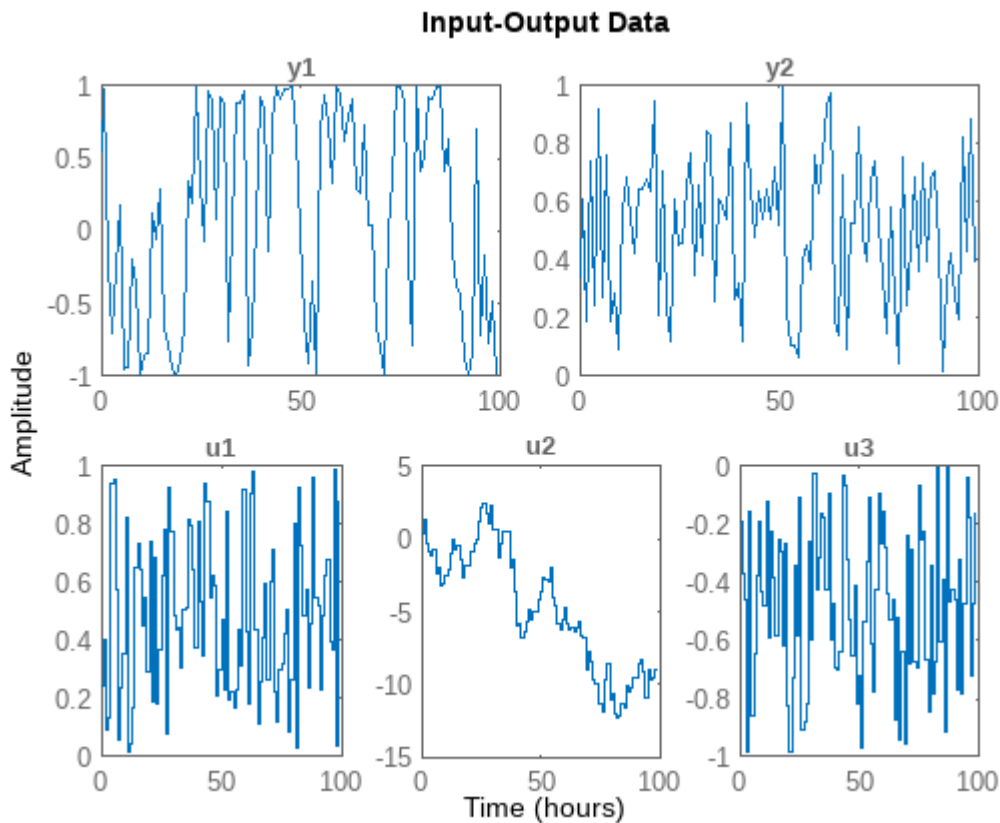
Input-Output Data

Note that for most identification and analysis purposes, the most important properties of the iddata object to assign are the input/output data and the sampling (time vector) information. The other properties are optional. However, it does help to assign the input and output names for book-keeping and for interpreting any time plots (such as plot, compare, resid, or sim).

2. Numeric Matrices

The model training, validation, and analysis commands accept numeric matrices for the input and output data. The observations are along the rows and the channels are along the columns. This is the simplest data format and is often sufficient for identifying discrete-time models. Note that this format does not provide any information about the time vector. **Hence this format is not recommended for identifying continuous-time models.**

When using the numeric (double) matrices, the input and output matrices must be specified as separate input arguments in the estimation command.

```
% load double matrices
% umat1: input vector, ymat1: output vector
load sdata1 umat1 ymat1

% Estimate a discrete-time state-space model of 4th order using the N4SID command
% Input and output data are specified using the first and second input arguments respectively
sysd = n4sid(umat1, ymat1, 4)

sysd =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + B u(t) + K e(t)
```

```
      y(t) = C x(t) + D u(t) + e(t)

  A =
            x1         x2         x3         x4
   x1     0.8392    -0.3129    0.02105    0.03743
   x2     0.4768     0.6671     0.1428   0.003757
   x3    -0.01951    0.08374   -0.09761      1.046
   x4   -0.003885   -0.02914    -0.8796   -0.03171

  B =
            u1
   x1     0.02635
   x2    -0.03301
   x3   7.256e-05
   x4   0.0005861

  C =
            x1        x2        x3        x4
   y1     69.08     26.64    -2.237   -0.5601

  D =
      u1
   y1    0

  K =
            y1
   x1     0.003282
   x2     0.009339
   x3    -0.003232
   x4     0.003809

Sample time: 1 seconds

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: estimate
   Number of free coefficients: 28
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using N4SID on time domain data "umat1".
Fit to estimation data: 76.33% (prediction focus)
FPE: 1.21, MSE: 1.087

Model Properties

sysd.Ts

ans = 1
```

Since the data does not provide the sample time, it is assumed to be 1 second. Consequently, the model sample time, `sysd.Ts`, is also 1 second (note that n4sid estimates a discrete-time model by default). Since the data does not provide the sample time information, you can choose a different sample time for the model by using the `'Ts'/<value>` name-value pair.

```
% Estimate a discrete-time state-space model of sample time 0.1 seconds
sysd = n4sid(umat1, ymat1, 4, 'Ts', 0.1);
```

In the above call, the sample time (Ts) of the model is assigned a value of 0.1 seconds. In order to achieve this, the data is first assigned a sample time of 0.1 seconds. This simple assignment of the sample time does not work if a continuous-time model is desired.

```
% Estimate a continuous-time state-space model
sysc = n4sid(umat1, ymat1, 4, 'Ts', 0);
```

Warning: Data sample time is assumed to be 1 seconds. To specify a different value for the data s

In this case, there is no direct way of specifying the data sample time; it is assumed to be 1 second. The warning issued is pointing to this fact. If you have the data in numeric format, convert it into a timetable or an iddata object wherein you can add the information regarding the sampling (such as the value of sample time and the time unit). Suppose the true sample time of the data above is 0.1 hours. In order to pass this formation to N4SID, you can proceed as follows.

```
N = size(umat1,1);
% (assume a start time of 1 hr)
t = hours(0.1*(1:N)');
dataTT = timetable(umat1,ymat1,'RowTimes',t);
sysc = n4sid(dataTT, 4, 'Ts', 0);
sysc.Ts
```

ans = 0

```
sysc.TimeUnit
```

ans =
'hours'

You can, alternatively, use an iddata object to represent the data.

```
dataIDDATA = iddata(ymat1, umat1, 0.1, 'TimeUnit', 'hours');
sysc = n4sid(dataIDDATA, 4, 'Ts', 0);
sysc.Ts
```

ans = 0

```
sysc.TimeUnit
```

ans =
'hours'

Numeric Data - Multi-experiment case

As in the case of timetables, multi-experiment data is represented by a cell array of matrices, where each cell element represents one experiment.

3. The IDDATA Object

The iddata object provides a convenient means of keeping all the data attributes together. You can store the measured signals, their sampling information, names and units in a single object. Furthermore, the iddata object can be used to represent both time- and frequency-domain signals.

So why not use this object all the time, as opposed to using double matrices or timetables? Here are the main reasons:

1   The main reason is that the iddata object is specific to the toolbox and may not be familiar to the users who have not used that this toolbox before. Requiring the data to be packaged into a

specialized object can sometimes appear burdensome. This is unlike doubles and timetables which are built-in datatypes in MATLAB and hence available for use in any toolbox.

2   Representing the data in frequency domain requires the use of iddata objects. The conversion of the time-domain data into the frequency domain is achieved by the FFT operation. Frequency domain representation can sometimes be beneficial for system identification. You can compress the data, sometimes significantly, by storing only the selected frequency samples (such as those closer to the dynamics of interest). The data can be used in dedicated frequency-domain specific identification algorithms. These algorithms can sometimes yield faster and more accurate results for linear systems.

3   Packaging all the input and output signals into an iddata object requires that all signals be measured at the same sampling instants, that is, they all share a common time vector. In use cases where the input and output signals use different time vectors, these signals need to be specified as separate input arguments in the estimation and analysis commands. Currently, on the Neural State Space models support such data.

IDDATA Object: Time-Domain Configuration

```
% Create some data.
u = sign(randn(200,2)); % 2 inputs
y = randn(200,1);       % 1 output
ts = 0.1;               % The sample time

% Create an iddata object
z = iddata(y,u,ts)

z =

Time domain data set with 200 samples.
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
   u2
```

By default, the object represents time-domain data. You can verify this by checking the `Domain` property of the object.

```
z.Domain

ans =
'Time'
```

The data is plotted as iddata by the plot command, as in:

```
% plot all the data
plot(z)
```

2-121
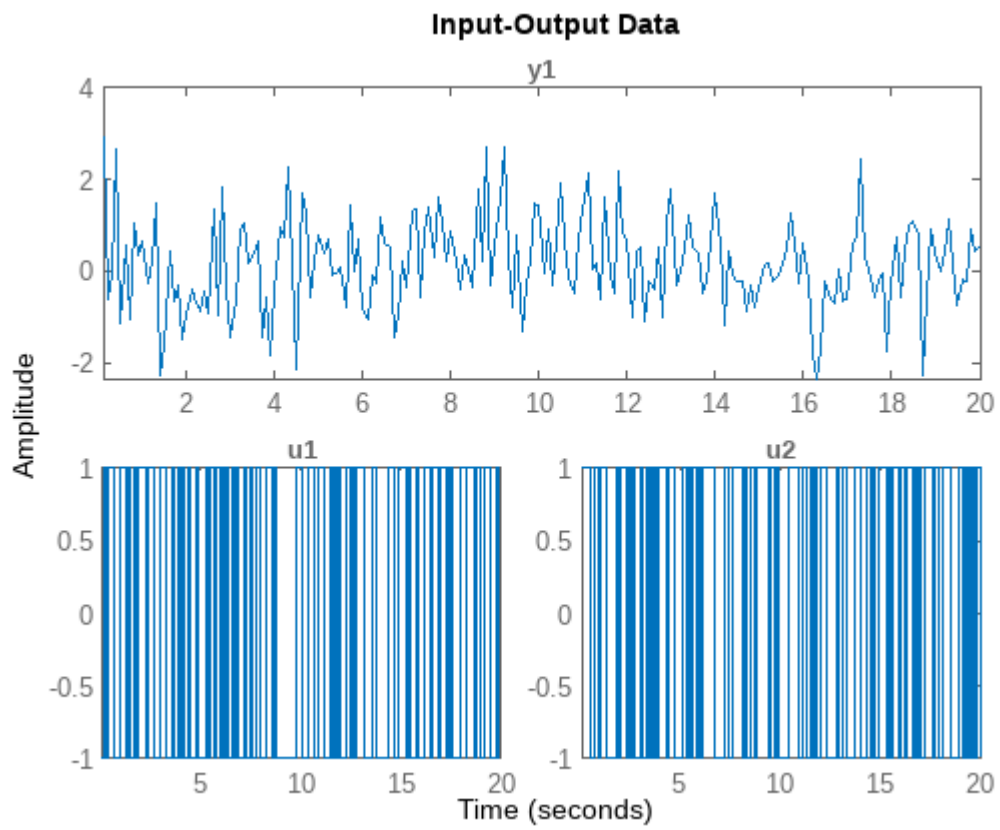
Input-Output Data

```
% Plot only the first 100 samples of the output signal and the second input signal
plot(z(1:100,:,2))
```
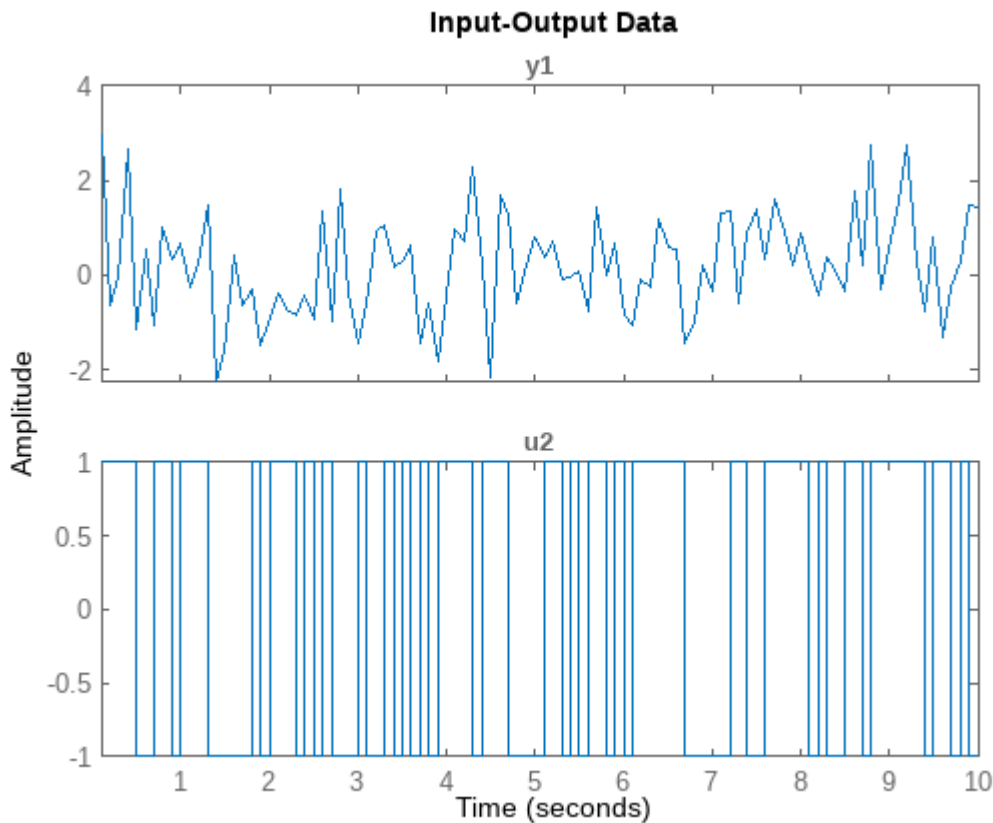
Input-Output Data

To retrieve the outputs and inputs, use

```
u = z.u;    % or, equivalently u = get(z,'u'), or u = z.InputData
y = z.y;    % or, equivalently y = get(z,'y'), or y = z.OutputData
```

To select a portion of the data:

```
zp = z(48:79); % select only the samples 48 through 79
```

To select the first output and the second input:

```
zs = z(:,1,2);   % The ':' refers to all the data time points.
```

The channels are given default names 'y1', 'u2', etc. This can be changed to any values by

```
z = set(z,'InputName',{'Voltage';'Current'},'OutputName','Speed');
```

Equivalently you can use:

```
z.InputName = {'Voltage';'Current'};
z.OutputName = 'Speed';
```

For bookkeeping and plots, also units can be set:

```
z.InputUnit = {'Volt';'Ampere'};
z.OutputUnit = 'm/s';
z
```

```
z =

Time domain data set with 200 samples.
Sample time: 0.1 seconds

Outputs        Unit (if specified)
   Speed           m/s

Inputs         Unit (if specified)
   Voltage         Volt
   Current         Ampere
```

All current properties are obtained by `get`:

```
get(z)
```

```
ans = struct with fields:
               Domain: 'Time'
                 Name: ''
           OutputData: [200×1 double]
                    y: 'Same as OutputData'
           OutputName: {'Speed'}
           OutputUnit: {'m/s'}
            InputData: [200×2 double]
                    u: 'Same as InputData'
            InputName: {2×1 cell}
            InputUnit: {2×1 cell}
               Period: [2×1 double]
          InterSample: {2×1 cell}
                   Ts: 0.1000
               Tstart: 0.1000
     SamplingInstants: [200×1 double]
             TimeUnit: 'seconds'
       ExperimentName: 'Exp1'
                Notes: {}
             UserData: []
```

You can add channels (both input and output) by "horizontal concatenation", i.e. `z = [z1 z2]`:

```
z2 = iddata(rand(200,1),ones(200,1),0.1,'OutputName','New Output',...
    'InputName','New Input');
z3 = [z,z2]

z3 =

Time domain data set with 200 samples.
Sample time: 0.1 seconds

Outputs           Unit (if specified)
   Speed              m/s
   New Output

Inputs            Unit (if specified)
   Voltage            Volt
   Current            Ampere
   New Input
```

IDDATA Object: Frequency-Domain Configuration

Frequency domain signals are represented by a set of complex input/output vectors (which represent the FFT of their time-domain counterparts) and their corresponding frequency points. Frequency-domain signals can also be the continuous-time Fourier transforms of their time-domain counterparts leading to a dataset with a sample time of zero.

Load some frequency domain signals obtained by discrete Fourier transform of certain uniformly sampled time domain signals with sample time of 0.1 seconds.

```
load('demofr','U','Y')
W = (0:500)'/100; % frequency vector in Hz
tiledlayout(2,1)
nexttile
loglog(W,abs(Y),W,abs(U))
ylabel('Amplitude')
nexttile
semilogx(W,unwrap(angle(Y))*180/pi,W,unwrap(angle(U))*180/pi)
ylabel('Phase (degrees)')
xlabel('Frequency (Hz)')
```
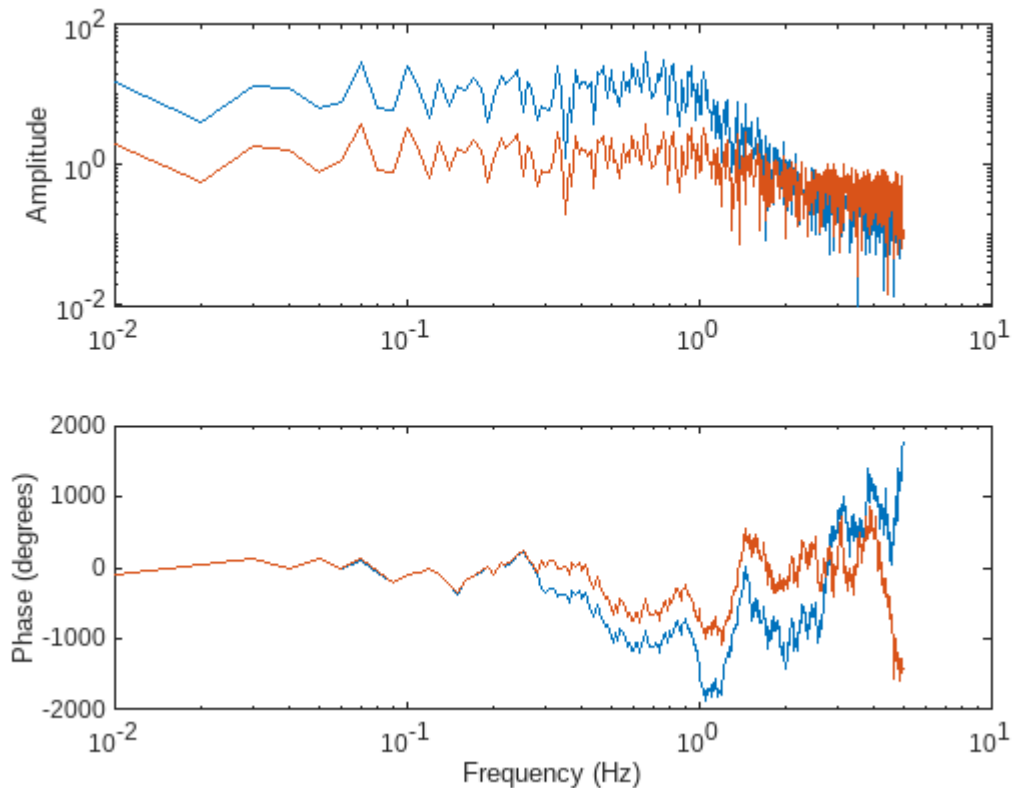

The signals U, Y, and frequency W can be put in an iddata object as follows:

```
Ts = .1; % sample time
zf = iddata(Y,U,Ts,'Frequency',W,'FrequencyUnit','Hz')
```

```
zf =
```

```
Frequency domain data set with responses at 501 frequencies.
Frequency range: 0 to 5 Hz
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

```
get(zf)
```

```
ans = struct with fields:
            Domain: 'Frequency'
              Name: ''
        OutputData: [501×1 double]
                 y: 'Same as OutputData'
        OutputName: {'y1'}
        OutputUnit: {''}
         InputData: [501×1 double]
                 u: 'Same as InputData'
         InputName: {'u1'}
         InputUnit: {''}
            Period: Inf
       InterSample: 'zoh'
                Ts: 0.1000
     FrequencyUnit: 'Hz'
         Frequency: [501×1 double]
          TimeUnit: 'seconds'
    ExperimentName: 'Exp1'
             Notes: {}
          UserData: []
```

```
zf.Domain
```

```
ans =
'Frequency'
```

The Domain property value shows that the iddata object zf represents a frequency-domain dataset. The property Frequency stores the frequency vector in the chosen frequency units. Plot the data.

```
clf
plot(zf)
```
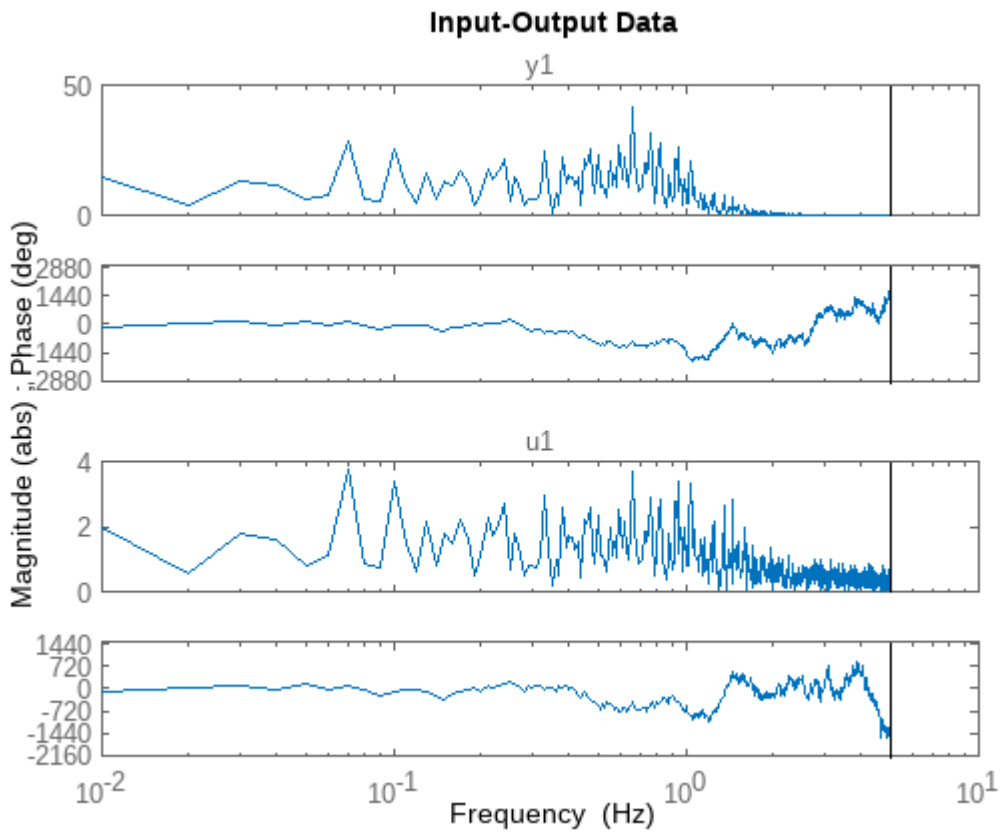
The plots shows the magnitude and the phase of the input and output frequency domain signals on separate axes.

The syntaxes for selecting a subset of samples or channels, and those for concatenation and subreferencing are same as in the time-domain case. If the data represents continuous-time Fourier transforms, use a sample time of zero, as in `zf = iddata(Y,U,0,'Frequency',W,'FrequencyUnit','Hz')`

If you are starting with time-domain signals, a convenient way of obtaining the frequency-domain representation is by using the `fft` method of the iddata object.

```
u = sign(randn(200,2)); % 2 inputs
y = randn(200,1);        % 1 output
ts = 0.1;                % The sample time

% Create time-domain iddata object
z_time = iddata(y,u,ts);
z_frequency = fft(z_time) % frequency domain representation of the dataset z

z_frequency =

Frequency domain data set with responses at 101 frequencies.
Frequency range: 0 to 31.416 rad/seconds
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1
```

```
Inputs       Unit (if specified)
   u1
   u2
```

You can convert frequency-domain dataset back into its time-domain representation using the `ifft` method, provided that the frequencies are uniformly spaced over the 0 to Nyquist frequency.

```
z_time = ifft(z_frequency)

z_time =

Time domain data set with 200 samples.
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
   u2
```

IDDATA Object: Representing Multi-Experiment Data

You can use an iddata object to store data corresponding to multiple experiments. To do so, use a cell array of signal values in the object constructor. The individual data experiments need not use the same sampling but they must all share the same input/output names and time/frequency units.

```
% Load two different
load iddata1 z1
load iddata2 z2
plot(z1,z1)
```
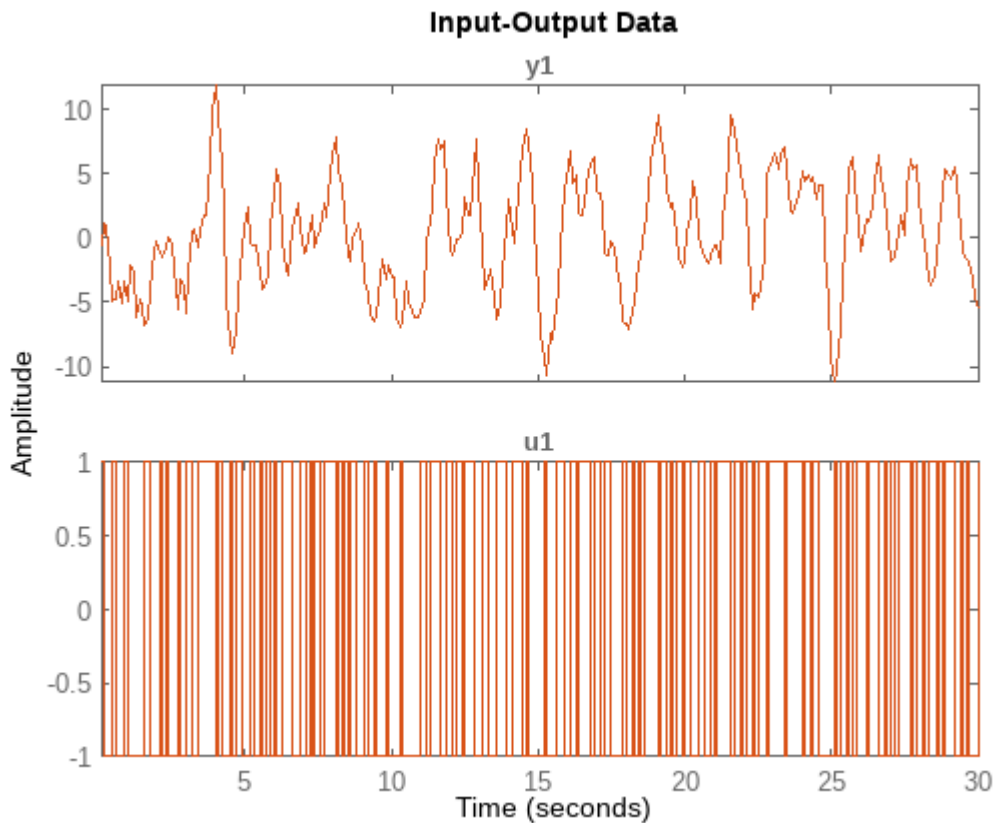
```
y1 = z1.y;
u1 = z1.u;
Ts1 = z1.Ts;
y2 = z2.y;
u2 = z2.u;
Ts2 = z2.Ts;
ycell = {y1,y2};
ucell = {u1,u2};
Tscell = {Ts1,Ts2};
zm = iddata(ycell,ucell,Tscell)
```

```
zm =
Time domain data set containing 2 experiments.

Experiment     Samples        Sample Time
   Exp1           300             0.1
   Exp2           400             0.1

Outputs        Unit (if specified)
   y1

Inputs         Unit (if specified)
   u1
```

Often, an easier way is to first create a set of single-experiment iddata objects and then combine them into a single, multi-experiment object using the merge command.

Both the datasets have the same input/output signal names and time units. Merge them into a single dataset:

```
zm = merge(z1,z2);
plot(zm)
```

Input-Output Data
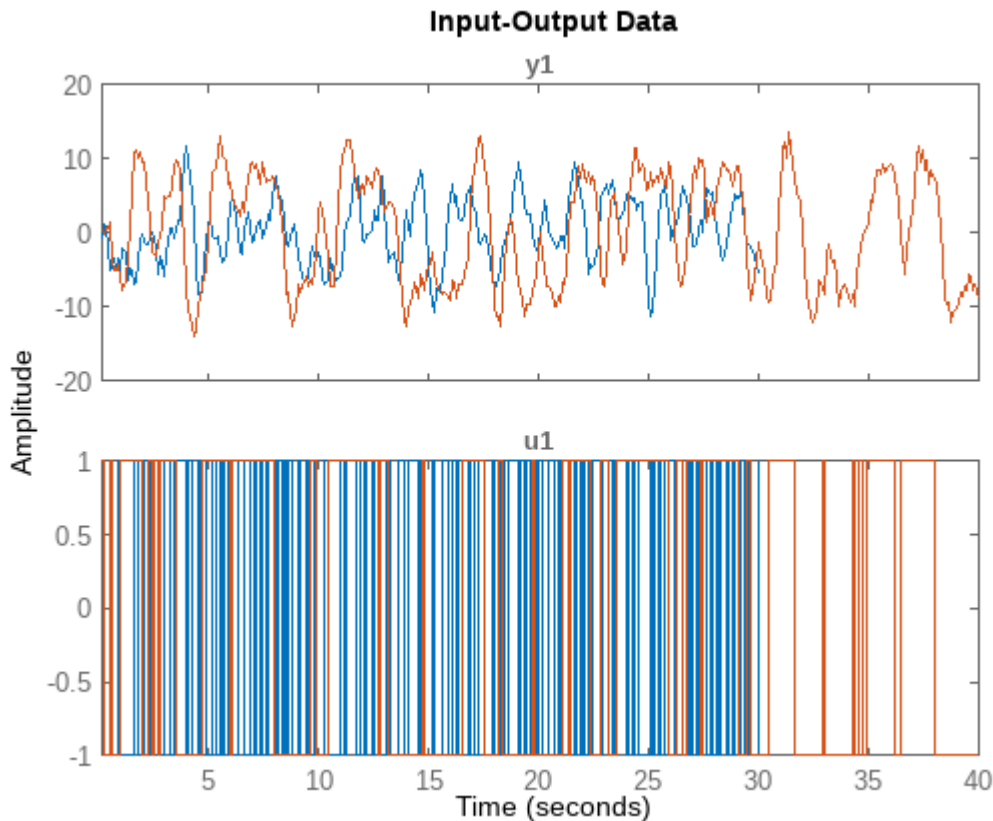
For more information, see the example *Dealing with Multi-Experiment Data and Merging Models.*

4. IDFRD Object

The IDFRD object is used to store measured frequency response data. To use it as a source of data in the various identification and analysis commands, the object must contain non-empty values in its properties that store the measured response:

- `ResponseData`, and `Frequency` properties for input/output models.
- SpectrumData and `Frequency` properties for time series models.

If you measured the frequency response data directly from hardware, such as by using a spectrum analyzer, set the data sample time to zero (`G = idfrd(response, frequency, 0)`). If the frequency response is obtained by converting from a time-domain dataset, the sample time of the idfrd object matches that of the original data. You can set the idfrd sample time to zero only if the original data used inputs that were band-limited. Note that setting the idfrd sample time to zero is desirable since it makes it possible to identify continuous-time parametric models with high speed and accuracy.

Convert Data Between Time and Frequency Domains

As described earlier, the time-domain signals can be converted into their frequency-domain counterparts by the discrete Fourier transform operation, implemented as an FFT operation. Conversion of the signal data (either time- or frequency-domain) into frequency response (FRF) is actually an exercise in estimation. There are two ways of doing so:

1 Use a non-parametric fitting method, such as etfe, spa, or spafdr. These commands produce an idfrd object as output with sample time matching that of the input data.

2 Identify an linear parametric model (e.g., using tfest, ssest, arx) and fetch the frequency response by calling the idfrd command on the result.

In either of these approaches, the input data must be uniformly sampled. See the example *Frequency Domain Identification: Estimating Models Using Frequency Domain Data* for more information on how to identify parametric models from frequency response data.

Specifying Input Intersample Behavior for Continuous-Time Modeling

An attribute of the input data that is important for continuous-time modeling is its intersample behavior, whether it is held constant between the samples (zero-order-hold), is linear (first-order-hold), or is bandlimited. Specification of this data attribute allows the estimation algorithm to account for potential aliasing effects in the data and deliver the correct result. Use the "InputInterSample" option in the estimation option set to assign the correct value.
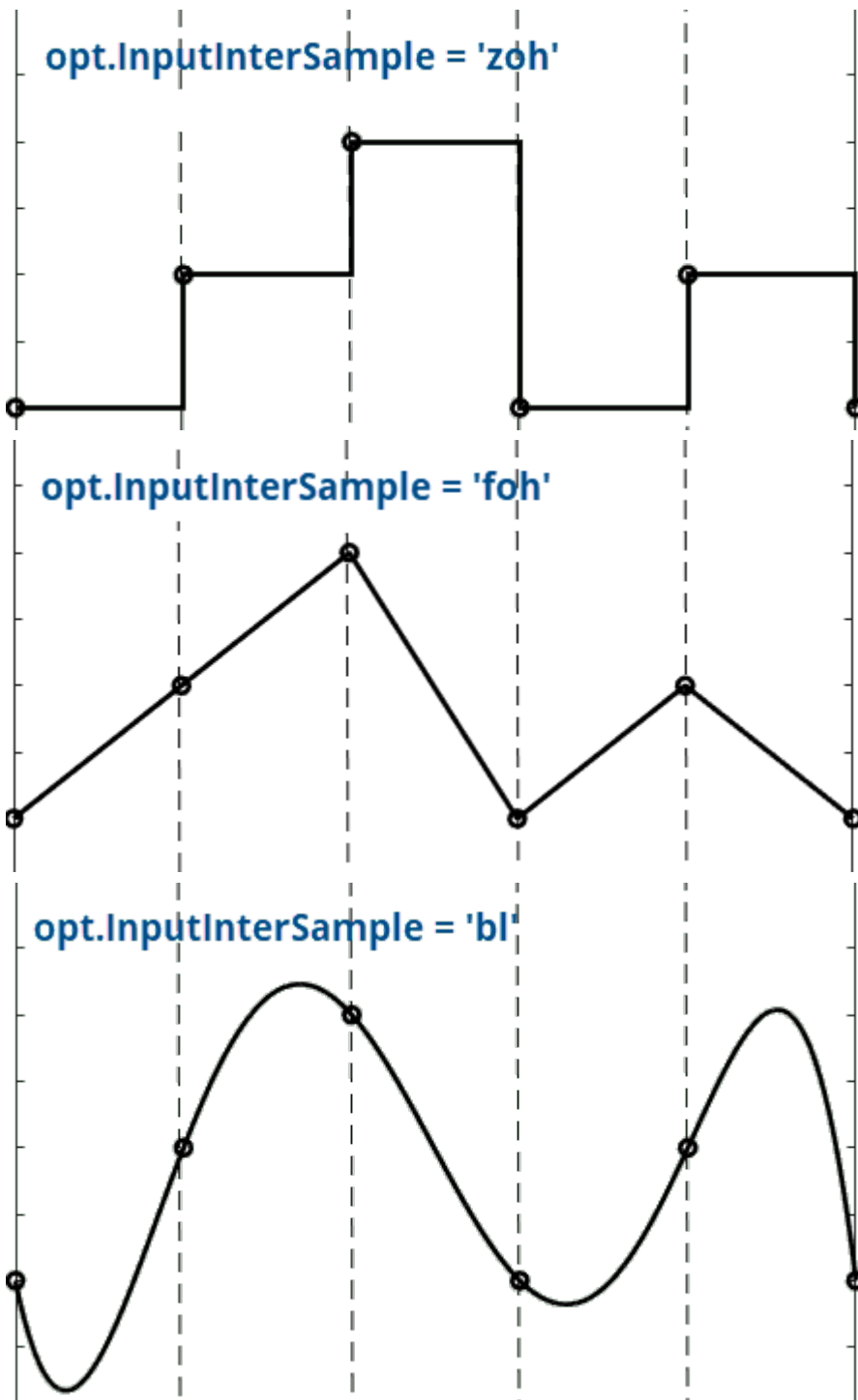
```
opt = tfestOptions;
opt.InputInterSample = 'foh';
sysTF_foh = tfest(tt8, np, nz, 'InputName',{'u1','u2','u3'},'OutputName','y',opt);
opt.InputInterSample = 'zoh';
sysTF_zoh = tfest(tt8, np, nz, 'InputName',{'u1','u2','u3'},'OutputName','y',opt);
opt.InputInterSample = 'bl';
```

```
sysTF_bl = tfest(tt8, np, nz, 'InputName',{'u1','u2','u3'},'OutputName','y',opt);
bodemag(sysTF_foh, sysTF_zoh, sysTF_bl)
```
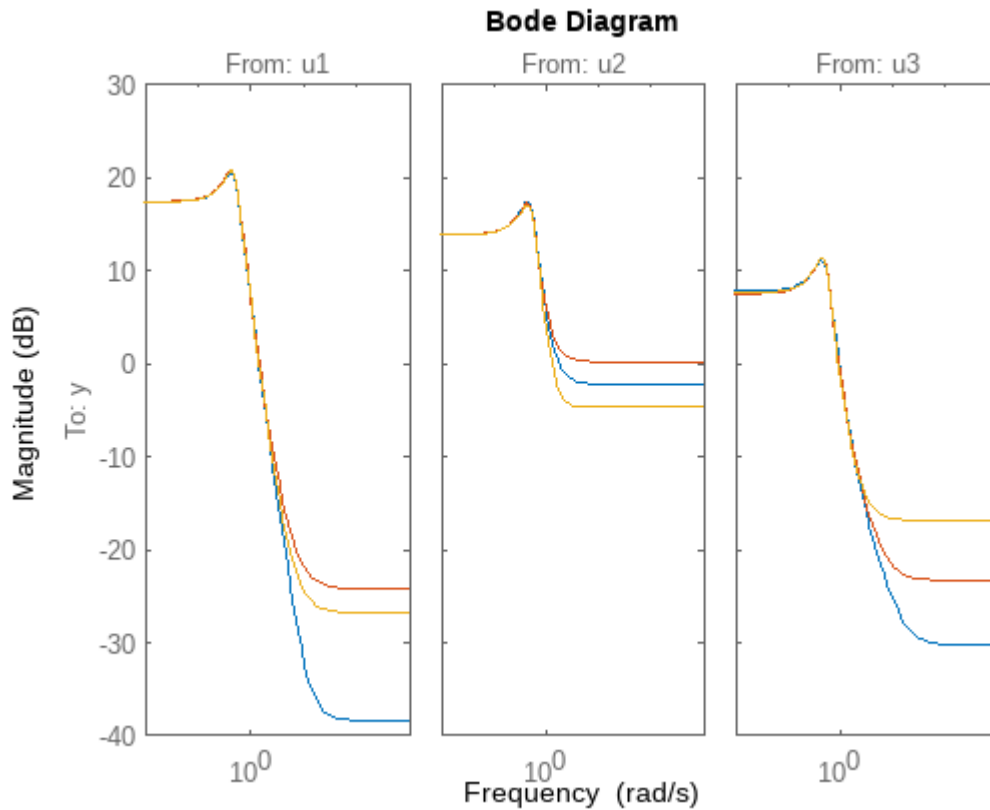

Note that the inter-sample behavior considerations only apply to models with inputs.

Time-Series and Signal Modeling

Time series, or autonomous system, models are models with no measured input signals. `size(model,2)` returns 0. Such models can be identified, often using the same commands as those used for identifying the input-output models (such as `ssest`, `arx`, `nlarx`, `greyest` etc). The toolbox also provides dedicated routines such as `ar`, `ivar` for time-series modeling.

When you are fitting a time-series model, you only provide the output signal to the estimation command. So if the data is in the form of a timetable containing several variables, you may need to instruct the estimation command that none of those variables should be treated as measured inputs. Here is how different commands handle data specified in various formats (iddata object, double matrix, or timetable). For all model types, except idNeuralStateSpace, the data must be uniformly sampled.

AR, IVAR Commands

These commands identify univariate time series models - they need only a single data vector.

Double matrix: data must be a column vector

```
% Example: fit a 5th order AR model to a moving average process
data = movmean(randn(100,1),5);
```

```
model = ar(data(1:10), 5);

% forecast response 20 steps ahead
forecast(model, data(1:10), 20)
```


20-Step Ahead Forecast
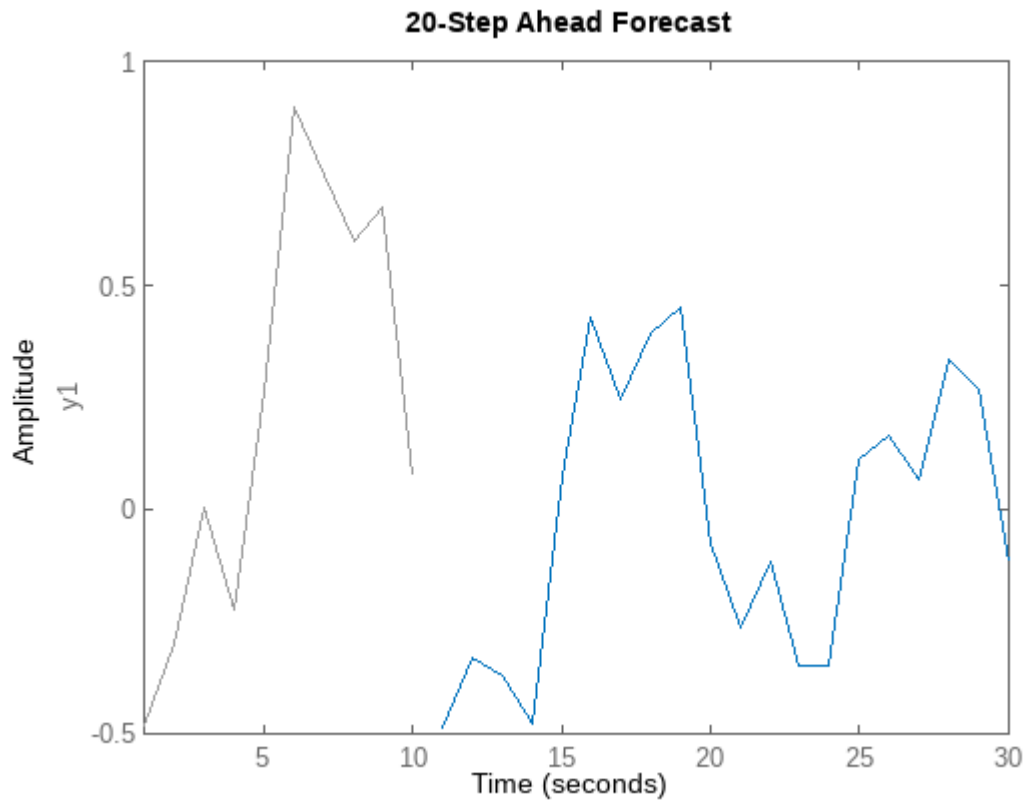
Iddata object: data must contain a single output channel and no inputs

```
t = (0:100)'/100;
data = iddata(exp(-0.2*t).*sin(2*pi*3*t), [], 0.01);
plot(data)
```

Input-Output Data
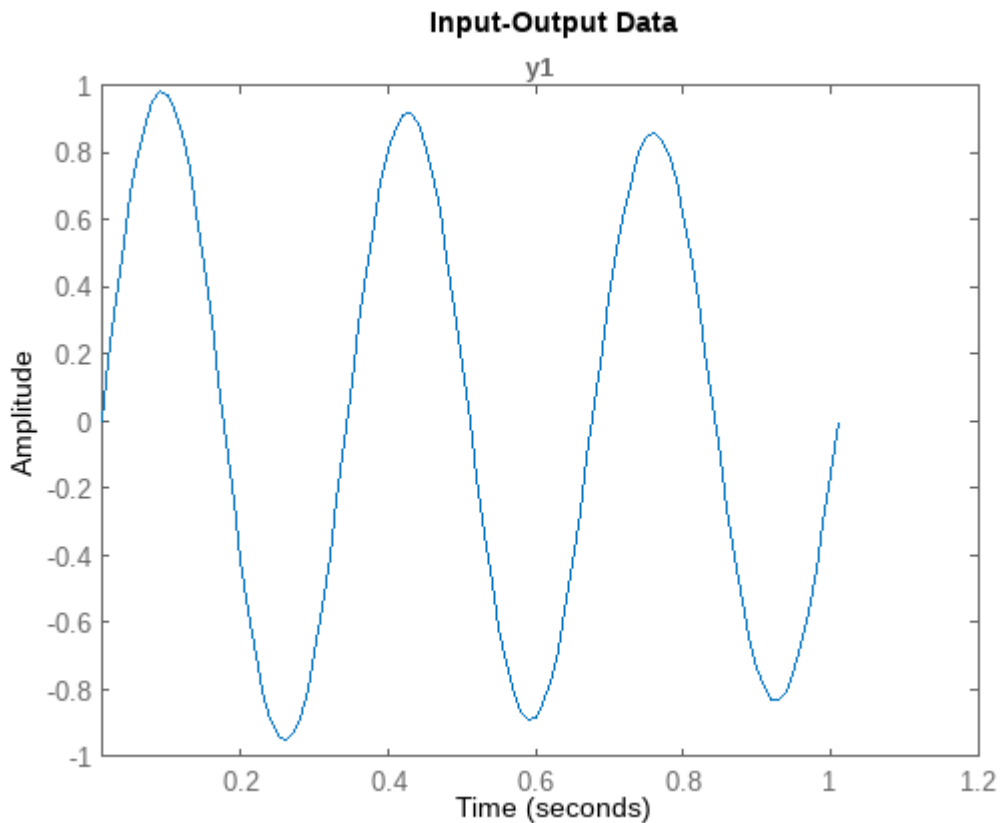

```
% fit the model using the first 50 samples of data
model = ar(data(1:50), 3)

model =
Discrete-time AR model: A(z)y(t) = e(t)
  A(z) = 1 - 2.945 z^-1 + 2.926 z^-2 - 0.9801 z^-3

Sample time: 0.01 seconds

Parameterization:
   Polynomial orders:    na=3
   Number of free coefficients: 3
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using AR ('fb/now') on time domain data.
Fit to estimation data: 99.99%
FPE: 1.005e-08, MSE: 8.915e-09

Model Properties

% forecast response 100 steps ahead, given the measurements of 50 samples.
forecast(model, data(1:50), 100)
```

100-Step Ahead Forecast
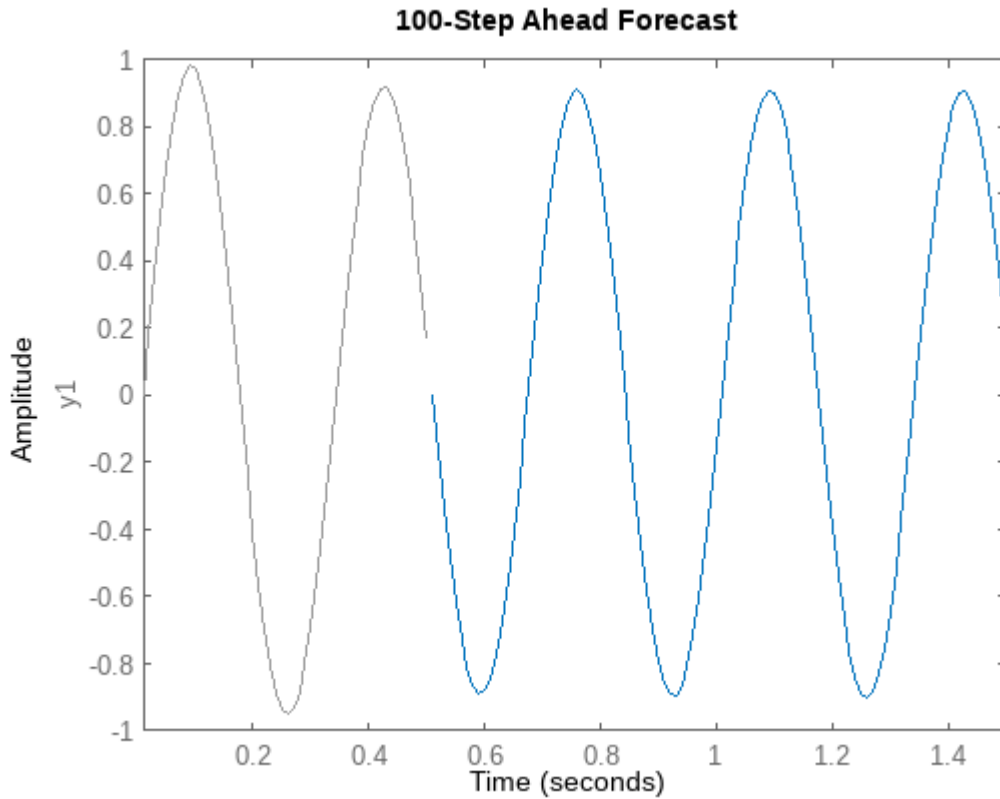
Timetable: the first variable that contains numeric, scalar value (i.e. scalar value in one row) is used. All others are ignored.

```
t = (0:100)'/100;
x1 = randn(length(t),2); % some other vector valued data
x2 = rand(101,1);
x2 = discretize(x2,[0 .25 .75 1],'categorical',{'small', 'medium', 'large'});
x3 = exp(-0.2*t).*sin(2*pi*3*t); % signal of interest
x4 = exp(-0.4*t).*sin(2*pi*6*t+1); % some other valid signal

T = timetable(minutes(t),x1,x2,x3,x4);
head(T)
```

Time	x1		x2	x3	x4
0 min	0.75911	-1.029	medium	0	0.84147
0.01 min	-0.081664	0.20651	medium	0.18701	0.97736
0.02 min	-0.15847	1.3411	large	0.36665	0.97543
0.03 min	0.006498	1.3327	medium	0.53262	0.83706
0.04 min	-1.0646	-1.2849	medium	0.67909	0.58267
0.05 min	-1.6439	1.6184	medium	0.80097	0.2488
0.06 min	1.5815	0.66164	large	0.89403	-0.11722
0.07 min	0.29261	0.22735	medium	0.95512	-0.46392

The timetable T contains 3 variables - x1, x2, x3 and x4. x1 is not a valid signal for identification since its value at a given time instant is a 1-by-2 vector. SImilarly, x2 is not valid since its value is

categorical - we need doubles. `x3` and `x4` are valid signals. What happens when you use T as data in the `ar` or the `ivar` command?

```
% fit the 3rd order auto-regressive model using the first 50 rows of T
model1 = ar(T(1:50,:), 3)

Warning: The timetable variable "x1" cannot be used as a signal because it contains multi-column

Warning: The timetable variable "x2" cannot be used as a signal because it contains categorical o

model1 =
Discrete-time AR model: A(z)y(t) = e(t)
  A(z) = 1 - 2.945 z^-1 + 2.926 z^-2 - 0.9801 z^-3

Sample time: 0.6 seconds

Parameterization:
   Polynomial orders:    na=3
   Number of free coefficients: 3
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using AR ('fb/now') on time domain data.
Fit to estimation data: 99.99%
FPE: 1.005e-08, MSE: 8.915e-09

Model Properties

model1.OutputName

ans = 1×1 cell array
    {'x3'}
```

The model used the variable `x3` as the data to fit. This is because it is the first available valid signal in the timetable. What if you wanted to fit the model to the signal `x4` instead? To make that happen, you will need to specify 'OutputName' as name-value pair.

```
model2 = ar(T(1:50,:), 5, 'OutputName', 'x4')

model2 =
Discrete-time AR model: A(z)y(t) = e(t)

A(z) = 1 - 2.911 z^-1 + 2.453 z^-2 + 0.6907 z^-3 - 2.005 z^-4 + 0.808 z^-5

Sample time: 0.6 seconds

Parameterization:
   Polynomial orders:    na=5
   Number of free coefficients: 5
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using AR ('fb/now') on time domain data.
Fit to estimation data: 100%
FPE: 4.61e-30, MSE: 3.772e-30

Model Properties
```

2-137

```
model2.OutputName
```

ans = 1×1 cell array
    {'x4'}

Validate results and forecast

`% how well can the two models predict their respective signals in the timetable with 100-step pre`
`compare(T, model1, model2, 100)`
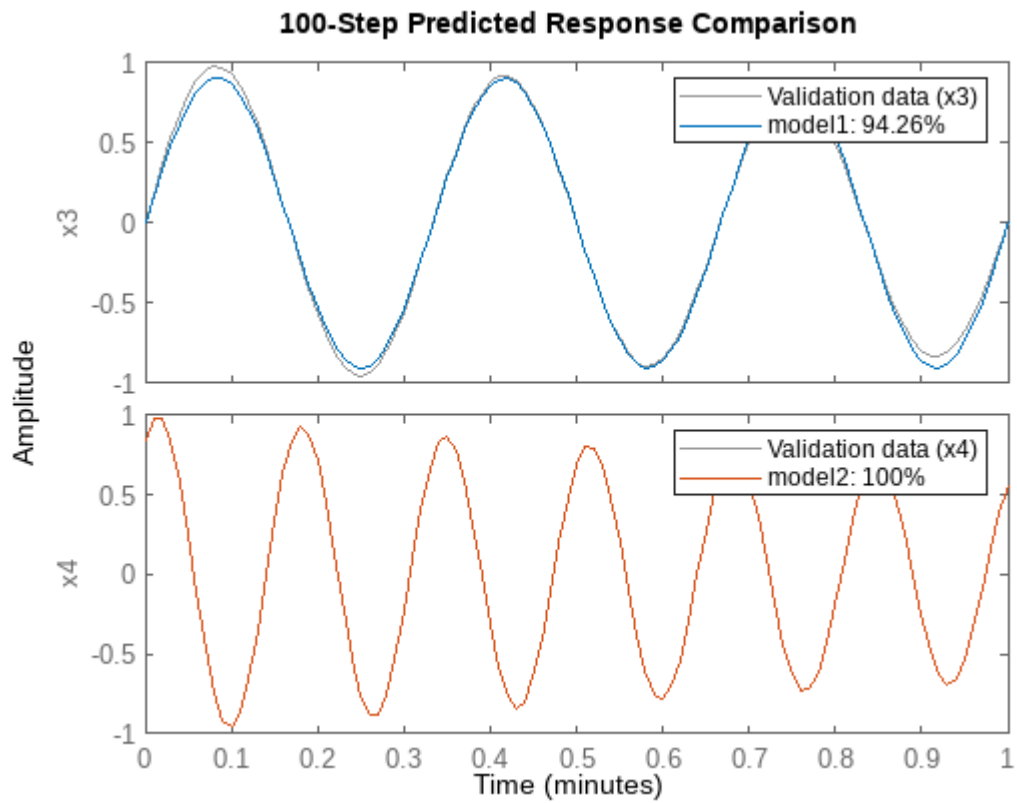
`% evaluate model residuals`
`resid(T, model1, model2)`
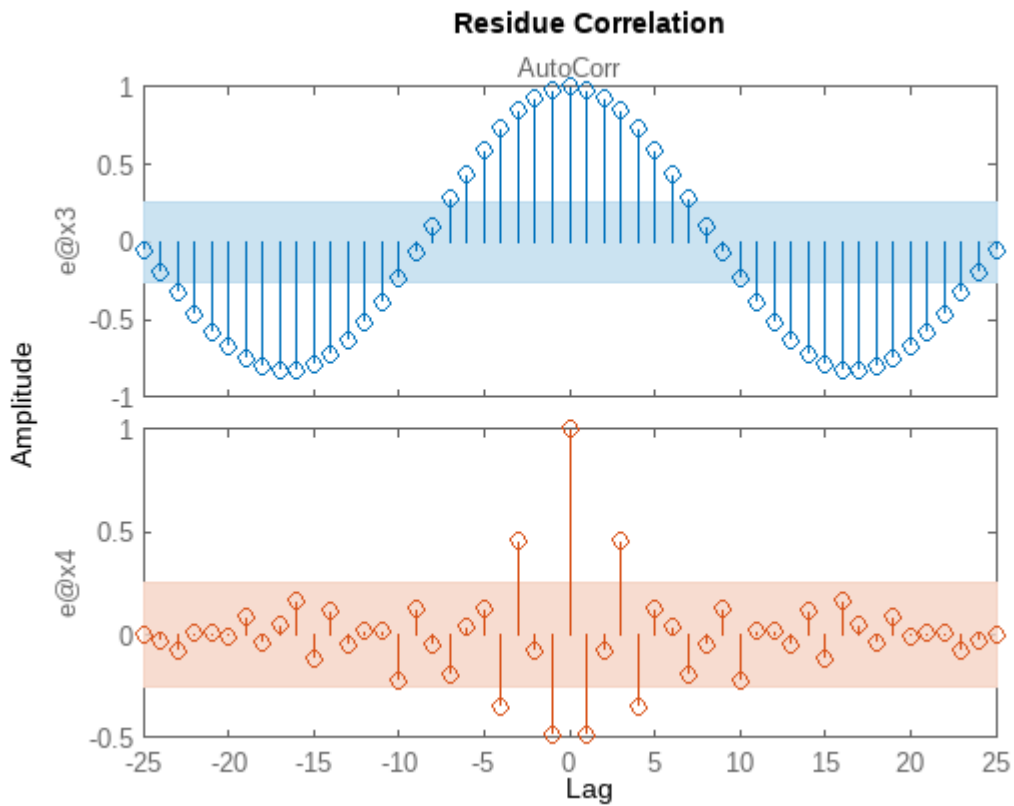
The models are able to predict outcomes 100 steps ahead but the model residues (with 1-step horizon) are not uncorrelated. This hints at using a higher model order or tweaking estimation algorithm choices. Finally, forecast the respective signals 100 steps ahead.

```
forecast(model1, model2, T, 100)
```
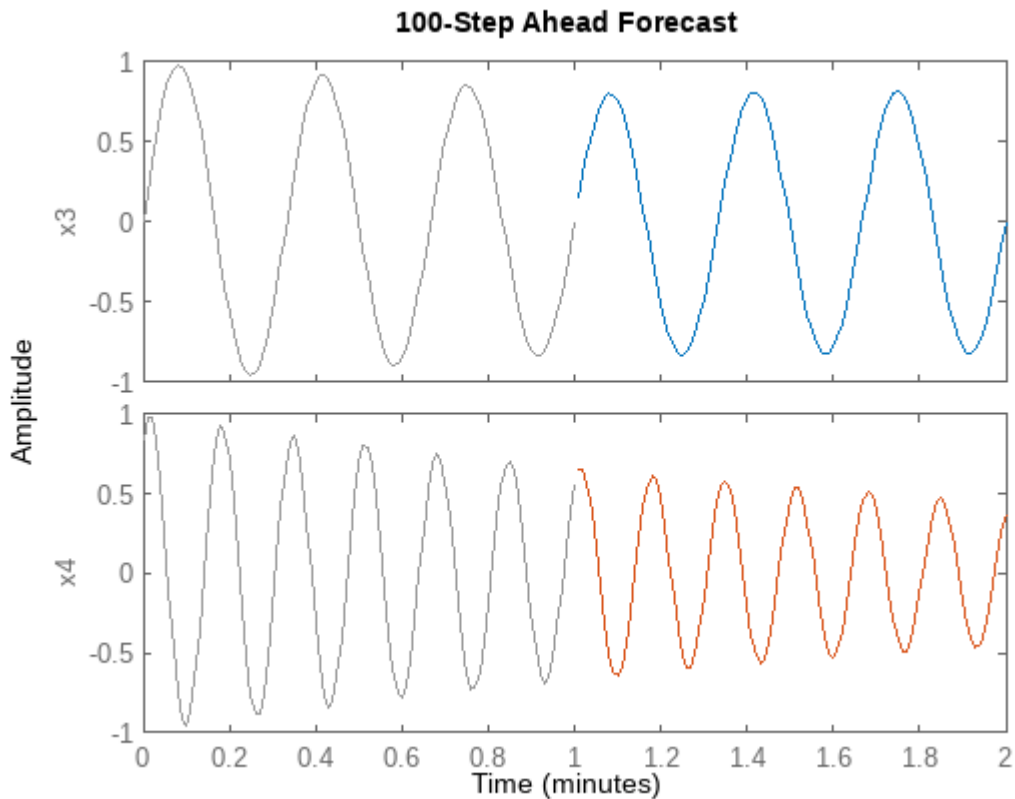
State Space Time Series Models

State space models are syntactically the simplest ones to estimate since, in the simplest case, they require just the data and a rough guess about the model order. The commands `ssest, n4sid, era,` and `ssregest` can be used to identify a time series model in the state space form. These commands can identify multivariate models.

Using double matrices: leave the first input argument empty

When using double data, the input and the output data must be specified using separate input arguments, as in `ssest(input, output, model_order)`. For time series modeling, you must use `input = []`.

```
t = (0:100)'/100;
x1 = exp(-0.2*t).*sin(2*pi*3*t);
x2 = exp(-0.4*t).*sin(2*pi*6*t+1);
output = [x1,x2];

% fit a 6th order disrete-time state space model
model_dt = ssest([],output,'best','Ts',0.01) % automatically pick the best order in the 1:10 rang

model_dt =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + K e(t)
        y(t) = C x(t) + e(t)
```

```
A =
            x1         x2         x3         x4
  x1     0.9497     0.2879    0.07526    0.03245
  x2    -0.3003     0.9329   -0.02131     0.2194
  x3   -0.08673    0.01416     0.9771     -0.162
  x4    0.03299    -0.2092     0.1872      0.953

C =
            x1         x2         x3         x4
  y1       2.95      0.409      2.241      3.735
  y2     -3.553    -0.9476    -0.3008      3.071

K =
            y1         y2
  x1    0.05216   -0.06347
  x2   0.002149   -0.03596
  x3    0.04893   -0.01924
  x4    0.02691     0.0441

Sample time: 0.01 seconds

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Disturbance component: estimate
   Number of free coefficients: 32
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on time domain data.
Fit to estimation data: [100;100]% (prediction focus)
FPE: 2.437e-60, MSE: 2.616e-30

Model Properties
```

```matlab
forecast(model_dt,output,100) % 100-step ahead response forecast
```

100-Step Ahead Forecast
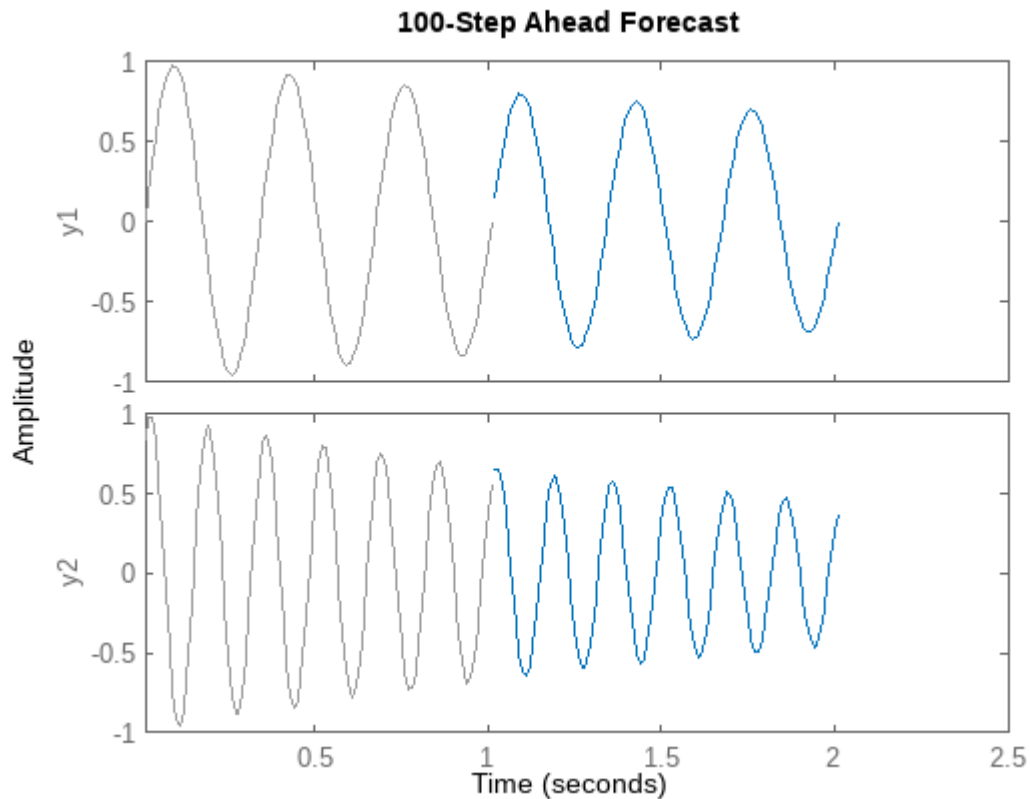
As noted before, the use of double data for identifying continuous-time models is not recommended. You must use iddata, or timetable format of the data.

Using iddata object: Use an object with nu=0 input channels and ny>=1 output channels.

The model is fit to all the available output channels in the data object.

```
dataobj = iddata(output, [], 0.01, 'Tstart', 0);
% identify a continuous-time model
model_ct = ssest(dataobj,'best'); % automatically pick the best order in the 1:10 range
forecast(model_ct,output,100) % 100-step ahead response forecast
```

100-Step Ahead Forecast
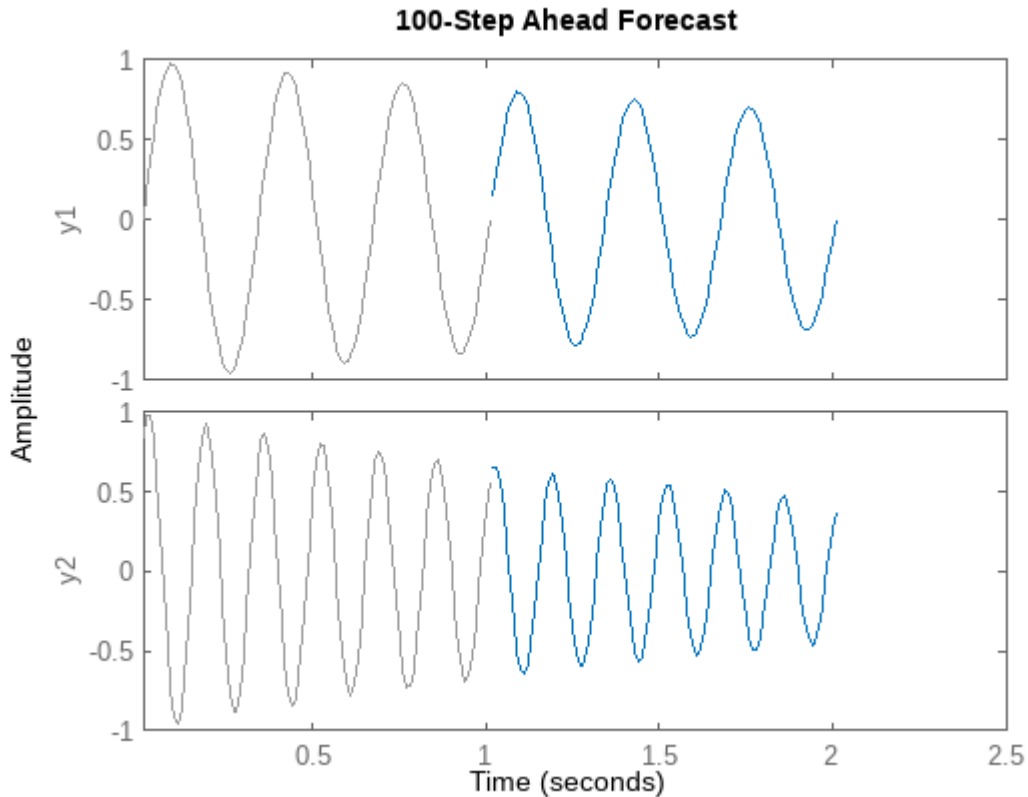
Using timetables: must use `'InputName'`, `'OutputName'` name-value pair if the number of variables is greater than 1

If the timetable contains N>1 valid variables, the ssest command (similarly the n4sid or ssregest command) treats the first N-1 variables as inputs and the last one as the output resulting in an input-output model (models with 1 or more inputs) rather than a multivariate time series model. In order to treat all the valid variables as time series data, you must designate them as "outputs" using the 'OutputName'/Value name-value pair, where the Value is a cell array containing the names of all the valid variables in the timetable. Alternatively, use 'InputName'/[] to indicate that all the valid variables are outputs.

```
% supress the warning about the table containing invalid variables
Warn = warning;
warning('off','Ident:utility:timetableDataMultiCol');
warning('off','Ident:utility:timetableDataTypeNotAcceptable');
Restore = onCleanup(@()warning(Warn));

% create some tabular data
t = (0:100)'/100;
x1 = randn(length(t),2); % some other vector valued data
x2 = cumsum(t);
x3 = rand(101,1);
x3 = discretize(x3,[0 .25 .75 1],'categorical',{'small', 'medium', 'large'});
x4 = exp(-0.2*t).*sin(2*pi*3*t); % signal of interest
x5 = exp(-0.4*t).*sin(2*pi*6*t+1); % some other valid signal
T = timetable(minutes(t),x1,x2,x3,x4,x5);
head(T)
```

Time	x1	x2	x3	x4	x5
0 min	-0.83494	-0.75297	0	medium	0	0.84147
0.01 min	0.17968	-1.0331	0.01	medium	0.18701	0.97736
0.02 min	1.1663	1.1638	0.03	medium	0.36665	0.97543
0.03 min	0.055986	-0.58011	0.06	large	0.53262	0.83706
0.04 min	-2.1	0.4173	0.1	medium	0.67909	0.58267
0.05 min	1.2353	-1.6481	0.15	small	0.80097	0.2488
0.06 min	0.0026715	-0.57268	0.21	medium	0.89403	-0.11722
0.07 min	-0.45954	0.65923	0.28	medium	0.95512	-0.46392

```
% x2, x4 and x5 are valid signals
% Identify a multivariate time series models for {'x2','x4','x5'} signals
sys1 = n4sid(T,7,'OutputName',{'x2';'x4';'x5'})

sys1 =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + K e(t)
       y(t) = C x(t) + e(t)

  A =
            x1        x2        x3        x4        x5        x6        x7
    x1     120.1    -145.7     4.411    0.1017    -23.21     63.04      -166
    x2    -59.81     74.15    -2.315  -0.03415     11.63    -31.68     83.35
    x3     1.253    -1.449     1.012   0.03292   -0.1766    0.8812    -1.753
    x4    -4.391     5.384   -0.2207    0.9264     1.148    -2.222     6.122
    x5     -10.4     12.76   -0.4461   -0.3291     2.968    -5.469     14.49
    x6     32.03    -39.11    0.9799  -0.08308     -6.33     17.91     -44.6
    x7     151.6    -185.4     5.633    0.1183    -29.55     80.25    -210.3

  C =
            x1        x2        x3        x4        x5        x6        x7
    x2    -124.3     11.61      0.63     -0.45   -0.5675     2.689     124.9
    x4   -0.1134     1.798     1.282   0.09119    -2.941     4.024   -0.2603
    x5   -0.1976    0.9301   -0.5758     -1.33     3.912      2.63    0.2018

  K =
            x2        x4        x5
    x1     0.0668    -6.652    -1.338
    x2   -0.03393     3.275    0.6335
    x3  0.0006712  -0.05902  -0.006004
    x4   -0.00248    0.2269    0.04379
    x5  -0.005874    0.5324    0.1649
    x6    0.01811    -1.677   -0.3007
    x7    0.08523     -8.41    -1.708

Sample time: 0.6 seconds

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Disturbance component: estimate
   Number of free coefficients: 91
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using N4SID on time domain data "T".
```

```
Fit to estimation data: [100;100;100]% (prediction focus)
FPE: 1.043e-40, MSE: 1.528e-10

Model Properties
```

size(sys1)

Identified state-space model with 3 outputs, 0 inputs, 7 states and 91 free parameters.

The outputs of model sys1 are 'x2', 'x4', 'x5'. There are no inputs since the timetable T does not contain any other valid variables. Alternatively:

sys2 = n4sid(T,7,'InputName',[])

```
sys2 =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + K e(t)
       y(t) = C x(t) + e(t)

  A =
             x1        x2        x3        x4        x5        x6        x7
   x1      120.1    -145.7     4.411    0.1017    -23.21     63.04      -166
   x2     -59.81     74.15    -2.315  -0.03415     11.63    -31.68     83.35
   x3      1.253    -1.449     1.012   0.03292   -0.1766    0.8812    -1.753
   x4     -4.391     5.384   -0.2207    0.9264     1.148    -2.222     6.122
   x5      -10.4     12.76   -0.4461   -0.3291     2.968    -5.469     14.49
   x6      32.03    -39.11    0.9799  -0.08308     -6.33     17.91     -44.6
   x7      151.6    -185.4     5.633    0.1183    -29.55     80.25    -210.3

  C =
             x1        x2        x3        x4        x5        x6        x7
   x2     -124.3     11.61      0.63     -0.45   -0.5675     2.689     124.9
   x4    -0.1134     1.798     1.282   0.09119    -2.941     4.024   -0.2603
   x5    -0.1976    0.9301   -0.5758     -1.33     3.912      2.63    0.2018

  K =
             x2        x4        x5
   x1     0.0668    -6.652    -1.338
   x2   -0.03393     3.275    0.6335
   x3  0.0006712  -0.05902  -0.006004
   x4   -0.00248    0.2269   0.04379
   x5  -0.005874    0.5324    0.1649
   x6    0.01811    -1.677   -0.3007
   x7    0.08523     -8.41    -1.708

Sample time: 0.6 seconds

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Disturbance component: estimate
   Number of free coefficients: 91
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using N4SID on time domain data "T".
Fit to estimation data: [100;100;100]% (prediction focus)
FPE: 1.043e-40, MSE: 1.528e-10

Model Properties
```

```
size(sys2)
```

```
Identified state-space model with 3 outputs, 0 inputs, 7 states and 91 free parameters.
```

What if you wanted to fit only the signals `'x4'` and `'x5'`, ignoring the rest? You need to specify both 'InputName' and 'OutputName' using name-value pairs.

```
% fit a bivariate model to x4, x5 signals where the model order is determined automatically
sys = n4sid(T,'best','InputName',[],'OutputName',{'x4';'x5'})
```

```
sys =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + K e(t)
        y(t) = C x(t) + e(t)

  A =
             x1        x2        x3        x4
    x1     0.9497    0.2879   0.07526   0.03245
    x2    -0.3003    0.9329  -0.02131   0.2194
    x3   -0.08673   0.01416    0.9771    -0.162
    x4    0.03299   -0.2092    0.1872     0.953

  C =
             x1        x2        x3        x4
    x4      2.95     0.409     2.241     3.735
    x5    -3.553   -0.9476   -0.3008     3.071

  K =
             x4        x5
    x1    0.05216  -0.06347
    x2   0.002149  -0.03596
    x3    0.04893  -0.01924
    x4    0.02691    0.0441

Sample time: 0.6 seconds

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Disturbance component: estimate
   Number of free coefficients: 32
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using N4SID on time domain data "T".
Fit to estimation data: [100;100]% (prediction focus)
FPE: 2.437e-60, MSE: 2.616e-30

Model Properties
```

```
forecast(sys,T,100) % 100 step ahead forecast
```

100-Step Ahead Forecast
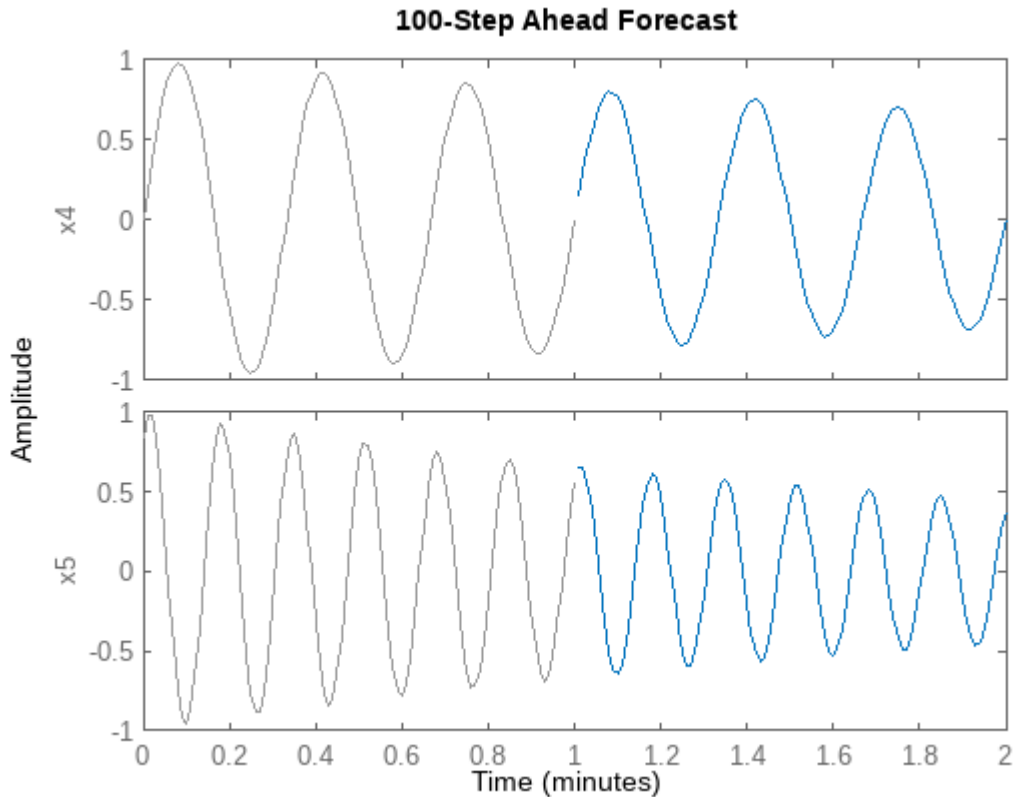
Polynomial Time Series Models

The polynomial time series models take the form $A(q)y(t) = C(q)/D(q)e(t)$, where $y(t)$ is the measured signal (can be multivariate), $e(t)$ is unmeasured disturbance, and `A(q)`, `C(q)`, `D(q)` are polynomials expressed in the lag operator `q^-1`. They can be identified using commands such as `ar, ivar, arx, bj, armax, polyest`.In the nonlinear case, the equation takes the form $y(t) = f(y(t-1), y(t-2), ..., y(t-n)) + e(t)$, where `f(.)` is a nonlinear fucntion. Nonlinear time series models can be identified using the `nlarx` command. The data use in the `ar` and `ivar` commands was described above. For arx, armax, bj, and polyest commands, the number of inputs and the outputs used by the model are uniquely determined by the order matrix `NN`. The number of rows of `NN` equals the number of outputs (`ny = size(NN,1)`). The number of inputs nu can be determined by the number of columns in the order variable `nb` (`nu = size(nb,2)`). For example, `NN = [na nb nc nk]` for the `armax` command. `ny = size(NN,1), nu = size(nb,2)`.

In the timeseries case, `nu = 0`.

- ARX: NN = `na` (ny-by-ny matrix)
- ARMAX: NN = `[na nc]` (ny-by-(ny+1) matrix)
- BJ: NN = `[nc nd]` (ny-by-2) matrix
- POLYEST: NN = `[na nc nd]` (ny-by-(ny+2)) matrix, either na or nd must be zero

Using double matrices

Specify the input and output matrices as separate input arguments to the estimation command. Input data matrix must be empty.

```
t = (0:100)'/100;
x1 = exp(-0.2*t).*sin(2*pi*3*t);
x2 = exp(-0.4*t).*sin(2*pi*6*t+1);
output = [x1,x2];

% fit a 6th order disrete-time state space model
na = [3 1; 0 3]; % (the off-diagonal values could have been chosen to be zero since x1 and x2 are
model_arx = arx([],output,na);
present(model_arx)


model_arx =
Discrete-time AR model:
  Model for output "y1": A(z)y_1(t) = - A_i(z)y_i(t) + e_1(t)

    A(z) = 1 + 0 (+/- 0.1951) z^-1 - 2.848 (+/- 0.3825) z^-2 + 1.953 (
                                                      +/- 0.1943) z^-3

    A_2(z) = 2.166e-16 (+/- 2.843e-16) z^-1

  Model for output "y2": A(z)y_2(t) = e_2(t)

    A(z) = 1 - 1.317 (+/- 0.08134) z^-1 + 0 (+/- 0.1507) z^-2 + 0.5313 (
                                                      +/- 0.08069) z^-3

Sample time: 1 seconds

Parameterization:
   Polynomial orders:   na=[3 1;0 3]
   Number of free coefficients: 7
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using ARX on time domain data.
Fit to estimation data: [100;100]% (prediction focus)
FPE: 4.269e-60, MSE: 3.851e-30
More information in model's "Report" property.

Model Properties
```

Using Timetables

Unless name-value pairs are used, the first ny valid variables in the timetable are used as outputs.

```
t = (0:100)'/100;
x1 = randn(length(t),2); % some other vector valued data
x2 = cumsum(t);
x3 = rand(101,1);
x3 = discretize(x3,[0 .25 .75 1],'categorical',{'small', 'medium', 'large'});
x4 = exp(-0.2*t).*sin(2*pi*3*t); % signal of interest
x5 = exp(-0.4*t).*sin(2*pi*6*t+1); % some other valid signal
T = timetable(minutes(t),x1,x2,x3,x4,x5);
head(T)
```

Time	x1		x2	x3	x4	x5
0 min	-1.238	-0.39275	0	large	0	0.84147
0.01 min	0.24413	-0.62204	0.01	medium	0.18701	0.97736

```
    0.02 min        1.3983       -1.1905      0.03     medium     0.36665       0.97543
    0.03 min     -0.095473       -1.8785      0.06     small      0.53262       0.83706
    0.04 min       0.38762       -0.42401      0.1     small      0.67909       0.58267
    0.05 min      -0.96631        0.77724     0.15     small      0.80097        0.2488
    0.06 min        1.5092       -0.71393     0.21     medium     0.89403      -0.11722
    0.07 min       0.40383        1.5846      0.28     small      0.95512      -0.46392
```

```matlab
% fit an armax model to x2 which is the first valid signal in T
NN = [1 1]; % na = nc = 1
model_armax1 = armax(T,NN,'OutputName','x2');
model_armax1.OutputName
```

```
ans = 1×1 cell array
    {'x2'}
```

```matlab
% fit an armax model to x4 which is the second valid signal in T
NN = [3 2]; % na = 3, nc = 2
model_armax2 = armax(T,NN,'OutputName','x4');
model_armax2.OutputName
```

```
ans = 1×1 cell array
    {'x4'}
```

```matlab
% fit a multi-variate BJ model to x5 , x2 in that order
nc = [3; 2];
nd = [3; 3];
NN = [nc, nd];
model_bj = bj(T,NN,'OutputName',{'x5';'x2'});
model_bj.OutputName
```

```
ans = 2×1 cell
    {'x5'}
    {'x2'}
```

In the case of Nonlinear ARX models, if the regressors are specified using model order matrix, or by using a linear model, then the I/O size can be determined uniquely as in the case of ARX models. Otherwise, name-value pairs 'InputName' and 'OutputName' must be used. In particular, use 'InputName'/[] to indicate the absence of inputs.

```matlab
% Identify a Nonlinear ARX time series model
% Prepare some data
t = (0:100)'/100;
x1 = randn(length(t),2); % some other vector valued data
x2 = cumsum(t).^2;
x3 = rand(101,1);
x3 = discretize(x3,[0 .25 .75 1],'categorical',{'small', 'medium', 'large'});
x4 = exp(-0.2*t).*sin(2*pi*3*t).*(t+1)./(t+5); % some signal
x5 = exp(-0.4*t).*sin(2*pi*6*t+1); % some other signal
T = timetable(minutes(t),x1,x2,x3,x4,x5);
clf
stackedplot(T)
```
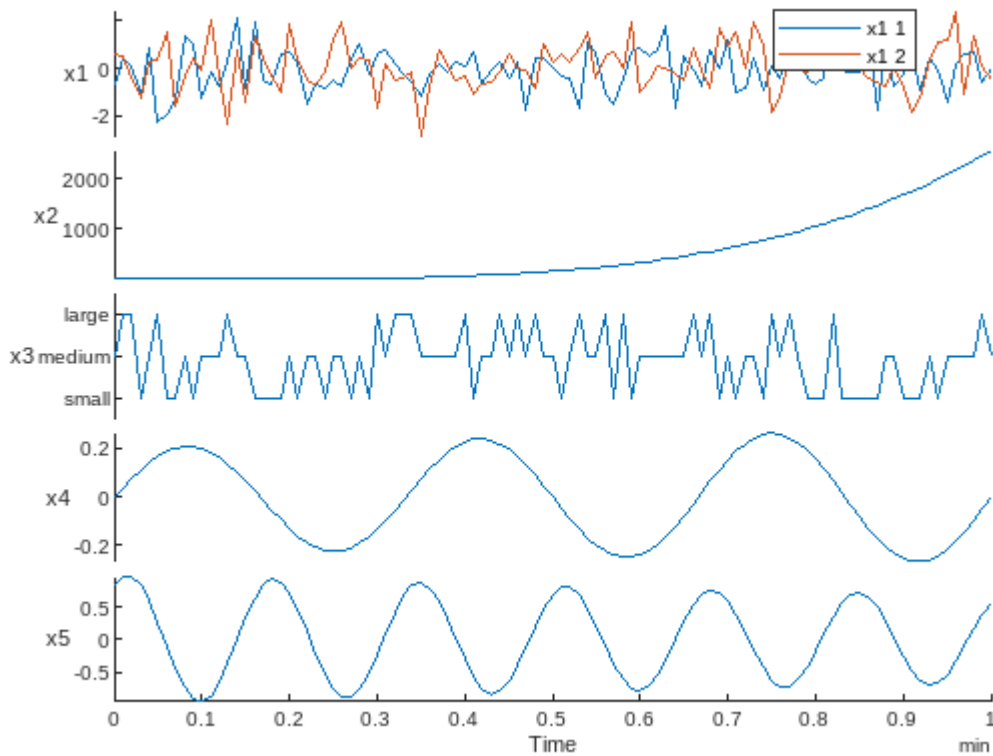
```
% x2, x4 and x5 are valid signals
% Identify a multivariate time series models for {'x2','x4','x5'} signals
L = linearRegressor({'x2','x4','x5'},1:10); % linear terms in x2, x4, x5
P1 = polynomialRegressor({'x2','x4'},1:2,2); % second order polynomial terms in x2, x4
P2 = polynomialRegressor({'x5'},1:2,3);    % third order polynomial terms in x5
H = periodicRegressor({'x2','x4','x5'},5:2:15,.20,10);
% Stack all the regressors into one variable
Regressors = [L;P1;P2;H]; % Regressors is a 4-by-1 object
% view all the regressors contained in these specifications
getreg(Regressors)'
```

```
ans = 1×396 string
  Columns 1 through 6

    "x2(t-1)"    "x2(t-2)"    "x2(t-3)"    "x2(t-4)"    "x2(t-5)"    "x2(t-6)"

  Columns 7 through 12

    "x2(t-7)"    "x2(t-8)"    "x2(t-9)"    "x2(t-10)"    "x4(t-1)"    "x4(t-2)"

  Columns 13 through 18

    "x4(t-3)"    "x4(t-4)"    "x4(t-5)"    "x4(t-6)"    "x4(t-7)"    "x4(t-8)"

  Columns 19 through 24

    "x4(t-9)"    "x4(t-10)"    "x5(t-1)"    "x5(t-2)"    "x5(t-3)"    "x5(t-4)"
```

```
Columns 25 through 30

  "x5(t-5)"     "x5(t-6)"     "x5(t-7)"     "x5(t-8)"     "x5(t-9)"     "x5(t-10)"

Columns 31 through 35

  "x2(t-1)^2"    "x2(t-2)^2"    "x4(t-1)^2"    "x4(t-2)^2"    "x5(t-1)^3"

Columns 36 through 38

  "x5(t-2)^3"    "sin(0.2*x2(t-5))"    "sin(2*0.2*x2(t-5))"

Columns 39 through 41

  "sin(3*0.2*x2(t-5))"    "sin(4*0.2*x2(t-5))"    "sin(5*0.2*x2(t-5))"

Columns 42 through 44

  "sin(6*0.2*x2(t-5))"    "sin(7*0.2*x2(t-5))"    "sin(8*0.2*x2(t-5))"

Columns 45 through 47

  "sin(9*0.2*x2(t-5))"    "sin(10*0.2*x2(t-5))"    "cos(0.2*x2(t-5))"

Columns 48 through 50

  "cos(2*0.2*x2(t-5))"    "cos(3*0.2*x2(t-5))"    "cos(4*0.2*x2(t-5))"

Columns 51 through 53

  "cos(5*0.2*x2(t-5))"    "cos(6*0.2*x2(t-5))"    "cos(7*0.2*x2(t-5))"

Columns 54 through 56

  "cos(8*0.2*x2(t-5))"    "cos(9*0.2*x2(t-5))"    "cos(10*0.2*x2(t-5))"

Columns 57 through 59

  "sin(0.2*x2(t-7))"    "sin(2*0.2*x2(t-7))"    "sin(3*0.2*x2(t-7))"

Columns 60 through 62

  "sin(4*0.2*x2(t-7))"    "sin(5*0.2*x2(t-7))"    "sin(6*0.2*x2(t-7))"

Columns 63 through 65

  "sin(7*0.2*x2(t-7))"    "sin(8*0.2*x2(t-7))"    "sin(9*0.2*x2(t-7))"

Columns 66 through 68

  "sin(10*0.2*x2(t-7))"    "cos(0.2*x2(t-7))"    "cos(2*0.2*x2(t-7))"

Columns 69 through 71

  "cos(3*0.2*x2(t-7))"    "cos(4*0.2*x2(t-7))"    "cos(5*0.2*x2(t-7))"

Columns 72 through 74
```

```
   "cos(6*0.2*x2(t-7))"      "cos(7*0.2*x2(t-7))"      "cos(8*0.2*x2(t-7))"

Columns 75 through 77

   "cos(9*0.2*x2(t-7))"      "cos(10*0.2*x2(t-7))"     "sin(0.2*x2(t-9))"

Columns 78 through 80

   "sin(2*0.2*x2(t-9))"      "sin(3*0.2*x2(t-9))"      "sin(4*0.2*x2(t-9))"

Columns 81 through 83

   "sin(5*0.2*x2(t-9))"      "sin(6*0.2*x2(t-9))"      "sin(7*0.2*x2(t-9))"

Columns 84 through 86

   "sin(8*0.2*x2(t-9))"      "sin(9*0.2*x2(t-9))"      "sin(10*0.2*x2(t-9))"

Columns 87 through 89

   "cos(0.2*x2(t-9))"       "cos(2*0.2*x2(t-9))"       "cos(3*0.2*x2(t-9))"

Columns 90 through 92

   "cos(4*0.2*x2(t-9))"      "cos(5*0.2*x2(t-9))"      "cos(6*0.2*x2(t-9))"

Columns 93 through 95

   "cos(7*0.2*x2(t-9))"      "cos(8*0.2*x2(t-9))"      "cos(9*0.2*x2(t-9))"

Columns 96 through 98

   "cos(10*0.2*x2(t-9))"     "sin(0.2*x2(t-11))"       "sin(2*0.2*x2(t-11))"

Columns 99 through 101

   "sin(3*0.2*x2(t-11))"     "sin(4*0.2*x2(t-11))"     "sin(5*0.2*x2(t-11))"

Columns 102 through 104

   "sin(6*0.2*x2(t-11))"     "sin(7*0.2*x2(t-11))"     "sin(8*0.2*x2(t-11))"

Columns 105 through 107

   "sin(9*0.2*x2(t-11))"     "sin(10*0.2*x2(t-11…"     "cos(0.2*x2(t-11))"

Columns 108 through 110

   "cos(2*0.2*x2(t-11))"     "cos(3*0.2*x2(t-11))"     "cos(4*0.2*x2(t-11))"

Columns 111 through 113

   "cos(5*0.2*x2(t-11))"     "cos(6*0.2*x2(t-11))"     "cos(7*0.2*x2(t-11))"

Columns 114 through 116

   "cos(8*0.2*x2(t-11))"     "cos(9*0.2*x2(t-11))"     "cos(10*0.2*x2(t-11…"
```

```
Columns 117 through 119

  "sin(0.2*x2(t-13))"    "sin(2*0.2*x2(t-13))"    "sin(3*0.2*x2(t-13))"

Columns 120 through 122

  "sin(4*0.2*x2(t-13))"    "sin(5*0.2*x2(t-13))"    "sin(6*0.2*x2(t-13))"

Columns 123 through 125

  "sin(7*0.2*x2(t-13))"    "sin(8*0.2*x2(t-13))"    "sin(9*0.2*x2(t-13))"

Columns 126 through 128

  "sin(10*0.2*x2(t-13…"    "cos(0.2*x2(t-13))"    "cos(2*0.2*x2(t-13))"

Columns 129 through 131

  "cos(3*0.2*x2(t-13))"    "cos(4*0.2*x2(t-13))"    "cos(5*0.2*x2(t-13))"

Columns 132 through 134

  "cos(6*0.2*x2(t-13))"    "cos(7*0.2*x2(t-13))"    "cos(8*0.2*x2(t-13))"

Columns 135 through 137

  "cos(9*0.2*x2(t-13))"    "cos(10*0.2*x2(t-13…"    "sin(0.2*x2(t-15))"

Columns 138 through 140

  "sin(2*0.2*x2(t-15))"    "sin(3*0.2*x2(t-15))"    "sin(4*0.2*x2(t-15))"

Columns 141 through 143

  "sin(5*0.2*x2(t-15))"    "sin(6*0.2*x2(t-15))"    "sin(7*0.2*x2(t-15))"

Columns 144 through 146

  "sin(8*0.2*x2(t-15))"    "sin(9*0.2*x2(t-15))"    "sin(10*0.2*x2(t-15…"

Columns 147 through 149

  "cos(0.2*x2(t-15))"    "cos(2*0.2*x2(t-15))"    "cos(3*0.2*x2(t-15))"

Columns 150 through 152

  "cos(4*0.2*x2(t-15))"    "cos(5*0.2*x2(t-15))"    "cos(6*0.2*x2(t-15))"

Columns 153 through 155

  "cos(7*0.2*x2(t-15))"    "cos(8*0.2*x2(t-15))"    "cos(9*0.2*x2(t-15))"

Columns 156 through 158

  "cos(10*0.2*x2(t-15…"    "sin(0.2*x4(t-5))"    "sin(2*0.2*x4(t-5))"

Columns 159 through 161
```

```
   "sin(3*0.2*x4(t-5))"     "sin(4*0.2*x4(t-5))"     "sin(5*0.2*x4(t-5))"

 Columns 162 through 164

   "sin(6*0.2*x4(t-5))"     "sin(7*0.2*x4(t-5))"     "sin(8*0.2*x4(t-5))"

 Columns 165 through 167

   "sin(9*0.2*x4(t-5))"     "sin(10*0.2*x4(t-5))"     "cos(0.2*x4(t-5))"

 Columns 168 through 170

   "cos(2*0.2*x4(t-5))"     "cos(3*0.2*x4(t-5))"     "cos(4*0.2*x4(t-5))"

 Columns 171 through 173

   "cos(5*0.2*x4(t-5))"     "cos(6*0.2*x4(t-5))"     "cos(7*0.2*x4(t-5))"

 Columns 174 through 176

   "cos(8*0.2*x4(t-5))"     "cos(9*0.2*x4(t-5))"     "cos(10*0.2*x4(t-5))"

 Columns 177 through 179

   "sin(0.2*x4(t-7))"     "sin(2*0.2*x4(t-7))"     "sin(3*0.2*x4(t-7))"

 Columns 180 through 182

   "sin(4*0.2*x4(t-7))"     "sin(5*0.2*x4(t-7))"     "sin(6*0.2*x4(t-7))"

 Columns 183 through 185

   "sin(7*0.2*x4(t-7))"     "sin(8*0.2*x4(t-7))"     "sin(9*0.2*x4(t-7))"

 Columns 186 through 188

   "sin(10*0.2*x4(t-7))"     "cos(0.2*x4(t-7))"     "cos(2*0.2*x4(t-7))"

 Columns 189 through 191

   "cos(3*0.2*x4(t-7))"     "cos(4*0.2*x4(t-7))"     "cos(5*0.2*x4(t-7))"

 Columns 192 through 194

   "cos(6*0.2*x4(t-7))"     "cos(7*0.2*x4(t-7))"     "cos(8*0.2*x4(t-7))"

 Columns 195 through 197

   "cos(9*0.2*x4(t-7))"     "cos(10*0.2*x4(t-7))"     "sin(0.2*x4(t-9))"

 Columns 198 through 200

   "sin(2*0.2*x4(t-9))"     "sin(3*0.2*x4(t-9))"     "sin(4*0.2*x4(t-9))"

 Columns 201 through 203

   "sin(5*0.2*x4(t-9))"     "sin(6*0.2*x4(t-9))"     "sin(7*0.2*x4(t-9))"
```

```
Columns 204 through 206

   "sin(8*0.2*x4(t-9))"     "sin(9*0.2*x4(t-9))"     "sin(10*0.2*x4(t-9))"

Columns 207 through 209

   "cos(0.2*x4(t-9))"     "cos(2*0.2*x4(t-9))"     "cos(3*0.2*x4(t-9))"

Columns 210 through 212

   "cos(4*0.2*x4(t-9))"     "cos(5*0.2*x4(t-9))"     "cos(6*0.2*x4(t-9))"

Columns 213 through 215

   "cos(7*0.2*x4(t-9))"     "cos(8*0.2*x4(t-9))"     "cos(9*0.2*x4(t-9))"

Columns 216 through 218

   "cos(10*0.2*x4(t-9))"     "sin(0.2*x4(t-11))"     "sin(2*0.2*x4(t-11))"

Columns 219 through 221

   "sin(3*0.2*x4(t-11))"     "sin(4*0.2*x4(t-11))"     "sin(5*0.2*x4(t-11))"

Columns 222 through 224

   "sin(6*0.2*x4(t-11))"     "sin(7*0.2*x4(t-11))"     "sin(8*0.2*x4(t-11))"

Columns 225 through 227

   "sin(9*0.2*x4(t-11))"     "sin(10*0.2*x4(t-11…"     "cos(0.2*x4(t-11))"

Columns 228 through 230

   "cos(2*0.2*x4(t-11))"     "cos(3*0.2*x4(t-11))"     "cos(4*0.2*x4(t-11))"

Columns 231 through 233

   "cos(5*0.2*x4(t-11))"     "cos(6*0.2*x4(t-11))"     "cos(7*0.2*x4(t-11))"

Columns 234 through 236

   "cos(8*0.2*x4(t-11))"     "cos(9*0.2*x4(t-11))"     "cos(10*0.2*x4(t-11…"

Columns 237 through 239

   "sin(0.2*x4(t-13))"     "sin(2*0.2*x4(t-13))"     "sin(3*0.2*x4(t-13))"

Columns 240 through 242

   "sin(4*0.2*x4(t-13))"     "sin(5*0.2*x4(t-13))"     "sin(6*0.2*x4(t-13))"

Columns 243 through 245

   "sin(7*0.2*x4(t-13))"     "sin(8*0.2*x4(t-13))"     "sin(9*0.2*x4(t-13))"

Columns 246 through 248
```

```
    "sin(10*0.2*x4(t-13…"    "cos(0.2*x4(t-13))"    "cos(2*0.2*x4(t-13))"

  Columns 249 through 251

    "cos(3*0.2*x4(t-13))"    "cos(4*0.2*x4(t-13))"    "cos(5*0.2*x4(t-13))"

  Columns 252 through 254

    "cos(6*0.2*x4(t-13))"    "cos(7*0.2*x4(t-13))"    "cos(8*0.2*x4(t-13))"

  Columns 255 through 257

    "cos(9*0.2*x4(t-13))"    "cos(10*0.2*x4(t-13…"    "sin(0.2*x4(t-15))"

  Columns 258 through 260

    "sin(2*0.2*x4(t-15))"    "sin(3*0.2*x4(t-15))"    "sin(4*0.2*x4(t-15))"

  Columns 261 through 263

    "sin(5*0.2*x4(t-15))"    "sin(6*0.2*x4(t-15))"    "sin(7*0.2*x4(t-15))"

  Columns 264 through 266

    "sin(8*0.2*x4(t-15))"    "sin(9*0.2*x4(t-15))"    "sin(10*0.2*x4(t-15…"

  Columns 267 through 269

    "cos(0.2*x4(t-15))"    "cos(2*0.2*x4(t-15))"    "cos(3*0.2*x4(t-15))"

  Columns 270 through 272

    "cos(4*0.2*x4(t-15))"    "cos(5*0.2*x4(t-15))"    "cos(6*0.2*x4(t-15))"

  Columns 273 through 275

    "cos(7*0.2*x4(t-15))"    "cos(8*0.2*x4(t-15))"    "cos(9*0.2*x4(t-15))"

  Columns 276 through 278

    "cos(10*0.2*x4(t-15…"    "sin(0.2*x5(t-5))"    "sin(2*0.2*x5(t-5))"

  Columns 279 through 281

    "sin(3*0.2*x5(t-5))"    "sin(4*0.2*x5(t-5))"    "sin(5*0.2*x5(t-5))"

  Columns 282 through 284

    "sin(6*0.2*x5(t-5))"    "sin(7*0.2*x5(t-5))"    "sin(8*0.2*x5(t-5))"

  Columns 285 through 287

    "sin(9*0.2*x5(t-5))"    "sin(10*0.2*x5(t-5))"    "cos(0.2*x5(t-5))"

  Columns 288 through 290

    "cos(2*0.2*x5(t-5))"    "cos(3*0.2*x5(t-5))"    "cos(4*0.2*x5(t-5))"
```

```
Columns 291 through 293

  "cos(5*0.2*x5(t-5))"     "cos(6*0.2*x5(t-5))"     "cos(7*0.2*x5(t-5))"

Columns 294 through 296

  "cos(8*0.2*x5(t-5))"     "cos(9*0.2*x5(t-5))"     "cos(10*0.2*x5(t-5))"

Columns 297 through 299

  "sin(0.2*x5(t-7))"     "sin(2*0.2*x5(t-7))"     "sin(3*0.2*x5(t-7))"

Columns 300 through 302

  "sin(4*0.2*x5(t-7))"     "sin(5*0.2*x5(t-7))"     "sin(6*0.2*x5(t-7))"

Columns 303 through 305

  "sin(7*0.2*x5(t-7))"     "sin(8*0.2*x5(t-7))"     "sin(9*0.2*x5(t-7))"

Columns 306 through 308

  "sin(10*0.2*x5(t-7))"     "cos(0.2*x5(t-7))"     "cos(2*0.2*x5(t-7))"

Columns 309 through 311

  "cos(3*0.2*x5(t-7))"     "cos(4*0.2*x5(t-7))"     "cos(5*0.2*x5(t-7))"

Columns 312 through 314

  "cos(6*0.2*x5(t-7))"     "cos(7*0.2*x5(t-7))"     "cos(8*0.2*x5(t-7))"

Columns 315 through 317

  "cos(9*0.2*x5(t-7))"     "cos(10*0.2*x5(t-7))"     "sin(0.2*x5(t-9))"

Columns 318 through 320

  "sin(2*0.2*x5(t-9))"     "sin(3*0.2*x5(t-9))"     "sin(4*0.2*x5(t-9))"

Columns 321 through 323

  "sin(5*0.2*x5(t-9))"     "sin(6*0.2*x5(t-9))"     "sin(7*0.2*x5(t-9))"

Columns 324 through 326

  "sin(8*0.2*x5(t-9))"     "sin(9*0.2*x5(t-9))"     "sin(10*0.2*x5(t-9))"

Columns 327 through 329

  "cos(0.2*x5(t-9))"     "cos(2*0.2*x5(t-9))"     "cos(3*0.2*x5(t-9))"

Columns 330 through 332

  "cos(4*0.2*x5(t-9))"     "cos(5*0.2*x5(t-9))"     "cos(6*0.2*x5(t-9))"

Columns 333 through 335
```

```
   "cos(7*0.2*x5(t-9))"     "cos(8*0.2*x5(t-9))"     "cos(9*0.2*x5(t-9))"

Columns 336 through 338

   "cos(10*0.2*x5(t-9))"     "sin(0.2*x5(t-11))"     "sin(2*0.2*x5(t-11))"

Columns 339 through 341

   "sin(3*0.2*x5(t-11))"     "sin(4*0.2*x5(t-11))"     "sin(5*0.2*x5(t-11))"

Columns 342 through 344

   "sin(6*0.2*x5(t-11))"     "sin(7*0.2*x5(t-11))"     "sin(8*0.2*x5(t-11))"

Columns 345 through 347

   "sin(9*0.2*x5(t-11))"     "sin(10*0.2*x5(t-11…"     "cos(0.2*x5(t-11))"

Columns 348 through 350

   "cos(2*0.2*x5(t-11))"     "cos(3*0.2*x5(t-11))"     "cos(4*0.2*x5(t-11))"

Columns 351 through 353

   "cos(5*0.2*x5(t-11))"     "cos(6*0.2*x5(t-11))"     "cos(7*0.2*x5(t-11))"

Columns 354 through 356

   "cos(8*0.2*x5(t-11))"     "cos(9*0.2*x5(t-11))"     "cos(10*0.2*x5(t-11…"

Columns 357 through 359

   "sin(0.2*x5(t-13))"     "sin(2*0.2*x5(t-13))"     "sin(3*0.2*x5(t-13))"

Columns 360 through 362

   "sin(4*0.2*x5(t-13))"     "sin(5*0.2*x5(t-13))"     "sin(6*0.2*x5(t-13))"

Columns 363 through 365

   "sin(7*0.2*x5(t-13))"     "sin(8*0.2*x5(t-13))"     "sin(9*0.2*x5(t-13))"

Columns 366 through 368

   "sin(10*0.2*x5(t-13…"     "cos(0.2*x5(t-13))"     "cos(2*0.2*x5(t-13))"

Columns 369 through 371

   "cos(3*0.2*x5(t-13))"     "cos(4*0.2*x5(t-13))"     "cos(5*0.2*x5(t-13))"

Columns 372 through 374

   "cos(6*0.2*x5(t-13))"     "cos(7*0.2*x5(t-13))"     "cos(8*0.2*x5(t-13))"

Columns 375 through 377

   "cos(9*0.2*x5(t-13))"     "cos(10*0.2*x5(t-13…"     "sin(0.2*x5(t-15))"
```

```
   Columns 378 through 380

     "sin(2*0.2*x5(t-15))"    "sin(3*0.2*x5(t-15))"    "sin(4*0.2*x5(t-15))"

   Columns 381 through 383

     "sin(5*0.2*x5(t-15))"    "sin(6*0.2*x5(t-15))"    "sin(7*0.2*x5(t-15))"

   Columns 384 through 386

     "sin(8*0.2*x5(t-15))"    "sin(9*0.2*x5(t-15))"    "sin(10*0.2*x5(t-15…"

   Columns 387 through 389

     "cos(0.2*x5(t-15))"    "cos(2*0.2*x5(t-15))"    "cos(3*0.2*x5(t-15))"

   Columns 390 through 392

     "cos(4*0.2*x5(t-15))"    "cos(5*0.2*x5(t-15))"    "cos(6*0.2*x5(t-15))"

   Columns 393 through 395

     "cos(7*0.2*x5(t-15))"    "cos(8*0.2*x5(t-15))"    "cos(9*0.2*x5(t-15))"

   Column 396

     "cos(10*0.2*x5(t-15…"
```

```
% Fit an SVM based Nonlinear ARX model
NonlinearFcn = idSupportVectorMachine; % use SVM for all the 3 variables x2, x4, x5
warning('off','Ident:estimation:NparGTNsamp');
sysNL = nlarx(T, Regressors, NonlinearFcn, 'InputName', [])

sysNL =

Nonlinear multivariate time series model with 3 outputs
  Outputs: x2, x4, x5

Regressors:
  1. Linear regressors in variables x2, x4, x5
  2. Order 2 regressors in variables x2, x4
  3. Order 3 regressors in variables x5
  4. Periodic regressors in variables x2, x4, x5 with W = 0.2, and 10 Fourier terms
  List of all regressors

Output functions:
  Output 1:  Support Vector Machine function using a Gaussian kernel
  Output 2:  Support Vector Machine function using a Gaussian kernel
  Output 3:  Support Vector Machine function using a Gaussian kernel

Sample time: 0.6 seconds

Status:
Estimated using NLARX on time domain data "T".
Fit to estimation data: [75.08;70.92;44.11]% (prediction focus)
MSE: 2.986e+04
```

2-159

```
More information in model's "Report" property.

Model Properties

% forecast the response 100 steps ahead, using T as past data
clf
forecast(sysNL, T(1:50,:), 10)
xlim([0.4 0.6])
```
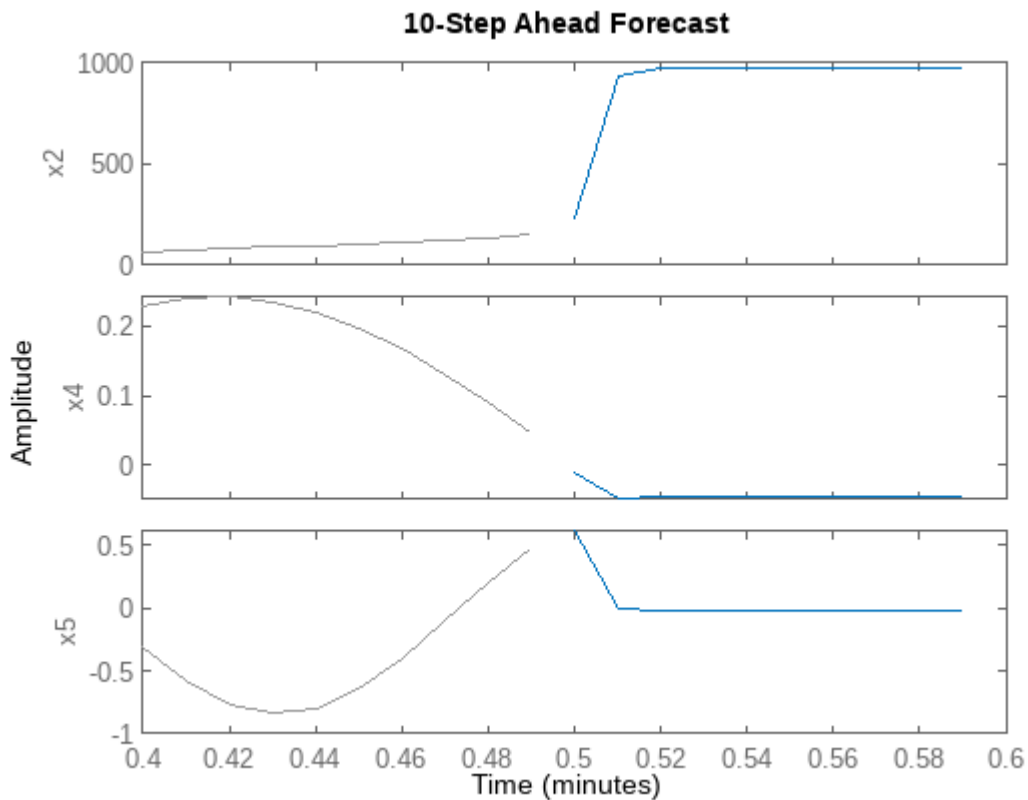

Using the iddata object

When using the iddata object, the input/output size of the data must match those of the model. Thus for bivariate time series model, create an iddata object with 2 outputs and 0 inputs.

```
z = iddata([x5 x2],[],0.01,'Tstart',0,'OutputName',{'x5';'x2'});
nc = [3; 2];
nd = [3; 3];
NN = [nc, nd];
model_bj = bj(z,NN)

model_bj =
Discrete-time Polynomial model:
  Model for output "x5": y_1(t) = [C(z)/D(z)]e_1(t)
    C(z) = 1 + 3.989e-15 z^-1 - 6.315e-15 z^-2 + 3.158e-15 z^-3

    D(z) = 1 - 1.171 z^-1 - 0.27 z^-2 + 0.676 z^-3

  Model for output "x2": y_2(t) = [C(z)/D(z)]e_2(t)
```

```
    C(z) = 1 + 1.882 z^-1 + 0.976 z^-2

    D(z) = 1 - 3.105 z^-1 + 3.214 z^-2 - 1.11 z^-3

Sample time: 0.01 seconds

Parameterization:
   Polynomial orders:    nc=[3;2]    nd=[3;3]
   Number of free coefficients: 11
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using BJ on time domain data "z".
Fit to estimation data: [100;100]% (prediction focus)
FPE: 5.691e-37, MSE: 2.319e-07

Model Properties
```

compare(z,model_bj)
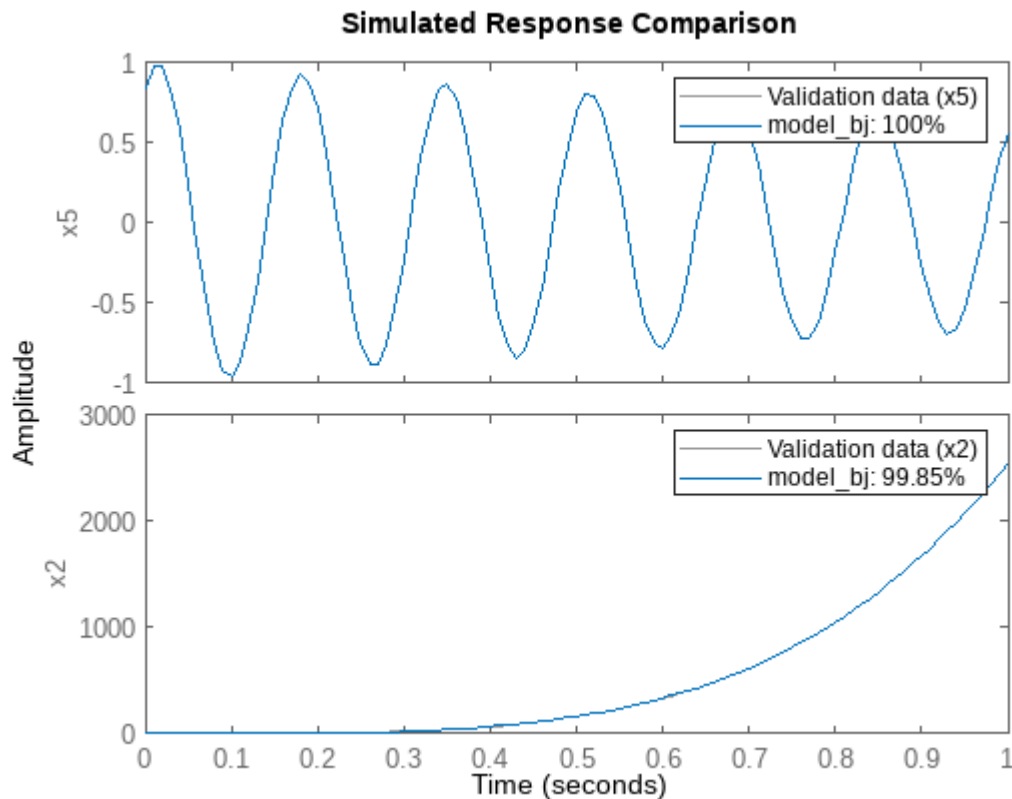

```
% restore warning state
delete(Restore)
```

Summary

System Identification Toolbox lets you work with data in either time- or frequency-domain for identifying dynamic systems. There is no restriction on the number of input/output channels; you can

wok with datasets that represent one or more output channels, and zero or more input channels. Key takeaways:

1 There are many ways of specifying time-domain signals for identification - using double matrices, timetables, or iddata objects.
2 Frequency-domain input/output signals must be represented using iddata objects.
3 For multi-experiment data, use a cell array of matrices or timetables, or a multi-experiment iddata object. A multi-experiment iddata representation is typically obtained by combining a set of single-experiment iddata objects together using the `merge` command.
4 Frequency response data (FRF) must be represented using idfrd objects. Set the data sample time to zero if the FRF is obtained by direct measurements such as by using a spectrum analyzer.
5 For continuous-time model identification, it is important to specify the correct sample time and input intersample behavior.

See Also
`timetable` | `array2timetable` | `iddata` | `idfrd` | `compare` | `sim` | `fft` | `merge (iddata)`

Related Examples
- "Representing Data in MATLAB Workspace" on page 2-8
- "Dealing with Multi-Experiment Data and Merging Models" on page 2-44
- "Frequency Domain Identification: Estimating Models Using Frequency Domain Data" on page 4-52

3

Transform Data

Supported Data Transformations

The following table shows the different ways you can transform data from one data domain to another. If the transformation is supported for a given row and column combination in the table, the command used by the software is listed in the cell at their intersection.

Original Data Format	To Time-Domain Data (iddata object)	To Frequency-Domain Data (iddata object)	To Frequency-Response Data (idfrd object)
Time-Domain Data (iddata object)	N/A	Use `fft`	• Use `etfe`, `spa`, or `spafdr`. • Estimate a linear parametric model from the `iddata` object, and use `idfrd` to compute frequency-response data.
Frequency-Domain Data (iddata object)	Use `ifft` (works only for evenly spaced frequency-domain data).	N/A	• Use `etfe`, `spa`, or `spafdr`. • Estimate a linear parametric model from the `iddata` object, and use `idfrd` to compute frequency-response data.
Frequency-Response Data (idfrd object)	Not supported	Use `iddata`. The software creates a frequency-domain `iddata` object that has the same ratio between output and input as the original `idfrd` object frequency-response data.	• Use `spafdr`. The software calculates frequency-response data with a different resolution (number and spacing of frequencies) than the original data.

Transforming from time-domain or frequency-domain data to frequency-response data is equivalent to creating a frequency-response model from the data. For more information, see "Frequency-Response Models".

See Also

Related Examples

More About

Transform Time-Domain Data in the App

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

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

1   In the System Identification app, drag the icon of the data you want to transform to the **Working Data** rectangle.

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

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

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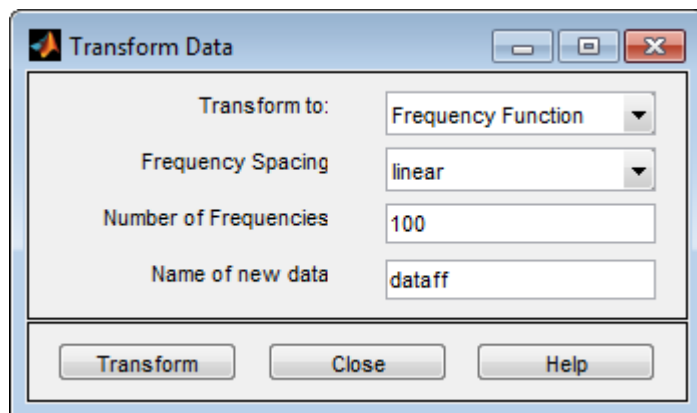
 • `Frequency Domain Data` — Create a new `iddata` object using the `fft` method. Go to step 6.

4   In the **Frequency Spacing** list, select the spacing of the frequencies at which the frequency function is estimated:

 • `linear` — Uniform spacing of frequency values between the endpoints.

 • `logarithmic` — Base-10 logarithmic spacing of frequency values between the endpoints.

5   In the **Number of Frequencies** field, enter the number of frequency values.

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

7   Click **Transform** to add the new data set to the Data Board in the System Identification app.

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

See Also

Related Examples

 •   "Transform Between Time-Domain and Frequency-Domain Data" on page 3-8

- "Transform Frequency-Domain Data in the App" on page 3-5
- "Transform Frequency-Response Data in the App" on page 3-6

More About

- "Representing Data in MATLAB Workspace" on page 2-8
- "Supported Data Transformations" on page 3-2

Transform Frequency-Domain Data in the App

In the System Identification app, frequency-domain data has an icon with a green background. You can transform frequency-domain data to time-domain or frequency-response (frequency-function) data.

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

1   In the System Identification app, drag the icon of the data you want to transform to the **Working Data** rectangle.

2   Select **<--Preprocess > Transform data**.

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

   - `Frequency Function` — Create a new `idfrd` object using the `spafdr` method. Go to step 4.
   - `Time Domain Data` — Create a new `iddata` object using the `ifft` (inverse fast Fourier transform) method. Go to step 6.

4   In the **Frequency Spacing** list, select the spacing of the frequencies at which the frequency function is estimated:

   - `linear` — Uniform spacing of frequency values between the endpoints.
   - `logarithmic` — Base-10 logarithmic spacing of frequency values between the endpoints.

5   In the **Number of Frequencies** field, enter the number of frequency values.

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

7   Click **Transform** to add the new data set to the Data Board in the System Identification app.

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

See Also

Related Examples
- "Transform Between Time-Domain and Frequency-Domain Data" on page 3-8
- "Transform Time-Domain Data in the App" on page 3-3
- "Transform Frequency-Response Data in the App" on page 3-6

More About
- "Representing Data in MATLAB Workspace" on page 2-8
- "Supported Data Transformations" on page 3-2

Transform Frequency-Response Data in the App

In the System Identification app, frequency-response data has an icon with a yellow background. You can transform frequency-response data to frequency-domain data (`iddata` object) or to frequency-response data with a different frequency resolution.

When you select to transform single-input/single-output (SISO) frequency-response data to frequency-domain data, the toolbox creates outputs that equal the frequency responses, and inputs equal to 1. Therefore, the ratio between the Fourier transform of the output and the Fourier transform of the input is equal to the system frequency response.

For the multiple-input case, the toolbox transforms the frequency-response data to frequency-domain data as if each input contributes independently to the entire output of the system and then combines information. For example, if a system has three inputs, `u1`, `u2`, and `u3` and two frequency samples, the input matrix is set to:

$$
\begin{bmatrix}
1 & 0 & 0 \\
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
0 & 0 & 1
\end{bmatrix}
$$

In general, for `nu` inputs and `ns` samples (the number of frequencies), the input matrix has `nu` columns and (`ns· nu`) rows.

Note To create a separate experiment for the response from each input, see "Transforming Between Frequency-Domain and Frequency-Response Data" on page 3-14.

When you transform frequency-response data by changing its frequency resolution, you can modify the number of frequency values by changing between linear or logarithmic spacing. You might specify variable frequency spacing to increase the number of data points near the system resonance frequencies, and also make the frequency vector coarser in the region outside the system dynamics. Typically, high-frequency noise dominates away from frequencies where interesting system dynamics occur. The System Identification app lets you specify logarithmic frequency spacing, which results in a variable frequency resolution.

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

---

1   In the System Identification app, drag the icon of the data you want to transform to the **Working Data** rectangle.

2   Select **<--Preprocess > Transform data**.

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

- `Frequency Domain Data` — Create a new `iddata` object. Go to step 6.
- `Frequency Function` — Create a new `idfrd` object with different resolution (number and spacing of frequencies) using the `spafdr` method. Go to step 4.

4 In the **Frequency Spacing** list, select the spacing of the frequencies at which the frequency function is estimated:

- linear — Uniform spacing of frequency values between the endpoints.
- logarithmic — Base-10 logarithmic spacing of frequency values between the endpoints.

5 In the **Number of Frequencies** field, enter the number of frequency values.

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

7 Click **Transform** to add the new data set to the Data Board in the System Identification app.

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

See Also

Related Examples

- "Transform Between Time-Domain and Frequency-Domain Data" on page 3-8
- "Transform Time-Domain Data in the App" on page 3-3
- "Transform Frequency-Domain Data in the App" on page 3-5

More About

- "Representing Data in MATLAB Workspace" on page 2-8
- "Supported Data Transformations" on page 3-2

Transform Between Time-Domain and Frequency-Domain Data

System Identification Toolbox provides tools for analyzing data and for estimating and evaluating models in both the time and the frequency domains. To use tools and methods that are not in the same domain as your measured data, you can transform your data between the time domain and the frequency domain.

The `iddata` object stores time-domain or frequency-domain data.

- Time-domain data consists of one or more input variables *u(t)* and one or more output variables *y(t)*, sampled as a function of time.
- 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.

For detailed information about representing time-domain and frequency-domain data in MATLAB, see "Representing Data in MATLAB Workspace" on page 2-8.

You can transform your data from one domain to the other. The table summarizes the commands for transforming data between the time and frequency domains. For more command information, see the corresponding command reference pages.

Command	Description	Syntax Example

`fft`	Transform 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`	Transform 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)`

Converting `iddata` data into the form of an `idfrd` frequency response is a type of estimation. If you want to estimate the frequency response using an `iddata` object, see "Transforming Between Frequency-Domain and Frequency-Response Data" on page 3-14.

Transform Data Between Time and Frequency Domains

Transform data from the time domain to the frequency domain and back to the time domain, and compare performance for models estimated from the original and transformed data.

Load and plot the time-domain data `z1`, which contains 300 samples.

```
load iddata1 z1
plot(z1)
```

Input-Output Data
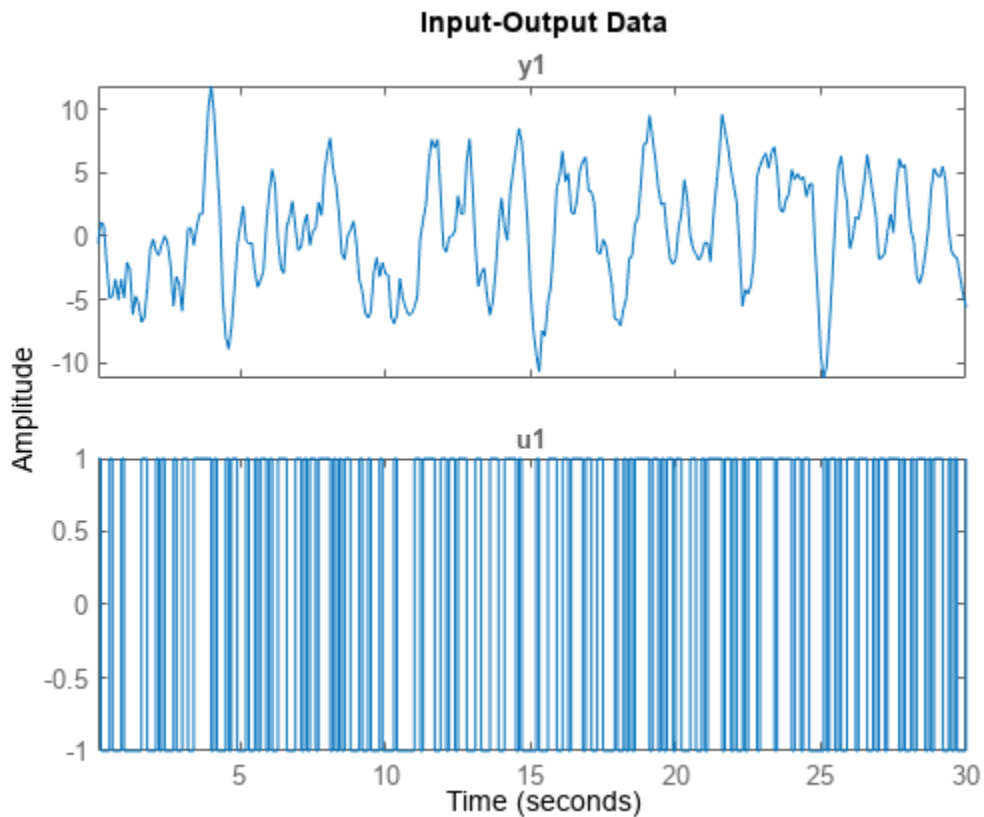
Find the sample time `Ts` of `z1`.

```
Ts = z1.Ts
```

```
Ts = 0.1000
```

The sample time is 0.1 s.

Transform `z1` into the frequency domain.

```
z1f = fft(z1)
```

```
z1f =

Frequency domain data set with responses at 151 frequencies.
Frequency range: 0 to 31.416 rad/seconds
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

The frequency range extends to 31.416 rad/s, which is equal to the Nyquist frequency of `pi/Ts`.

Plot the frequency-domain data.

```
plot(z1f)
```

Input-Output Data
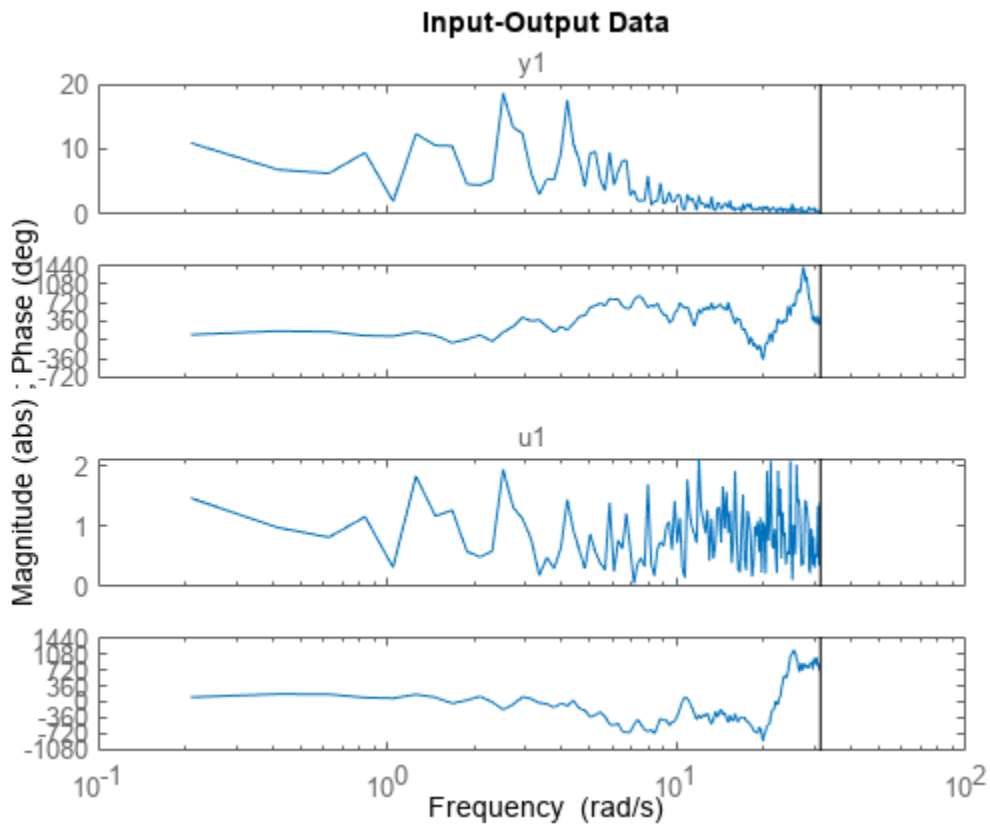
Transform `z1f` back to the time domain and plot the two time-domain signals together..

```
z1t = ifft(z1f)
```

```
z1t =
```

```
Time domain data set with 300 samples.
Sample time: 0.1 seconds
```

```
Outputs      Unit (if specified)
   y1
```

```
Inputs       Unit (if specified)
   u1
```

```
plot(z1t,z1)
```

Input-Output Data
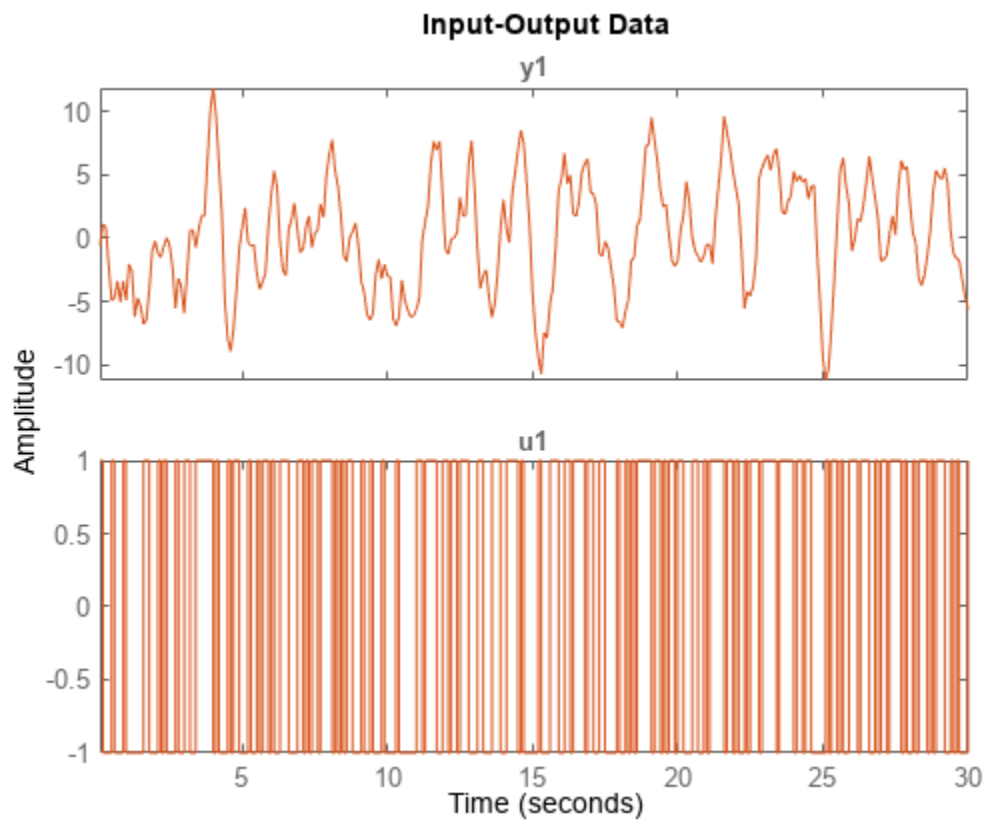
The signals line up precisely.

Estimate second-order state-space models for `z1` and `z1t`.

```
sys1 = ssest(z1,2);
sys1t = ssest(z1t,2);
compare(z1,sys1,sys1t)
```
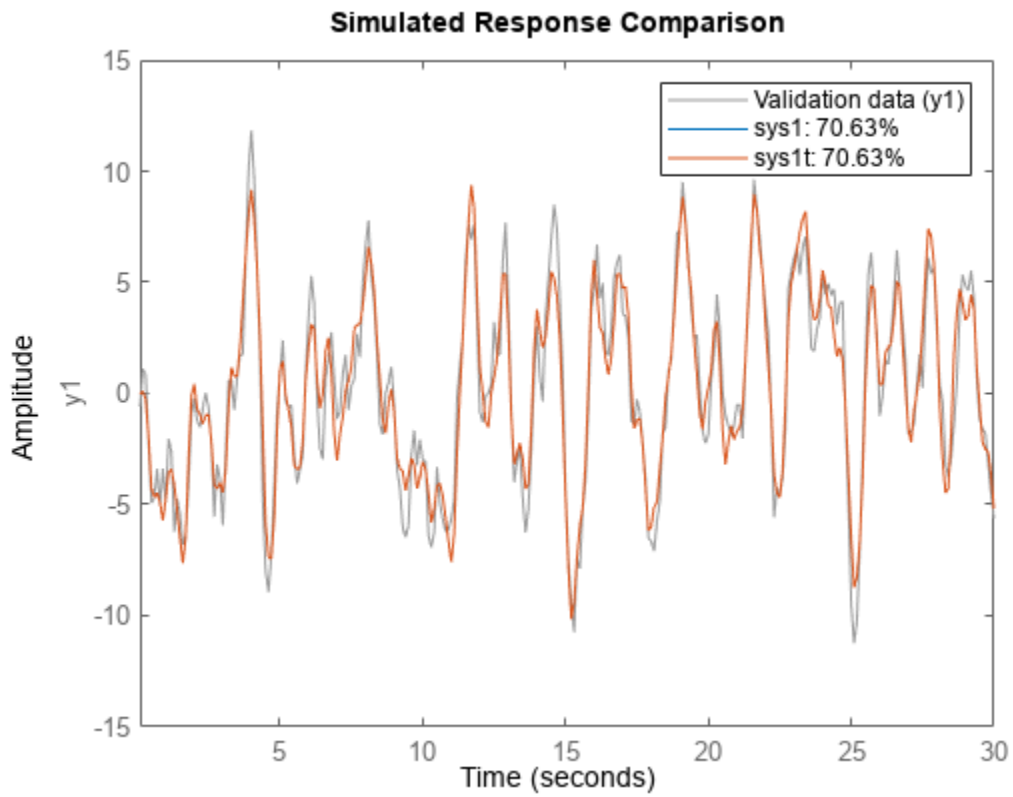
Estimate a state-space model for `z1f`.

```
sys1f = ssest(z1f,2);
compare(z1f,sys1f)
```
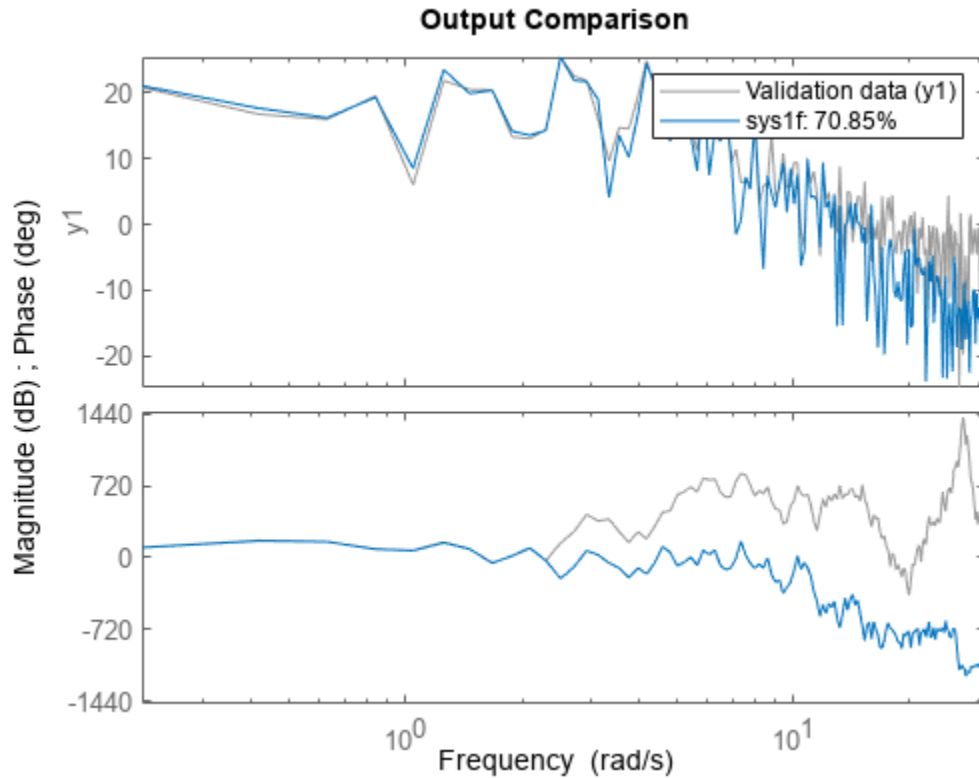
The fit percentages for the time-domain and frequency-domain models are similar.

See Also

`fft` | `iddata` | `ifft` | `etfe` | `spa` | `spafdr`

More About

- "Representing Data in MATLAB Workspace" on page 2-8
- "Supported Data Transformations" on page 3-2
- "Transforming Between Frequency-Domain and Frequency-Response Data" on page 3-14
- "Transform Time-Domain Data in the App" on page 3-3
- "Transform Frequency-Domain Data in the App" on page 3-5
- "Transform Frequency-Response Data in the App" on page 3-6

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 2-11.

When you select to transform single-input/single-output (SISO) frequency-response data to frequency-domain data, the toolbox creates outputs that equal the frequency responses, and inputs equal to 1. Therefore, the ratio between the Fourier transform of the output and the Fourier transform of the input is equal to the system frequency response.

For information about changing the frequency resolution of frequency-response data to a new constant or variable (frequency-dependent) resolution, see the `spafdr` reference page. You might use this feature to increase the number of data points near the system resonance frequencies and make the frequency vector coarser in the region outside the system dynamics. Typically, high-frequency noise dominates away from frequencies where interesting system dynamics occur.

Note You cannot transform an `idfrd` object to a time-domain `iddata` object.

To transform an `idfrd` object with the name `idfrdobj` to a frequency-domain `iddata` object, use the following syntax:

```
dataf = iddata(idfrdobj)
```

The resulting frequency-domain `iddata` object contains values at the same frequencies as the original `idfrd` object.

For the multiple-input case, the toolbox represents frequency-response data as if each input contributes independently to the entire output of the system and then combines information. For example, if a system has three inputs, `u1`, `u2`, and `u3` and two frequency samples, the input matrix is set to:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

In general, for `nu` inputs and `ns` samples, the input matrix has `nu` columns and (`ns`· `nu`) rows.

If you have `ny` outputs, the transformation operation produces an output matrix has `ny` columns and (`ns`· `nu`) rows using the values in the complex frequency response *G(iw)* matrix (`ny`-by-`nu`-by-`ns`). In this example, `y1` is determined by unfolding `G(1,1,:)`, `G(1,2,:)`, and `G(1,3,:)` into three column vectors and vertically concatenating these vectors into a single column. Similarly, `y2` is determined by unfolding `G(2,1,:)`, `G(2,2,:)`, and `G(2,3,:)` into three column vectors and vertically concatenating these vectors.

If you are working with multiple inputs, you also have the option of storing the contribution by each input as an independent experiment in a multiexperiment data set. To transform an `idfrd` object

with the name `idfrdobj` to a multiexperiment data set `datf`, where each experiment corresponds to each of the inputs in `idfrdobj`

```
datf = iddata(idfrdobj,'me')
```

In this example, the additional argument `'me'` specifies that multiple experiments are created.

By default, transformation from frequency-response to frequency-domain data strips away frequencies where the response is `inf` or `NaN`. To preserve the entire frequency vector, use `datf = iddata(idfrdobj,'inf')`. For more information, type `help idfrd/iddata`.

See Also

Related Examples

- "Transform Between Time-Domain and Frequency-Domain Data" on page 3-8
- "Transform Time-Domain Data in the App" on page 3-3
- "Transform Frequency-Domain Data in the App" on page 3-5
- "Transform Frequency-Response Data in the App" on page 3-6

More About

- "Representing Data in MATLAB Workspace" on page 2-8
- "Supported Data Transformations" on page 3-2

4

Linear Model Identification

Black-Box Modeling

Selecting Black-Box Model Structure and Order

Black-box modeling is useful when your primary interest is in fitting the data regardless of a particular mathematical structure of the model. The toolbox provides several linear and nonlinear black-box model structures, which have traditionally been useful for representing dynamic systems. These model structures vary in complexity depending on the flexibility you need to account for the dynamics and noise in your system. You can choose one of these structures and compute its parameters to fit the measured response data.

Black-box modeling is usually a trial-and-error process, where you estimate the parameters of various structures and compare the results. Typically, you start with the simple linear model structure and progress to more complex structures. You might also choose a model structure because you are more familiar with this structure or because you have specific application needs.

The simplest linear black-box structures require the fewest options to configure:

- Transfer function on page 8-2, with a given number of poles and zeros
- Linear ARX model on page 6-3, which is the simplest input-output polynomial model
- State-space model on page 7-2, which you can estimate by specifying the number of model states

Estimation of some of these structures also uses noniterative estimation algorithms, which further reduces complexity.

You can configure a model structure using the model order. The definition of model order varies depending on the type of model you select. For example, if you choose a transfer function representation, the model order is related to the number of poles and zeros. For state-space representation, the model order corresponds to the number of states. In some cases, such as for linear ARX and state-space model structures, you can estimate the model order from the data.

If the simple model structures do not produce good models, you can select more complex model structures by:

- Specifying a higher model order for the same linear model structure. A higher model order increases the model flexibility for capturing complex phenomena. However, an unnecessarily high order can make the model less reliable.
- Explicitly modeling the noise by including the $He(t)$ term, as shown in the following equation.

  $y(t) = Gu(t) + He(t)$

  Here, 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 1-15.

- Using a nonlinear model structure.

  Nonlinear models have more flexibility in capturing complex phenomena than linear models of similar orders. See "Nonlinear Model Structures" on page 11-6.

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, a best practice is to 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 through 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 the current and voltage and building a linear model that describes the relationship between power and temperature.

If you cannot determine variable transformations that yield a linear relationship between input and output variables, you can use nonlinear structures such as nonlinear ARX or Hammerstein-Wiener models. For a list of supported nonlinear model structures and when to use them, see "Nonlinear Model Structures" on page 11-6.

Black-Box Estimation Example

You can use the System Identification app or commands to estimate linear and nonlinear models of various structures. In most cases, you choose a model structure and estimate the model parameters using a single command.

Consider the mass-spring-damper system described in "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)
```

Here, `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 `oe` 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. 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)
```

Here, the second argument 2 represents the order, or the number of states in the model.

In black-box modeling, you do not need the equation of motion for the system — only a guess of the model orders.

For more information about building models, see "Steps for Using the System Identification App" on page 21-2 and "Model Estimation Commands" on page 1-31.

Refine Linear Parametric Models

When to Refine Models

There are two situations where you can refine estimates of linear parametric models.

In the first situation, you have already estimated a parametric model and wish to update the values of its free parameters to improve the fit to the estimation data. This is useful if your previous estimation terminated because of search algorithm constraints such as maximum number of iterations or function evaluations allowed reached. However, if your model captures the essential dynamics, it is usually not necessary to continue improving the fit—especially when the improvement is a fraction of a percent.

In the second situation, you might have constructed a model using one of the model constructors described in "Commands for Constructing Linear Model Structures" on page 1-16. In this case, you built initial parameter guesses into the model structure and wish to refine these parameter values.

What You Specify to Refine a Model

When you refine a model, you must provide two inputs:

- Parametric model
- Data — You can either use the same data set for refining the model as the one you originally used to estimate the model, or you can use a different data set.

Refine Linear Parametric Models Using System Identification App

The following procedure assumes that the model you want to refine is already in the System Identification app. You might have estimated this model in the current session or imported the model from the MATLAB workspace. For information about importing models into the app, see "Importing Models into the App" on page 21-5.

To refine your model:

1 In the System Identification app, verify that you have the correct data set in the **Working Data** area for refining your model.

   If you are using a different data set than the one you used to estimate the model, drag the correct data set into the **Working Data** area. For more information about specifying estimation data, see "Specify Estimation and Validation Data in the App" on page 2-22.

2 Select **Estimate** > **Refine Existing Models** to open the Model Refinement dialog box.
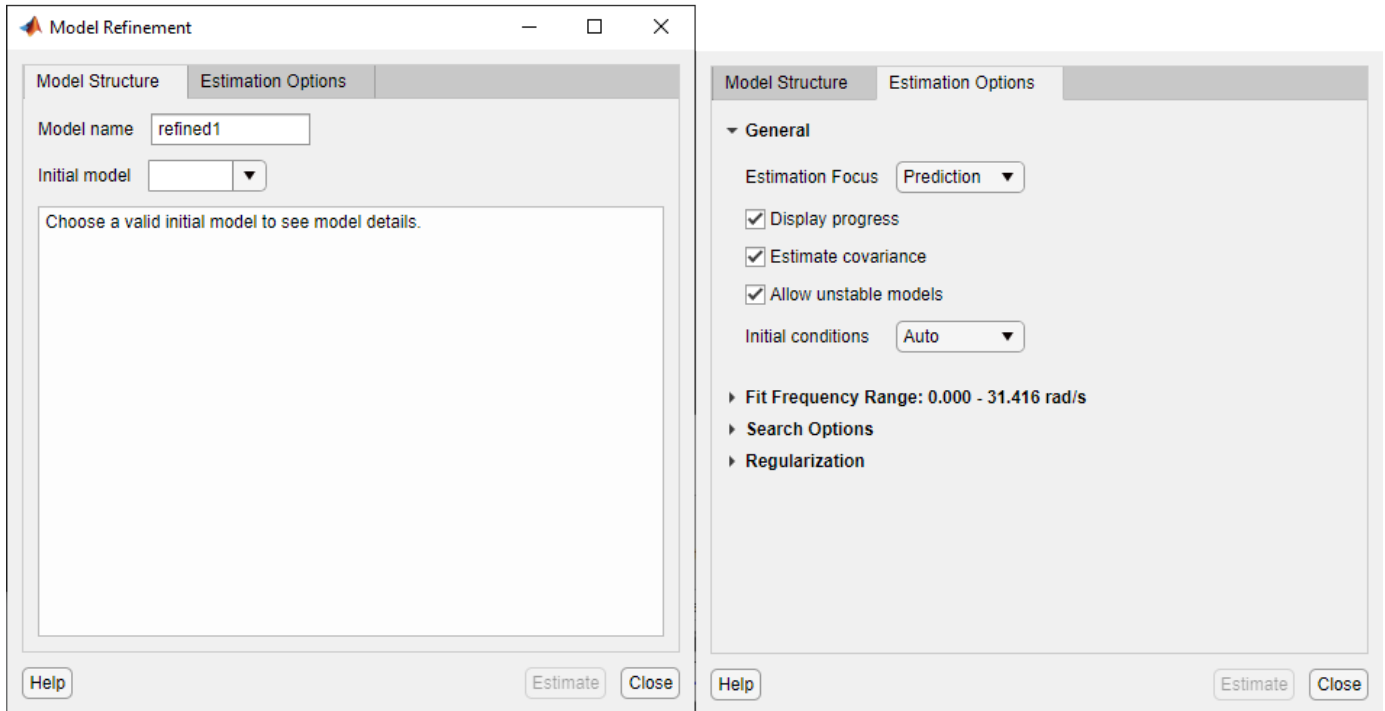
For more information on the options in the dialog box, click **Help**.

3 Select the model you want to refine in the `Initial model` drop-down list or type the model name.

The model name must be in the Model Board of the System Identification app or a variable in the MATLAB workspace. The model can be a state-space, polynomial, process, transfer function or linear grey-box model. The input-output dimensions of the model must match that of the working data.

4 (Optional) Modify the options in the **Estimation Options** tab.

When you enter the model name, the estimation options in the Model Refinement dialog box override the initial model settings.

- Expand **Fit Frequency Range** to specify a frequency range over which to fit the data.
- Expand **Search options** to specify a search method and other search criteria.
- Expand **Regularization** to obtain regularized estimates of model parameters. Specify the regularization constants in the Regularization Options dialog box. To learn more, see "Regularized Estimates of Model Parameters" on page 1-34.

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

6 Validate the new model. See "Ways to Validate Models" on page 17-2.

Refine Linear Parametric Models at the Command Line

If you are working at the command line, you can use `pem` to refine parametric model estimates. You can also use the various model-structure specific estimators — `ssest` for `idss` models, `polyest` for `idpoly` models, `tfest` for `idtf` models, and `greyest` for `idgrey` models.

The general syntax for refining initial models is as follows:

```
m = pem(data,init_model)
```

`pem` uses the properties of the initial model.

You can also specify the estimation options configuring the objective function and search algorithm settings. For more information, see the reference page of the estimating function.

Refine ARMAX Model with Initial Parameter Guesses at Command Line

This example shows how to refine models for which you have initial parameter guesses.

Estimate an ARMAX model for the data by initializing the A, B, and C polynomials. You must first create a model object and set the initial parameter values in the model properties. Next, you provide this initial model as input to `armax`, `polyest`, or `pem`, which refine the initial parameter guesses using the data.

Load estimation data.

```
load iddata8
```

Define model parameters.

Leading zeros in B indicate input delay (nk), which is 1 for each input channel.

```
A = [1 -1.2 0.7];
B{1} = [0 1 0.5 0.1]; % first input
B{2} = [0 1.5 -0.5]; % second input
B{3} = [0 -0.1 0.5 -0.1]; % third input
C = [1 0 0 0 0];
Ts = 1;
```

Create model object.

```
init_model = idpoly(A,B,C,'Ts',1);
```

Use polyest to update the parameters of the initial model.

```
model = polyest(z8,init_model);
```

Compare the two models.

```
compare(z8,init_model,model)
```
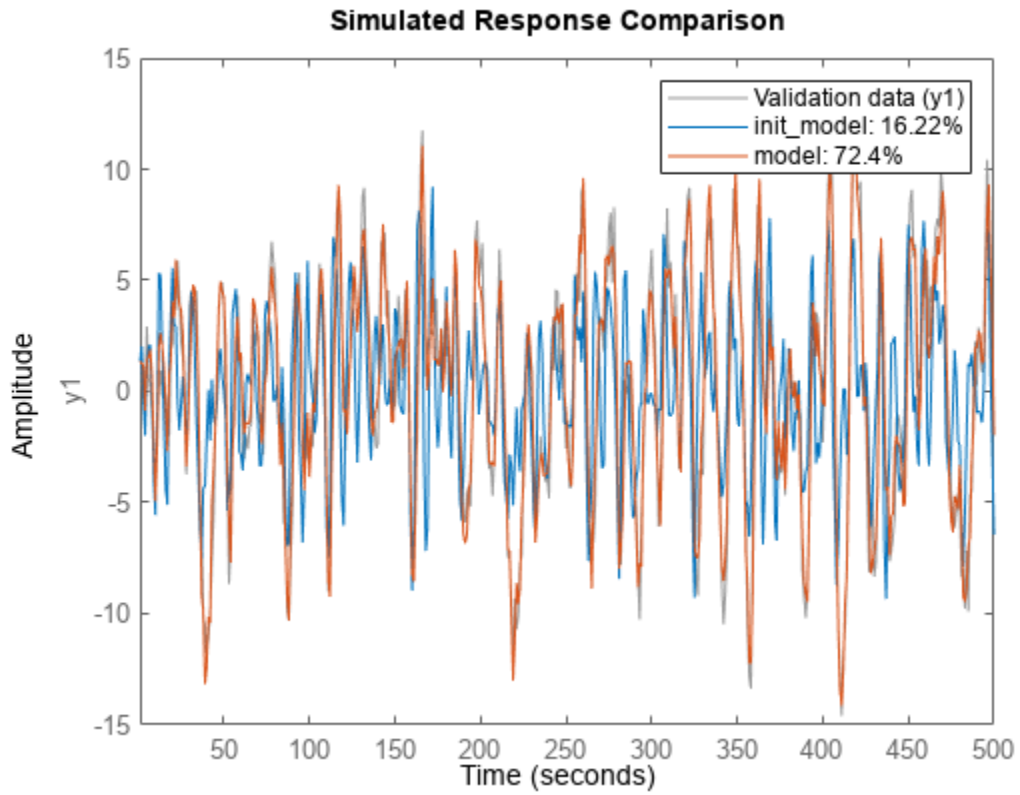
See Also
"Input-Output Polynomial Models"

Refine Initial ARMAX Model at Command Line

This example shows how to estimate an initial model and refine it using `pem`.

Load measured data.

```
load iddata8
```

Split the data into an initial estimation data set and a refinement data set.

```
init_data = z8(1:100);
refine_data = z8(101:end);
```

`init_data` is an `iddata` object containing the first 100 samples from `z8` and `refine_data` is an `iddata` object representing the remaining data in `z8`.

Estimate an ARMAX model.

```
na = 4;
nb = [3 2 3];
nc = 2;
nk = [0 0 0];

sys = armax(init_data,[na nb nc nk]);
```

`armax` uses the default algorithm properties to estimate `sys`.

Refine the estimated model by specifying the estimation algorithm options. Specify stricter tolerance and increase the maximum iterations.

```
opt = armaxOptions;
opt.SearchOptions.Tolerance = 1e-5;
opt.SearchOptions.MaxIterations = 50;

refine_sys = pem(refine_data,sys,opt);
```

Compare the fit of the initial and refined models.

```
compare(refine_data,sys,refine_sys)
```

Simulated Response Comparison
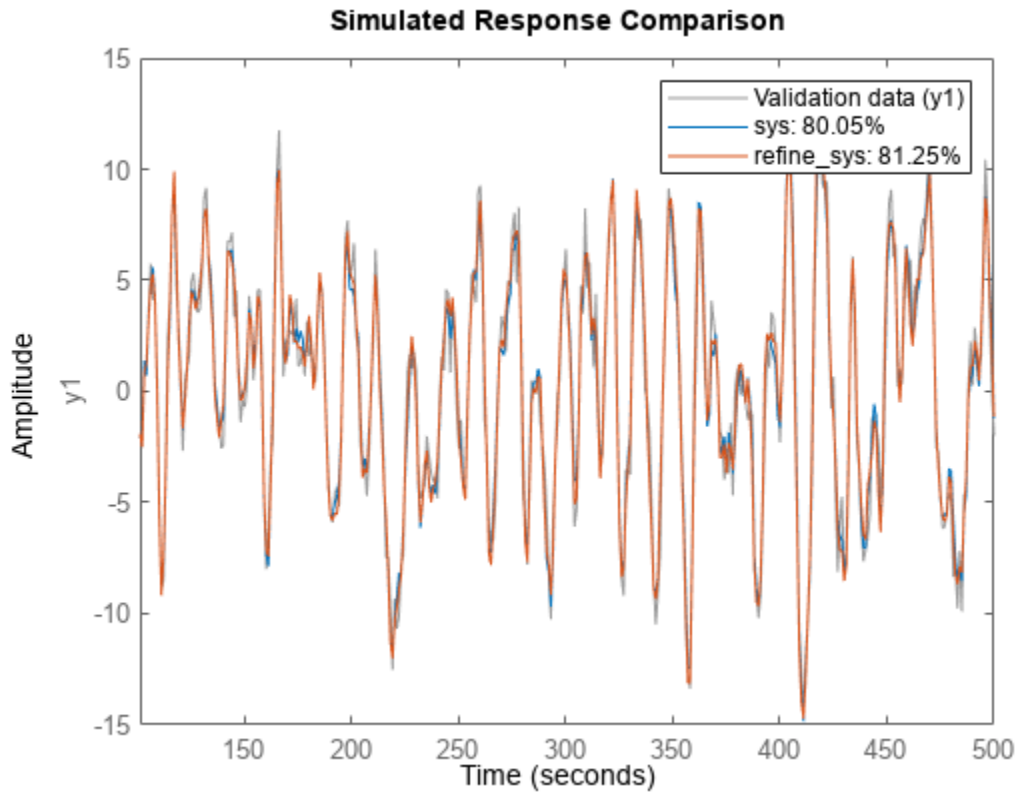
refine_sys provides a closer fit to the data than sys.

You can similarly use polyest or armax to refine the estimated model.

See Also

Functions
armax | pem | polyest

Extracting Numerical Model Data

You can extract the following numerical data from linear model objects:

- Coefficients and uncertainty

  For example, extract state-space matrices (A, B, C, D and K) for state-space models, or polynomials (A, B, C, D and F) for polynomial models.

  If you estimated model uncertainty data, this information is stored in the model in the form of the parameter covariance matrix. You can fetch the covariance matrix (in its raw or factored form) using the `getcov` command. The covariance matrix represents uncertainties in parameter estimates and is used to compute:

  - Confidence bounds on model output plots, Bode plots, residual plots, and pole-zero plots
  - Standard deviation in individual parameter values. For example, one standard deviation in the estimated value of the A polynomial in an ARX model, returned by the `polydata` command and displayed by the `present` command.

  The following table summarizes the commands for extracting model coefficients and uncertainty.

Commands for Extracting Model Coefficients and Uncertainty Data

Command	Description	Syntax
freqresp	Extracts frequency-response data (H) and corresponding covariance (CovH) from any linear identified model.	`[H,w,CovH] = freqresp(m)`
polydata	Extracts polynomials (such as A) from any linear identified model. The polynomial uncertainties (such as dA) are returned only for idpoly models.	`[A,B,C,D,F,dA,dB,dC,dD,dF] = ...` `        polydata(m)`

idssdata	Extracts state-space matrices (such as A) from any linear identified model. The matrix uncertainties (such as dA) are returned only for idss models.	`[A,B,C,D,K,X0,...`
		` dA,dB,dC,dD,dK,dX0] = ...`
		`        idssdata(m)`

tfdata	Extracts numerator and denominator polynomials (Num, Den) and their uncertainties (dnum, dden) from any linear identified model.	`[Num,Den,Ts,dNum,dDen] = ...` `    tfdata(m)`
zpkdata	Extracts zeros, poles, and gains (Z, P, K) and their covariances (covZ, covP, covK) from any linear identified model.	`[Z,P,K,Ts,covZ,covP,covK] = ...` `    zpkdata(m)`
getpvec	Obtain a list of model parameters and their uncertainties. To access parameter attributes such as values, free status, bounds or labels, use getpar.	`pvec = getpvec(m)`
getcov	Obtain parameter covariance information	`cov_data = getcov(m)`

You can also extract numerical model data by using dot notation to access model properties. For example, m.A displays the *A* polynomial coefficients from model m. Alternatively, you can use the get command, as follows: get(m,'A').

Tip To view a list of model properties, type get(model).

- Dynamic and noise models

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

$y = Gu + He$

G is an operator that takes the measured inputs *u* to the outputs and captures the system dynamics, also called the *measured model*. *H* is an operator that describes the properties of the additive output disturbance and takes the hypothetical (unmeasured) noise source inputs *e* to the outputs, also called the *noise model*. When you estimate a noise model, the toolbox includes one noise channel *e* for each output in your system.

You can operate on extracted model data as you would on any other MATLAB vectors, matrices and cell arrays. You can also pass these numerical values to Control System Toolbox commands, for example, or Simulink blocks.

Transforming Between Discrete-Time and Continuous-Time Representations

Why Transform Between Continuous and Discrete Time?

Transforming between continuous-time and discrete-time representations is useful, for example, if you have estimated a discrete-time linear model and require a continuous-time model instead for your application.

You can use c2d and d2c to transform any linear identified model between continuous-time and discrete-time representations. d2d is useful is you want to change the sample time of a discrete-time model. All of these operations change the sample time, which is called *resampling* the model.

These commands do not transform the estimated model uncertainty. If you want to translate the estimated parameter covariance during the conversion, use translatecov.

Note c2d and d2d correctly approximate the transformation of the noise model only when the sample time T is small compared to the bandwidth of the noise.

Using the c2d, d2c, and d2d Commands

The following table summarizes the commands for transforming between continuous-time and discrete-time model representations.

Command	Description	Usage Example
c2d	Converts continuous-time models to discrete-time models. You cannot use c2d for idproc models and for idgrey models whose FunctionType is not 'cd'. Convert these models into idpoly, idtf, or idss models before calling c2d.	To transform a continuous-time model mod_c to a discrete-time form, use the following command: mod_d = c2d(mod_c,T) where T is the sample time of the discrete-time model.
d2c	Converts parametric discrete-time models to continuous-time models. You cannot use d2c for idgrey models whose FunctionType is not 'cd'. Convert these models into idpoly, idtf, or idss models before calling d2c.	To transform a discrete-time model mod_d to a continuous-time form, use the following command: mod_c = d2c(mod_d)

Command	Description	Usage Example
d2d	Resample a linear discrete-time model and produce an equivalent discrete-time model with a new sample time. You can use the resampled model to simulate or predict output with a specified time interval.	To resample a discrete-time model `mod_d1` to a discrete-time form with a new sample time `Ts`, use the following command: `mod_d2 = d2d(mod_d1,Ts)`

The following commands compare estimated model `m` and its continuous-time counterpart `mc` on a Bode plot:

```
% Estimate discrete-time ARMAX model
% from the data
m = armax(data,[2 3 1 2]);
% Convert to continuous-time form
mc = d2c(m);
% Plot bode plot for both models
bode(m,mc)
```

Specifying Intersample Behavior

A sampled signal is characterized only by its values at the sampling instants. However, when you apply a continuous-time input to a continuous-time system, the output values at the sampling instants depend on the inputs at the sampling instants and on the inputs between these points. Thus, the `InterSample` data property describes how the algorithms should handle the input between samples. For example, you can specify the behavior between the samples to be piece-wise constant (zero-order hold, `zoh`) or linearly interpolated between the samples (first order hold, `foh`). The transformation formulas for `c2d` and `d2c` are affected by the intersample behavior of the input.

By default, `c2d` and `d2c` use the intersample behavior you assigned to the estimation data. To override this setting during transformation, add an extra argument in the syntax. For example:

```
% Set first-order hold intersample behavior
mod_d = c2d(mod_c,T,'foh')
```

Effects on the Noise Model

`c2d`, `d2c`, and `d2d` change the sample time of both the dynamic model and the noise model. Resampling a model affects the variance of its noise model.

A parametric noise model is a time-series model with the following mathematical description:

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

$$Ee^2 = \lambda$$

The noise spectrum is computed by the following discrete-time equation:

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

where λ is the variance of the white noise *e(t)*, and λT represents the spectral density of *e(t)*. Resampling the noise model preserves the spectral density λT. The spectral density λT is invariant up to the Nyquist frequency. For more information about spectrum normalization, see "Spectrum Normalization" on page 9-10.

`d2d` resampling of the noise model affects simulations with noise using `sim`. If you resample a model to a faster sampling rate, simulating this model results in higher noise level. This higher noise level results from the underlying continuous-time model being subject to continuous-time white noise disturbances, which have infinite, instantaneous variance. In this case, the *underlying continuous-time model* is the unique representation for discrete-time models. To maintain the same level of noise after interpolating the noise signal, scale the noise spectrum by $\sqrt{T_{New}/T_{Old}}$, where T_{new} is the new sample time and T_{old} is the original sample time. before applying `sim`.

See Also

More About
*   "Continuous-Discrete Conversion Methods" on page 4-18

Continuous-Discrete Conversion Methods

System Identification Toolbox offers several discretization and interpolation methods for converting identified dynamic system models between continuous time and discrete time and for resampling discrete-time models. Some methods tend to provide a better frequency-domain match between the original and converted systems, while others provide a better match in the time domain. Use the following table to help select the method that is best for your application.

Discretization Method	Use When
"Zero-Order Hold" on page 4-18	You want an exact discretization in the time domain for staircase inputs.
"First-Order Hold" on page 4-20	You want an exact discretization in the time domain for piecewise linear inputs.
"Impulse-Invariant Mapping" on page 4-20 (continuous-to-discrete conversion only)	You want an exact discretization in the time domain for impulse train inputs.
"Tustin Approximation" on page 4-21	• You want good matching in the frequency domain between the continuous- and discrete-time models. • Your model has important dynamics at some particular frequency.
"Zero-Pole Matching Equivalents" on page 4-24	• You have a SISO model. • You want good matching in the frequency domain between the continuous- and discrete-time models.
"Least Squares" (Control System Toolbox) (continuous-to-discrete conversion only)	• You have a SISO model. • You want good matching in the frequency domain between the continuous- and discrete-time models. • You want to capture fast system dynamics but must use a larger sample time.

For information about how to specify a conversion method at the command line, see `c2d`, `d2c`, and `d2d`. You can experiment interactively with different discretization methods in the Live Editor using the **Convert Model Rate** task (requires a license).

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 $H(s)$.
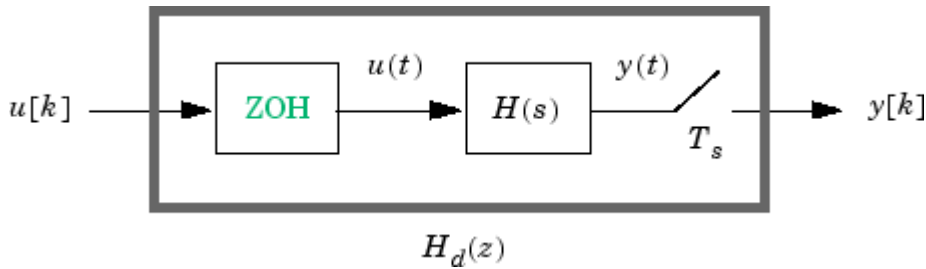
The ZOH block generates the continuous-time input signal $u(t)$ by holding each sample value $u(k)$ constant over one sample period:

$$u(t) = u[k], \qquad kT_s \leq t \leq (k+1)T_s$$

The signal $u(t)$ is the input to the continuous system $H(s)$. The output $y[k]$ results from sampling $y(t)$ every T_s seconds.

Conversely, given a discrete system $H_d(z)$, d2c produces a continuous system $H(s)$. The ZOH discretization of $H(s)$ coincides with $H_d(z)$.

The ZOH discrete-to-continuous conversion has the following limitations:

- d2c cannot convert LTI models with poles at $z = 0$.
- For discrete-time LTI models having negative real poles, ZOH d2c conversion produces a continuous system with higher order. The model order increases because a negative real pole in the z domain maps to a pure imaginary value in the s domain. Such mapping results in a continuous-time model with complex data. To avoid this issue, the software instead introduces a conjugate pair of complex poles in the s domain.

ZOH Method for Systems with Time Delays

You can use the ZOH method to discretize SISO or MIMO continuous-time models with time delays. The ZOH method yields an exact discretization for systems with input delays, output delays, or transport delays.

For systems with internal delays (delays in feedback loops), the ZOH method results in approximate discretizations. The following figure illustrates a system with an internal delay.
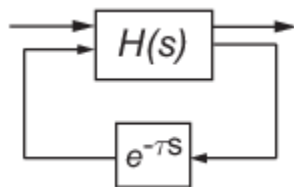
For such systems, c2d performs the following actions to compute an approximate ZOH discretization:

1. Decomposes the delay τ as $\tau = kT_s + \rho$ with $0 \leq \rho < T_s$.
2. Absorbs the fractional delay ρ into $H(s)$.
3. Discretizes $H(s)$ to $H(z)$.
4. Represents the integer portion of the delay kT_s as an internal discrete-time delay z^{-k}. The final discretized model appears in the following figure:
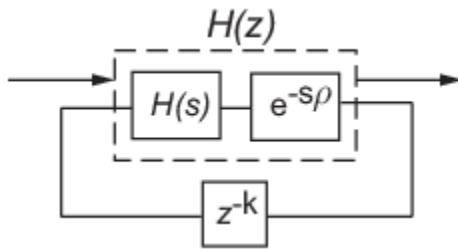
First-Order Hold

The First-Order Hold (FOH) method provides an exact match between the continuous- and discrete-time systems in the time domain for piecewise linear inputs.

FOH differs from ZOH by the underlying hold mechanism. To turn the input samples $u[k]$ into a continuous input $u(t)$, FOH uses linear interpolation between samples:

$$u(t) = u[k] + \frac{t - kT_s}{T_s}(u[k + 1] - u[k]), \quad kT_s \le t \le (k + 1)T_s$$

In general, this method is more accurate than ZOH for systems driven by smooth inputs.

This FOH method differs from standard causal FOH and is more appropriately called triangle approximation (see [2], p. 228). The method is also known as ramp-invariant approximation.

FOH Method for Systems with Time Delays

You can use the FOH method to discretize SISO or MIMO continuous-time models with time delays. The FOH method handles time delays in the same way as the ZOH method. See "ZOH Method for Systems with Time Delays" on page 4-19.

Impulse-Invariant Mapping

The impulse-invariant mapping produces a discrete-time model with the same impulse response as the continuous time system. For example, compare the impulse response of a first-order continuous system with the impulse-invariant discretization:

```
G = tf(1,[1,1]);
Gd1 = c2d(G,0.01,'impulse');
impulse(G,Gd1)
```
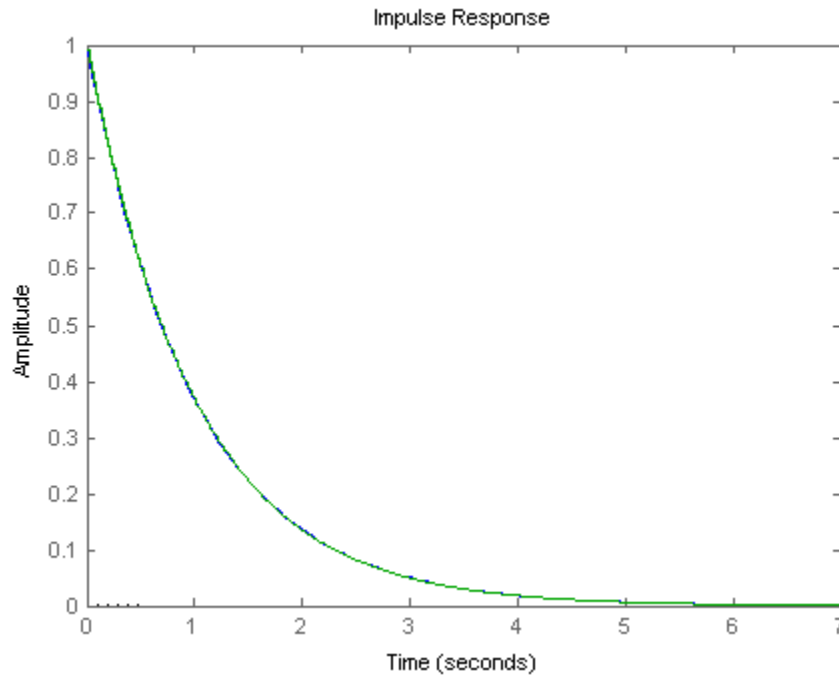
The impulse response plot shows that the impulse responses of the continuous and discretized systems match.

Impulse-Invariant Mapping for Systems with Time Delays

You can use impulse-invariant mapping to discretize SISO or MIMO continuous-time models with time delays, except that the method does not support `ss` models with internal delays. For supported models, impulse-invariant mapping yields an exact discretization of the time delay.

Tustin Approximation

The Tustin or bilinear approximation yields the best frequency-domain match between the continuous-time and discretized systems. This method relates the *s*-domain and *z*-domain transfer functions using the approximation:

$$z = e^{sT_s} \approx \frac{1 + sT_s/2}{1 - sT_s/2}.$$

In `c2d` conversions, the discretization $H_d(z)$ of a continuous transfer function $H(s)$ is:

$$H_d(z) = H(s'), \qquad s' = \frac{2}{T_s} \frac{z - 1}{z + 1}$$

Similarly, the `d2c` conversion relies on the inverse correspondence

$$H(s) = H_d(z'), \qquad z' = \frac{1 + sT_s/2}{1 - sT_s/2}$$

When you convert a state-space model using the Tustin method, the states are not preserved. The state transformation depends upon the state-space matrices and whether the system has time delays. For example, for an explicit ($E = I$) continuous-time model with no time delays, the state vector $w[k]$ of the discretized model is related to the continuous-time state vector $x(t)$ by:

$$w[kT_s] = \left(I - A\frac{T_s}{2}\right)x(kT_s) - \frac{T_s}{2}Bu(kT_s) = x(kT_s) - \frac{T_s}{2}(Ax(kT_s) + Bu(kT_s)).$$

T_s is the sample time of the discrete-time model. A and B are state-space matrices of the continuous-time model.

The Tustin approximation is not defined for systems with poles at $z = -1$ and is ill-conditioned for systems with poles near $z = -1$.

Tustin Approximation with Frequency Prewarping

If your system has important dynamics at a particular frequency that you want the transformation to preserve, you can use the Tustin method with frequency prewarping. This method ensures a match between the continuous- and discrete-time responses at the prewarp frequency.

The Tustin approximation with frequency prewarping uses the following transformation of variables:

$$H_d(z) = H(s'), \qquad s' = \frac{\omega}{\tan(\omega T_s/2)}\frac{z-1}{z+1}$$

This change of variable ensures the matching of the continuous- and discrete-time frequency responses at the prewarp frequency ω, because of the following correspondence:

$$H(j\omega) = H_d\left(e^{j\omega T_s}\right)$$

Tustin Approximation for Systems with Time Delays

You can use the Tustin approximation to discretize SISO or MIMO continuous-time models with time delays.

By default, the Tustin method rounds any time delay to the nearest multiple of the sample time. Therefore, for any time delay `tau`, the integer portion of the delay, `k*Ts`, maps to a delay of `k` sampling periods in the discretized model. This approach ignores the residual fractional delay, `tau - k*Ts`.

You can to approximate the fractional portion of the delay by a discrete all-pass filter (Thiran filter) of specified order. To do so, use the `FractDelayApproxOrder` option of `c2dOptions`.

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.
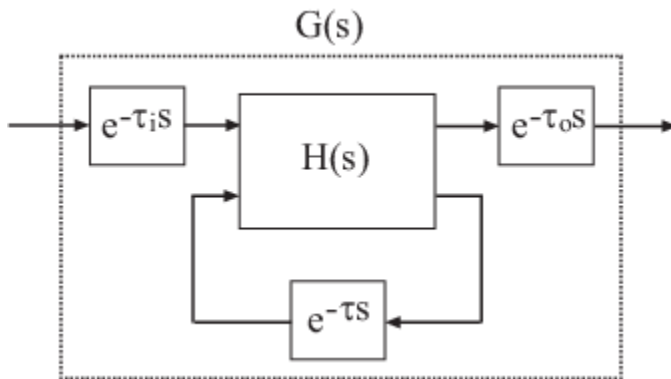
The following figure shows the general result of discretizing $G(s)$ using the Tustin method.
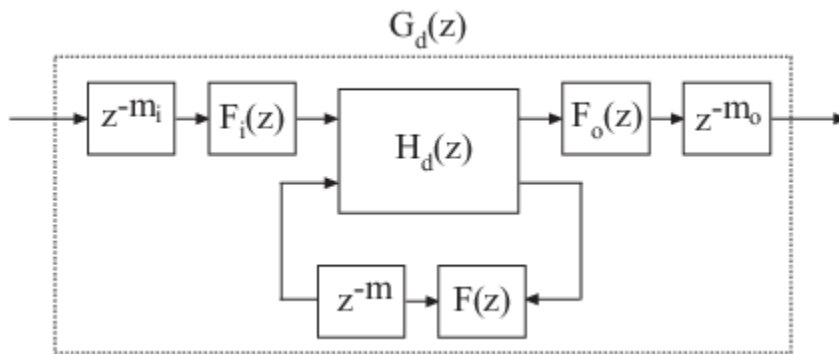
By default, c2d converts the time delays to pure integer time delays. The c2d command computes the integer delays by rounding each time delay to the nearest multiple of the sample time T_s. Thus, in the default case, $m_i = \text{round}(\tau_i/T_s)$, $m_o = \text{round}(\tau_o/T_s)$, and $m = \text{round}(\tau/T_s)$.. 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:

$\text{order}(F_i(z)) = \text{max}(\text{ceil}(\tau_i/T_s), \text{FractDelayApproxOrder})$.

If ceil(τ_i/T_s) < FractDelayApproxOrder, the Thiran filter $F_i(z)$ approximates the entire input delay τ_i. If ceil(τ_i/T_s) > FractDelayApproxOrder, the Thiran filter only approximates a portion of the input delay. In that case, c2d represents the remainder of the input delay as a chain of unit delays z^{-m_i}, where

$m_i = $ ceil(τ_i/T_s) – FractDelayApproxOrder

c2d uses Thiran filters and FractDelayApproxOrder in a similar way to approximate the output delay τ_o and the internal delay τ.

When you discretizetf and zpk models using the Tustin method, c2d first aggregates all input, output, and transport delays into a single transport delay τ_{TOT} for each channel. c2d then

4-23

approximates τ_{TOT} as a Thiran filter and a chain of unit delays in the same way as described for each of the time delays in `ss` models.

For more information about Thiran filters, see the `thiran` reference page and [4].

Zero-Pole Matching Equivalents

This method of conversion, which computes zero-pole matching equivalents, applies only to SISO systems. The continuous and discretized systems have matching DC gains. Their poles and zeros are related by the transformation:

$$z_i = e^{s_i T_s}$$

where:

- z_i is the ith pole or zero of the discrete-time system.
- s_i is the ith pole or zero of the continuous-time system.
- T_s is the sample time.

See [2] for more information.

Zero-Pole Matching for Systems with Time Delays

You can use zero-pole matching to discretize SISO continuous-time models with time delay, except that the method does not support `ss` models with internal delays. The zero-pole matching method handles time delays in the same way as the Tustin approximation. See "Tustin Approximation for Systems with Time Delays" on page 4-22.

Least Squares

The least squares method minimizes the error between the frequency responses of the continuous-time and discrete-time systems up to the Nyquist frequency using a vector-fitting optimization approach. This method is useful when you want to capture fast system dynamics but must use a larger sample time, for example, when computational resources are limited.

This method is supported only by the `c2d` function and only for SISO systems.

As with Tustin approximation and zero-pole matching, the least squares method provides a good match between the frequency responses of the original continuous-time system and the converted discrete-time system. However, when using the least squares method with:

- The same sample time as Tustin approximation or zero-pole matching, you get a smaller difference between the continuous-time and discrete-time frequency responses.
- A lower sample time than what you would use with Tustin approximation or zero-pole matching, you can still get a result that meets your requirements. Doing so is useful if computational resources are limited, since the slower sample time means that the processor must do less work.

References

[1] Åström, K.J. and B. Wittenmark, *Computer-Controlled Systems: Theory and Design*, Prentice-Hall, 1990, pp. 48-52.

[2] Franklin, G.F., Powell, D.J., and Workman, M.L., *Digital Control of Dynamic Systems* (3rd Edition), Prentice Hall, 1997.

[3] Smith, J.O. III, "Impulse Invariant Method", *Physical Audio Signal Processing*, August 2007. `https://www.dsprelated.com/dspbooks/pasp/Impulse_Invariant_Method.html`.

[4] T. Laakso, V. Valimaki, "Splitting the Unit Delay", *IEEE Signal Processing Magazine*, Vol. 13, No. 1, p.30-60, 1996.

See Also

Functions
c2d | d2c | c2dOptions | d2cOptions | d2d | d2dOptions | thiran

Live Editor Tasks
Convert Model Rate

Effect of Input Intersample Behavior on Continuous-Time Models

The intersample behavior of the input signals influences the estimation, simulation and prediction of continuous-time models. A sampled signal is characterized only by its values at the sampling instants. However, when you apply a continuous-time input to a continuous-time system, the output values at the sampling instants depend on the inputs at the sampling instants and on the inputs between these points.

The `iddata` and `idfrd` objects have an `InterSample` property which stores how the input behaves between the sampling instants. You can specify the behavior between the samples to be piecewise constant (zero-order hold), linearly interpolated between the samples (first-order hold) or band-limited. A band-limited intersample behavior of the input signal means:

- A filtered input signal (an input of finite bandwidth) was used to excite the system dynamics.
- The input was measured using a sampling device (A/D converter with antialiasing) that reported it to be band-limited even though the true input entering the system was piecewise constant or linear. In this case, the sampling devices can be assumed to be a part of the system being modeled.

When the input signal is a band-limited discrete-time frequency-domain data (`iddata` with `domain = 'frequency'` or `idfrd` with sample time *Ts≠0*), the model estimation is performed by treating the data as continuous-time data (*Ts = 0*). For more information, see Pintelon, R. and J. Schoukens, *System Identification. A Frequency Domain Approach*, section 10.2, pp-352-356,Wiley-IEEE Press, New York, 2001.

The intersample behavior of the input data also affects the results of simulation and prediction of continuous-time models. `sim` and `predict` commands use the `InterSample` property to choose the right algorithm for computing model response.

The following example simulates a system using first-order hold (`foh`) intersample behavior for input signal.

```
sys = idtf([-1 -2],[1 2 1 0.5]);
rng('default')
u = idinput([100 1 5],'sine',[],[],[5 10 1]);
Ts = 2;
y = lsim(sys,u,(0:Ts:999)','foh');
```

Create an `iddata` object for the simulated input-output data.

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

The default intersample behavior is zero-order hold (`zoh`).

```
data.InterSample
```

```
ans =
'zoh'
```

Estimate a transfer function using this data.

```
np = 3; % number of poles
nz = 1; % number of zeros
opt = tfestOptions('InitializeMethod','all','Display','on');
```

```
opt.SearchOptions.MaxIterations = 100;
modelZOH = tfest(data,np,nz,opt)

modelZOH =
  From input "u1" to output "y1":
          -217.2 s - 391.6
   --------------------------------
   s^3 + 354.4 s^2 + 140.2 s + 112.4

Continuous-time identified transfer function.

Parameterization:
   Number of poles: 3   Number of zeros: 1
   Number of free coefficients: 5
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using TFEST on time domain data "data".
Fit to estimation data: 81.38%
FPE: 0.1146, MSE: 0.111
```

The model gives about 80% fit to data. The sample time of the data is large enough that intersample inaccuracy (using `zoh` rather than `foh`) leads to significant modeling errors.

Re-estimate the model using `foh` intersample behavior.

```
data.InterSample = 'foh';
modelFOH = tfest(data,np,nz,opt)

modelFOH =
  From input "u1" to output "y1":
          -1.197 s - 0.06843
   -------------------------------------
   s^3 + 0.4824 s^2 + 0.3258 s + 0.01723

Continuous-time identified transfer function.

Parameterization:
   Number of poles: 3   Number of zeros: 1
   Number of free coefficients: 5
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using TFEST on time domain data "data".
Fit to estimation data: 97.7%
FPE: 0.001748, MSE: 0.001693
```

`modelFOH` is able to retrieve the original system correctly.

Compare the model outputs with data.

```
compare(data,modelZOH,modelFOH)
```
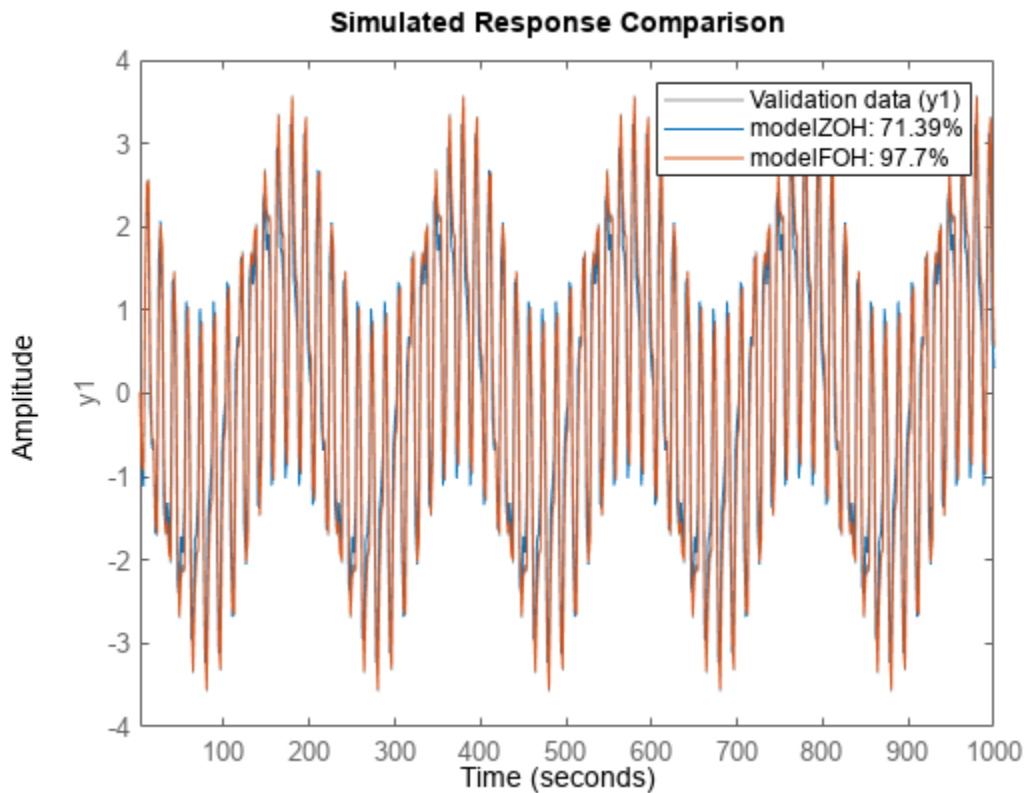
modelZOH is compared to data whose intersample behavior is `foh`. Therefore, its fit decreases to around 70%.

See Also

`iddata` | `idfrd`

More About

- "Frequency Domain Identification: Estimating Models Using Frequency Domain Data" on page 4-52

Transforming Between Linear Model Representations

You can transform linear models between state-space and polynomial forms. You can also transform between frequency-response, state-space, and polynomial forms.

If you used the System Identification app to estimate models, you must export the models to the MATLAB workspace before converting models.

For detailed information about each command in the following table, see the corresponding reference page.

Commands for Transforming Model Representations

Command	Model Type to Convert	Usage Example

`idfrd`	Converts any linear model to an `idfrd` model.
	
	If you have the Control System Toolbox product, this command converts any numeric LTI model too.
		To get frequency response of `m` at default frequencies, use the following command:
		
		`m_f = idfrd(m)`
		
		To get frequency response at specific frequencies, use the following command:
		
		`m_f = idfrd(m,f)`
		
		To get frequency response for a submodel from input 2 to output 3, use the following command:
		
		`m_f = idfrd(m(2,3))`

`idpoly`	Converts any linear identified model, except `idfrd`, to ARMAX representation if the original model has a nontrivial noise component, or OE if the noise model is trivial ($H = 1$). If you have the Control System Toolbox product, this command converts any numeric LTI model, except `frd`.	To get an ARMAX model from state-space model `m_ss`, use the following command: `m_p = idpoly(m_ss)`
`idss`	Converts any linear identified model, except `idfrd`, to state-space representation. If you have the Control System Toolbox product, this command converts any numeric LTI model, except `frd`.	To get a state-space model from an ARX model `m_arx`, use the following command: `m_ss = idss(m_arx)`
`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

What Is Subreferencing?

You can use subreferencing to create models with subsets of inputs and outputs from existing multivariable models. Subreferencing is also useful when you want to generate model plots for only certain channels, such as when you are exploring multiple-output models for input channels that have minimal effect on the output.

The toolbox supports subreferencing operations for `idtf`, `idpoly`, `idproc`, `idss`, and `idfrd` model objects.

Subreferencing is not supported for `idgrey` models. If you want to analyze the sub-model, convert it into an `idss` model first, and then subreference the I/Os of the `idss` model. If you want a grey-box representation of a subset of I/Os, create a new `idgrey` model that uses an ODE function returning the desired I/O dynamics.

In addition to subreferencing the model for specific combinations of measured inputs and output, you can subreference dynamic and noise models individually.

Limitation on Supported Models

Subreferencing nonlinear models is not supported.

Subreferencing Specific Measured Channels

Use the following general syntax to subreference specific input and output channels in models:

```
model(outputs,inputs)
```

In this syntax, `outputs` and `inputs` specify channel indexes or channel names.

To select all output or all input channels, use a colon (`:`). To select no channels, specify an empty matrix (`[]`). If you need to reference several channel names, use a cell array of character vectors.

For example, to create a new model `m2` from `m` from inputs `1` (`'power'`) and `4` (`'speed'`) to output number `3` (`'position'`), use either of the following equivalent commands:

```
m2 = m('position',{'power','speed'})
```

or

```
m2 = m(3,[1 4])
```

For a single-output model, you can use the following syntax to subreference specific input channels without ambiguity:

```
m3 = m(inputs)
```

Similarly, for a single-input model, you can use the following syntax to subreference specific output channels:

```
m4 = m(outputs)
```

Separation of Measured and Noise Components of Models

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

$$y = Gu + He$$

G is an operator that takes the measured inputs *u* to the outputs and captures the system dynamics.

H is an operator that describes the properties of the additive output disturbance and takes the hypothetical (unmeasured) noise source inputs to the outputs. *H* represents the noise model. When you specify to estimate a noise model, the resulting model include one noise channel *e* at the input for each output in your system.

Thus, linear, parametric models represent input-output relationships for two kinds of input channels: measured inputs and (unmeasured) noise inputs. For example, consider the ARX model given by one of the following equations:

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

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.

See Also

More About

- "Treating Noise Channels as Measured Inputs" on page 4-34

Treating Noise Channels as Measured Inputs

A linear models is given by:

$$y = Gu + He$$

Where G is an operator that takes the measured inputs u to the outputs and captures the system dynamics. H is an operator that describes the properties of the additive output disturbance and takes the hypothetical (unmeasured) noise source inputs to the outputs. H represents the noise model. When you specify to estimate a noise model, the resulting model include one noise channel e at the input for each output in your system.

To study noise contributions in more detail, it might be useful to convert the noise channels to measured channels using `noisecnv`:

```
m_GH = noisecnv(m)
```

This operation creates a model `m_GH` that represents both measured inputs `u` and noise inputs e, treating both sources as measured signals. `m_GH` is a model from `u` and e to `y`, describing the transfer functions G and H.

Converting noise channels to measured inputs loses information about the variance of the innovations `e`. For example, step response due to the noise channels does not take into consideration the magnitude of the noise contributions. To include this variance information, normalize e such that v becomes white noise with an identity covariance matrix, where

$$e = Lv$$

To normalize e, use the following command:

```
m_GH = noisecnv(m,'Norm')
```

This command creates a model where u and v are treated as measured signals, as follows:

$$y(t) = Gu(t) + HLv = [G\ HL]\begin{bmatrix} u \\ v \end{bmatrix}$$

For example, the scaling by L causes the step responses from v to y to reflect the size of the disturbance influence.

The converted noise sources are named in a way that relates the noise channel to the corresponding output. Unnormalized noise sources e are assigned names such as `'e@y1'`, `'e@y2'`, ..., `'e@yn'`, where `'e@yn'` refers to the noise input associated with the output `yn`. Similarly, normalized noise sources v, are named `'v@y1'`, `'v@y2'`, ..., `'v@yn'`.

If you want to create a model that has only the noise channels of an identified model as its measured inputs, use the `noise2meas` command. It results in a model with $y(t) = He$ or $y(t) = HLv$, where e or v is treated as a measured input.

Note When you plot models in the app that include noise sources, you can select to view the response of the noise model corresponding to specific outputs. For more information, see "Selecting Measured and Noise Channels in Plots" on page 21-10.

See Also

`noisecnv` | `noise2meas`

More About

*   "Subreferencing Models" on page 4-32

Concatenating Models

About Concatenating Models

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

When you concatenate identified models, such as `idtf`, `idpoly`, `idproc`, and `idss` model objects, the resulting model combines the parameters of the individual models. However, the estimated parameter covariance is lost. If you want to translate the covariance information during concatenation, use `translatecov`.

Concatenation is not supported for `idgrey` models; convert them to `idss` models first if you want to perform concatenation.

You can also concatenate nonparametric models, which contain the estimated impulse-response (`idtf` object) and frequency-response (`idfrd` object) of a system.

In case of `idfrd` models, concatenation combines information in the `ResponseData` properties of the individual model objects. `ResponseData` is an ny-by-nu-by-nf array that stores the response of the system, where `ny` is the number of output channels, `nu` is the number of input channels, and `nf` is the number of frequency values. The `(j,i,:)` vector of the resulting response data represents the frequency response from the `i`th input to the `j`th output at all frequencies.

Limitation on Supported Models

Concatenation is supported for linear models only.

Horizontal Concatenation of Model Objects

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

The following syntax creates a new model object `m` that contains the horizontal concatenation of `m1,m2,...,mN`:

`m = [m1,m2,...,mN]`

`m` takes all of the inputs of `m1,m2,...,mN` to the same outputs as in the original models. The following diagram is a graphical representation of horizontal concatenation of the models.

Vertical Concatenation of Model Objects

Vertical concatenation combines output channels of specified models. Vertical concatenation of model objects requires that they have the same inputs. If the input channel names are different and their dimensions are the same, the concatenation operation resets the input channel names to their default (`''`) values.

The following syntax creates a new model object `m` that contains the vertical concatenation of `m1,m2,...,mN`:

```
m = [m1;m2;... ;mN]
```

`m` takes the same inputs in the original models to all of the output of `m1,m2,...,mN`. The following diagram is a graphical representation of vertical concatenation of frequency-response data.

Concatenating Noise Spectrum Data of idfrd Objects

When `idfrd` models are obtained as a result of estimation (such as using `spa`), the `SpectrumData` property is not empty and contains the power spectra and cross spectra of the output noise in the system. For each output channel, this toolbox estimates one noise channel to explain the difference between the output of the model and the measured output.

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

In case of horizontal concatenation, there is no meaningful way to combine the `SpectrumData` of individual `idfrd` objects, and the resulting `SpectrumData` property is empty. An empty property results because each `idfrd` object has its own set of noise channels, where the number of noise channels equals the number of outputs. When the resulting `idfrd` object contains the same output channels as each of the individual `idfrd` objects, it cannot accommodate the noise data from all the `idfrd` objects.

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

$$
m.s = \begin{pmatrix} m1.s & & 0 \\ & \ddots & \\ 0 & & mN.s \end{pmatrix}
$$

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

See Also

If you have the Control System Toolbox product, see "Combining Model Objects" on page 19-4 about additional functionality for combining models.

Merging Models

You can merge models of the same structure to obtain a single model with parameters that are statistically weighed means of the parameters of the individual models. When computing the merged model, the covariance matrices of the individual models determine the weights of the parameters.

You can perform the merge operation for the `idtf`, `idgrey`, `idpoly`, `idproc`, and `idss` model objects.

Note Each merge operation merges the same type of model object.

Merging models is an alternative to merging data sets into a single multiexperiment data set, and then estimating a model for the merged data. Whereas merging data sets assumes that the signal-to-noise ratios are about the same in the two experiments, merging models allows greater variations in model uncertainty, which might result from greater disturbances in an experiment.

When the experimental conditions are about the same, merge the data instead of models. This approach is more efficient and typically involves better-conditioned calculations. For more information about merging data sets into a multiexperiment data set, see "Create Multiexperiment Data at the Command Line" on page 2-42.

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

Determining Model Order and Delay

Estimation requires you to specify the model order and delay. Many times, these values are not known. You can determine the model order and delay in one of the following ways:

- Guess their values by visually inspecting the data or based on the prior knowledge of the system.
- Estimate delay as a part of `idproc` or `idtf` model estimation. These models treat delay as an estimable parameter and you can determine their values by the estimation commands `procest` and `tfest`, respectively. However automatic estimation of delays can cause errors. Therefore, it is recommended that you analyze the data for delays in advance.
- To estimate delays, you can also use one of the following tools:

  - Estimate delay using `delayest`. The choice of the order of the underlying ARX model and the lower/upper bound on the value of the delay to be estimated influence the value returned by `delayest`.
  - Compute impulse response using `impulseest`. Plot the impulse response with a confidence interval of sufficient standard deviations (usually 3). The delay is indicated by the number of response samples that are inside the statistically zero region (marked by the confidence bound) before the response goes outside that region.
  - Select the model order in `n4sid` by specifying the model order as a vector.
  - Choose the model order of an ARX model using `arxstruc` or `ivstruc` and `selstruc`. These command select the number of poles, zeros and delay.

  See "Model Structure Selection: Determining Model Order and Input Delay" on page 4-40 for an example of using these tools.

Model Structure Selection: Determining Model Order and Input Delay

This example shows some methods for choosing and configuring the model structure. Estimation of a model using measurement data requires selection of a model structure (such as state-space or transfer function) and its order (e.g., number of poles and zeros) in advance. This choice is influenced by prior knowledge about the system being modeled, but can also be motivated by an analysis of data itself. This example describes some options for determining model orders and input delay.

Introduction

Choosing a model structure is usually the first step towards its estimation. There are various possibilities for structure - state-space, transfer functions and polynomial forms such as ARX, ARMAX, OE, BJ etc. If you do not have detailed prior knowledge of your system, such as its noise characteristics and indication of feedback, the choice of a reasonable structure may not be obvious. Also for a given choice of structure, the order of the model needs to be specified before the corresponding parameters are estimated. System Identification Toolbox™ offers some tools to assist in the task of model order selection.

The choice of a model order is also influenced by the amount of delay. A good idea of the input delay simplifies the task of figuring out the orders of other model coefficients. Discussed below are some options for input delay determination and model structure and order selection.

Choosing and Preparing Example Data for Analysis

This example uses the hair dryer data, also used by iddemo1 ("Estimating Simple Models from Real Laboratory Process Data"). The process consists of air being fanned through a tube. The air is heated at the inlet of the tube, and the input is the voltage applied to the heater. The output is the temperature at the outlet of the tube.

Let us begin by loading the measurement data and doing some basic preprocessing:

```
load dry2
```

Form a data set for estimation of the first half, and a reference set for validation purposes of the second half:

```
ze = dry2(1:500);
zr = dry2(501:1000);
```

Detrend each of the sets:

```
ze = detrend(ze);
zr = detrend(zr);
```

Let us look at a portion of the estimation data:

```
plot(ze(200:350))
```
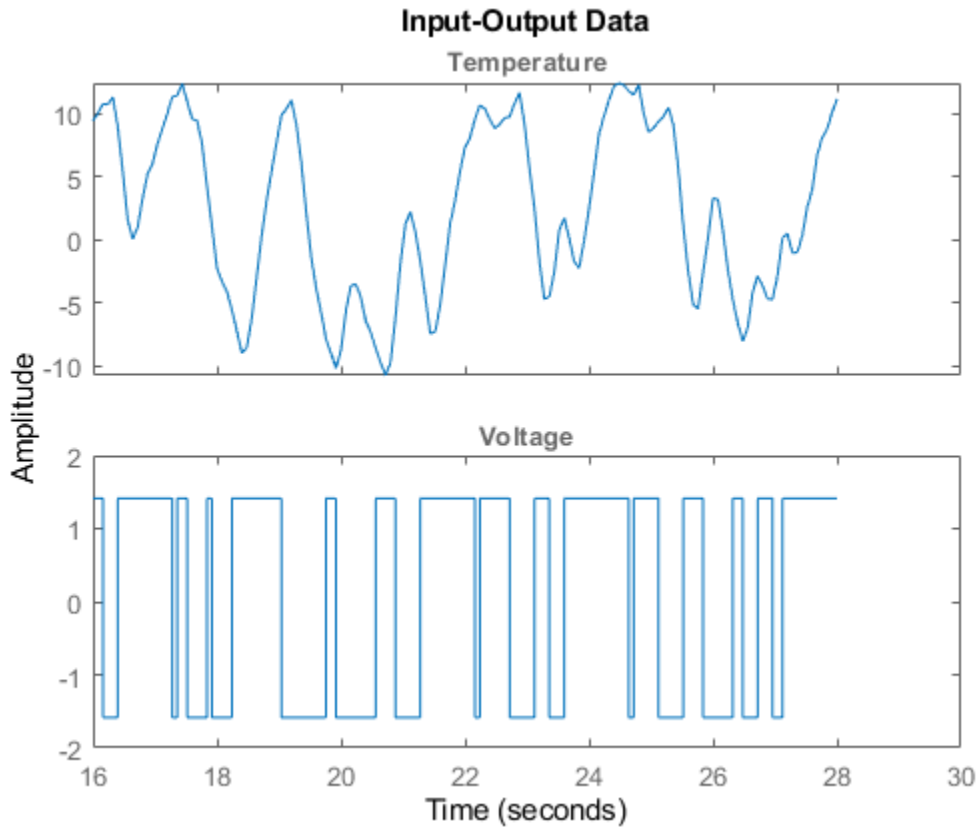
Estimating Input Delay

There are various options available for determining the time delay from input to output. These are:

- Using the DELAYEST utility.
- Using a non-parametric estimate of the impulse response, using IMPULSEEST.
- Using the state-space model estimator N4SID with a number of different orders and finding the delay of the 'best' one.

Using delayest:

Let us discuss the above options in detail. Function delayest returns an estimate of the delay for a given choice of orders of numerator and denominator polynomials. This function evaluates an ARX structure:

y(t) + a1*y(t-1) + ... + ana*y(t-na) = b1*u(t-nk) + ...+bnb*u(t-nb-nk+1)

with various delays and chooses the delay value that seems to return the best fit. In this process, chosen values of na and nb are used.

```
delay = delayest(ze) % na = nb = 2 is used, by default

delay =

    3
```

A value of 3 is returned by default. But this value may change a bit if the assumed orders of numerator and denominator polynomials (2 here) is changed. For example:

```
delay = delayest(ze,5,4)
```

```
delay =

    2
```

returns a value of 2. To gain insight into how `delayest` works, let us evaluate the loss function for various choices of delays explicitly. We select a second order model (`na=nb=2`), which is the default for `delayest`, and try out every time delay between 1 and 10. The loss function for the different models are computed using the validation data set:

```
V = arxstruc(ze,zr,struc(2,2,1:10));
```

We now select that delay that gives the best fit for the validation data:

```
[nn,Vm] = selstruc(V,0); % nn is given as [na nb nk]
```

The chosen structure was:

```
nn
```

```
nn =

    2    2    3
```

which show the best model has a delay of nn(3) = 3.

We can also check how the fit depends on the delay. This information is returned in the second output `Vm`. The logarithms of a quadratic loss function are given as the first row, while the indexes `na`, `nb` and `nk` are given as a column below the corresponding loss function.

```
Vm
```

```
Vm =

  Columns 1 through 7

   -0.1480   -1.3275   -1.8747   -0.2403   -0.0056    0.0736    0.1763
    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000
    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000
    1.0000    2.0000    3.0000    4.0000    5.0000    6.0000    7.0000

  Columns 8 through 10

    0.1906    0.1573    0.1474
    2.0000    2.0000    2.0000
    2.0000    2.0000    2.0000
    8.0000    9.0000   10.0000
```

The choice of 3 delays is thus rather clear, since the corresponding loss is minimum.

Using `impulse`

To gain a better insight into the dynamics, let us compute the impulse response of the system. We will use the function `impulseest` to compute a non-parametric impulse response model. We plot this response with a confidence interval represented by 3 standard deviations.

```
FIRModel = impulseest(ze);
clf
h = impulseplot(FIRModel);
showConfidence(h,3)
```

Impulse Response

From: Voltage To: Temperature
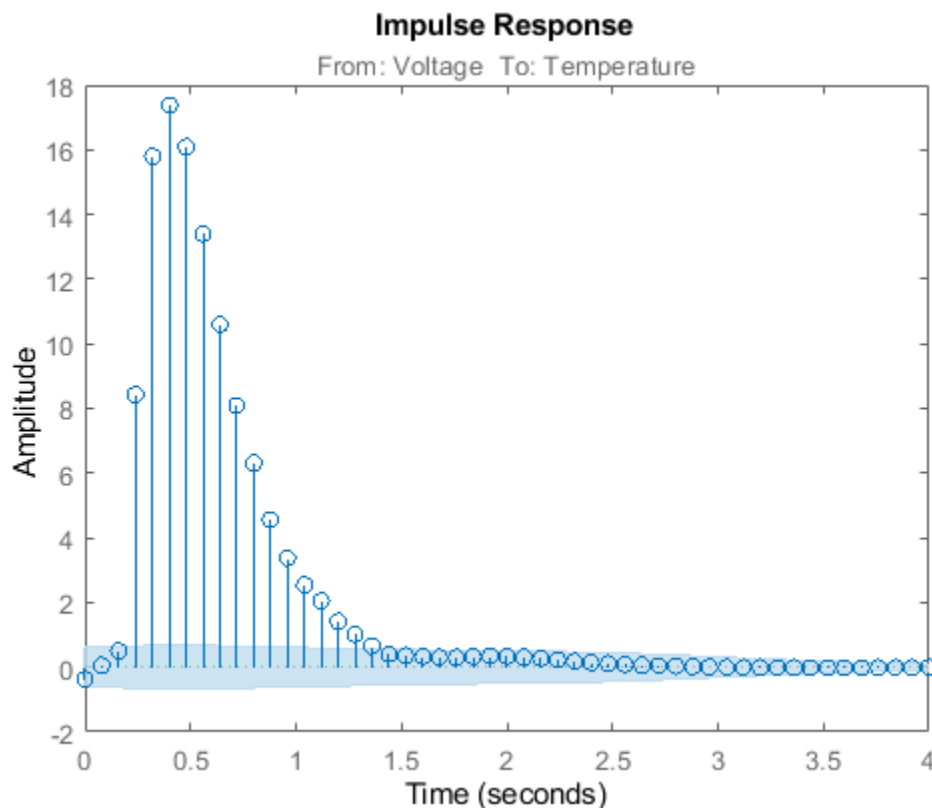
The filled light-blue region shows the confidence interval for the insignificant response in this estimation. There is a clear indication that the impulse response "takes off" (leaves the uncertainty region) after 3 samples. This points to a delay of three intervals.

Using `n4sid` based state-space evaluation

We may also estimate a family of parametric models to find the delay corresponding to the "best" model. In case of state-space models, a range of orders may be evaluated simultaneously and the best order picked from a Hankel Singular Value plot. Execute the following command to invoke `n4sid` in an interactive mode:

```
m = n4sid(ze,1:15); % All orders between 1 and 15.
```
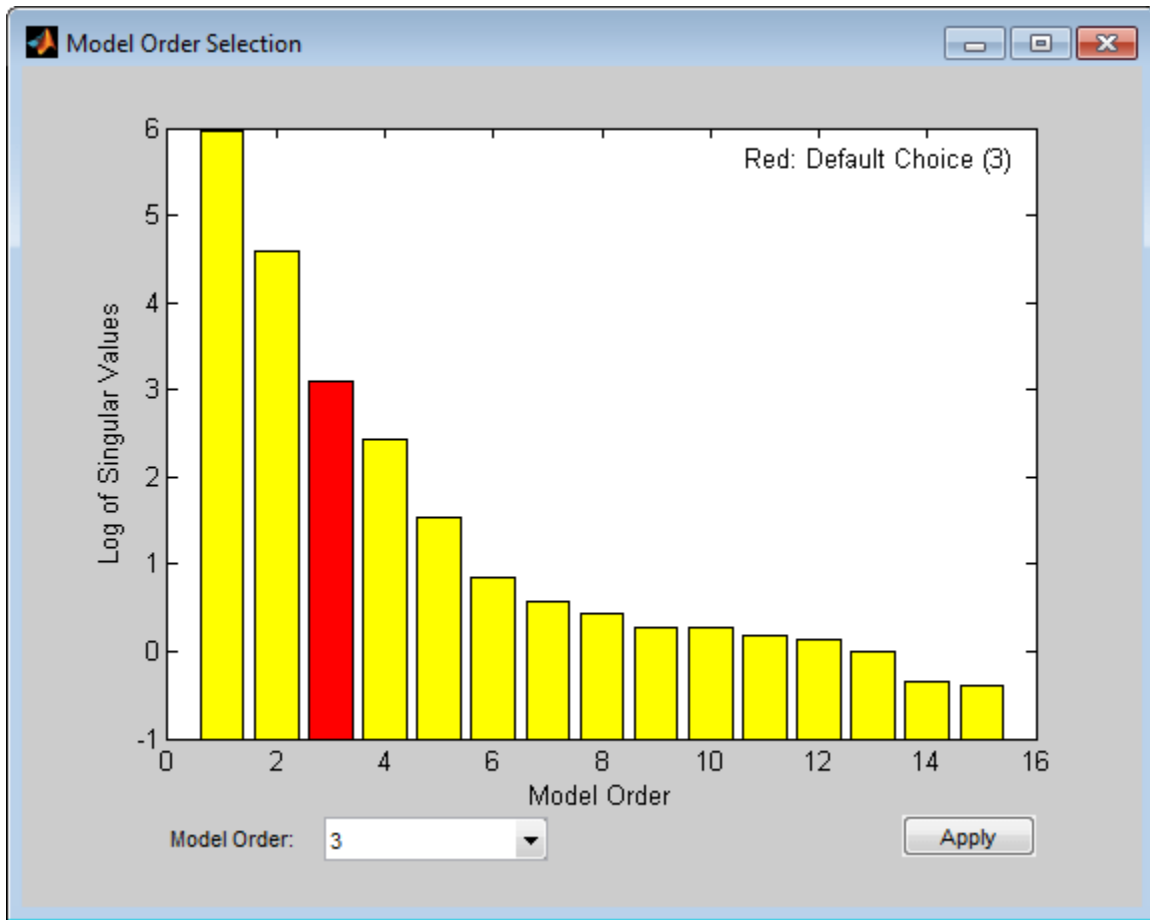
The plot indicates an order of 3 as the best value. For this choice, let us compute the impulse response of the model m:

```
m = n4sid(ze, 3);
showConfidence(impulseplot(m),3)
```

Impulse Response

From: Voltage To: Temperature
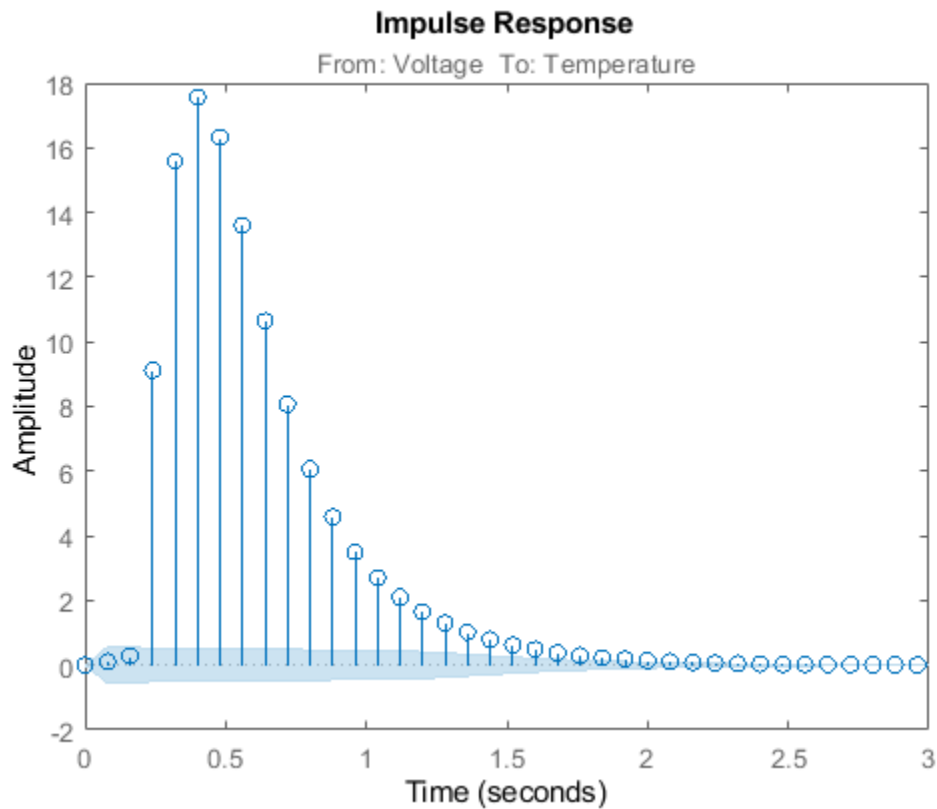
As with non-parametric impulse response, there is a clear indication that the delay from input to output is of three samples.

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
```
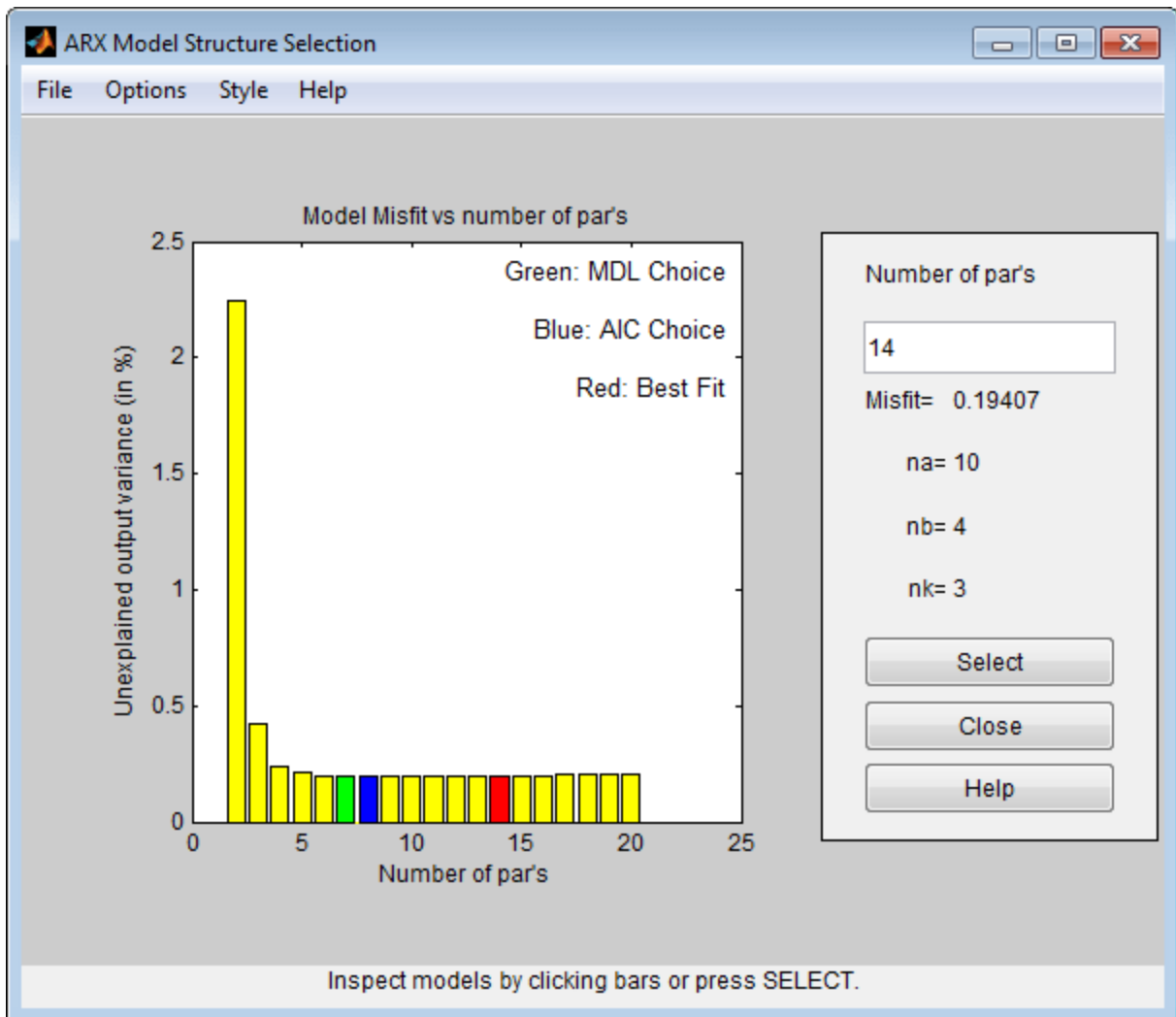
The best fit is thus obtained for nn = [4 4 3], while we see that the improved fit compared to nn = [2 2 3] is rather marginal.

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)
```
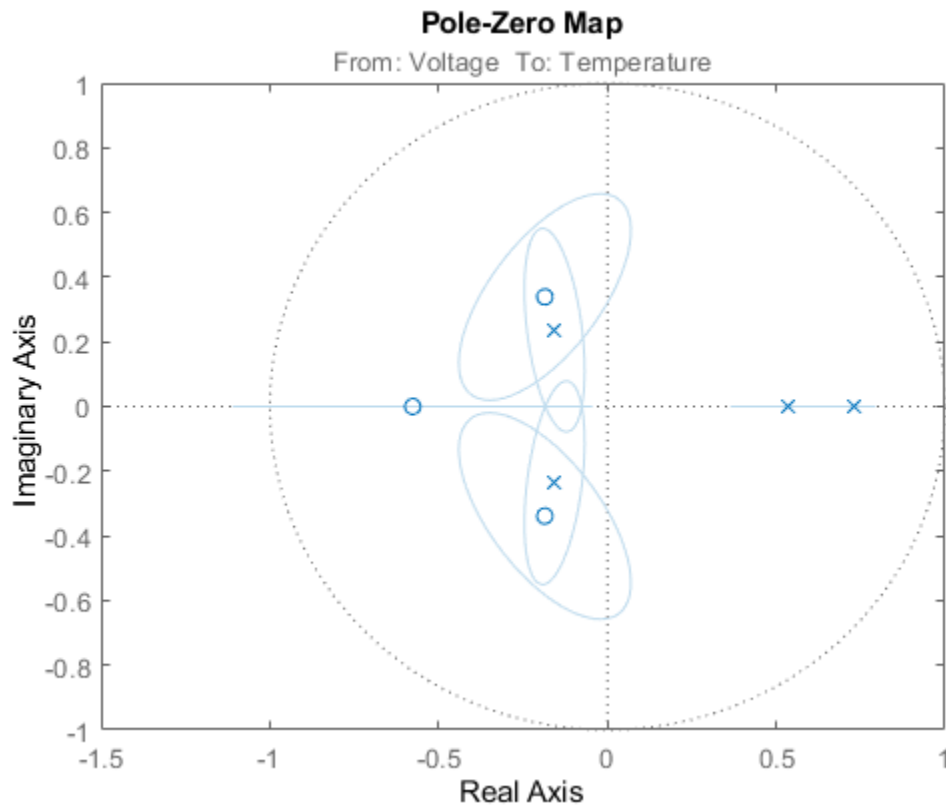

The confidence intervals for the two complex-conjugate poles and zeros overlap, indicating they are likely to cancel each other. Hence, a second order model might be adequate. Based on this evidence, let us compute a 2nd order ARX model:

```
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)
```

Simulated Response Comparison
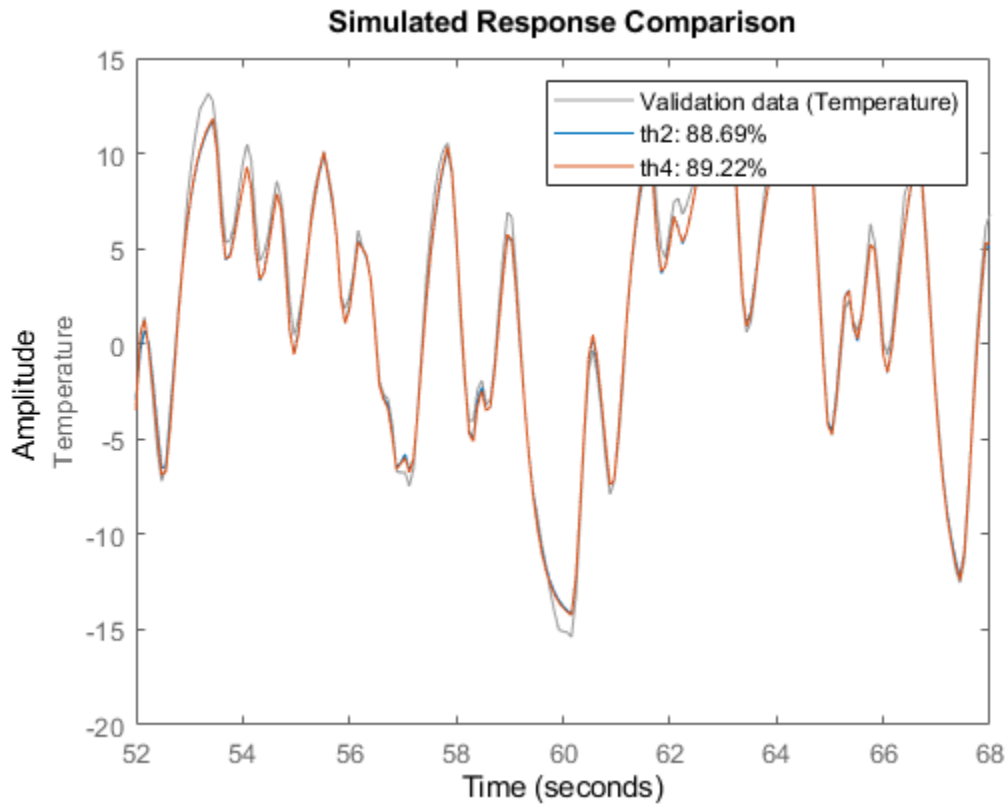
The plot indicates that there was no significant loss of accuracy in reducing the order from 4 to 2. We can also check the residuals ("leftovers") of this model, i.e., what is left unexplained by the model.

```
e = resid(ze,th2);
plot(e(:,1,[])), title('The residuals')
```
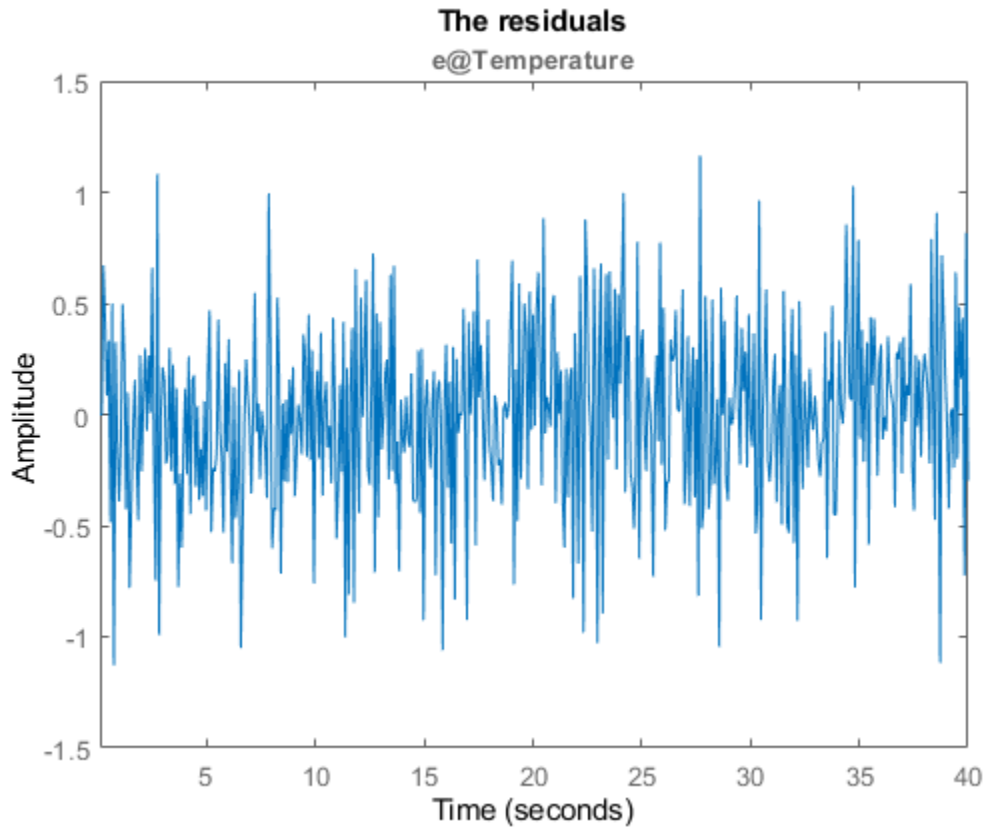
We see that the residuals are quite small compared to the signal level of the output, that they are reasonably well (although not perfectly) uncorrelated with the input and among themselves. We can thus be (provisionally) satisfied with the model `th2`.

Let us now check if we can determine the model order for a state-space structure. As before, we know the delay is 3 samples. We can try all orders from 1 to 15 with a total lag of 3 samples in `n4sid`. Execute the following command to try various orders and choose one interactively.

```
ms = n4sid(ze,[1:15],'InputDelay',2); %n4sid estimation with variable orders
```
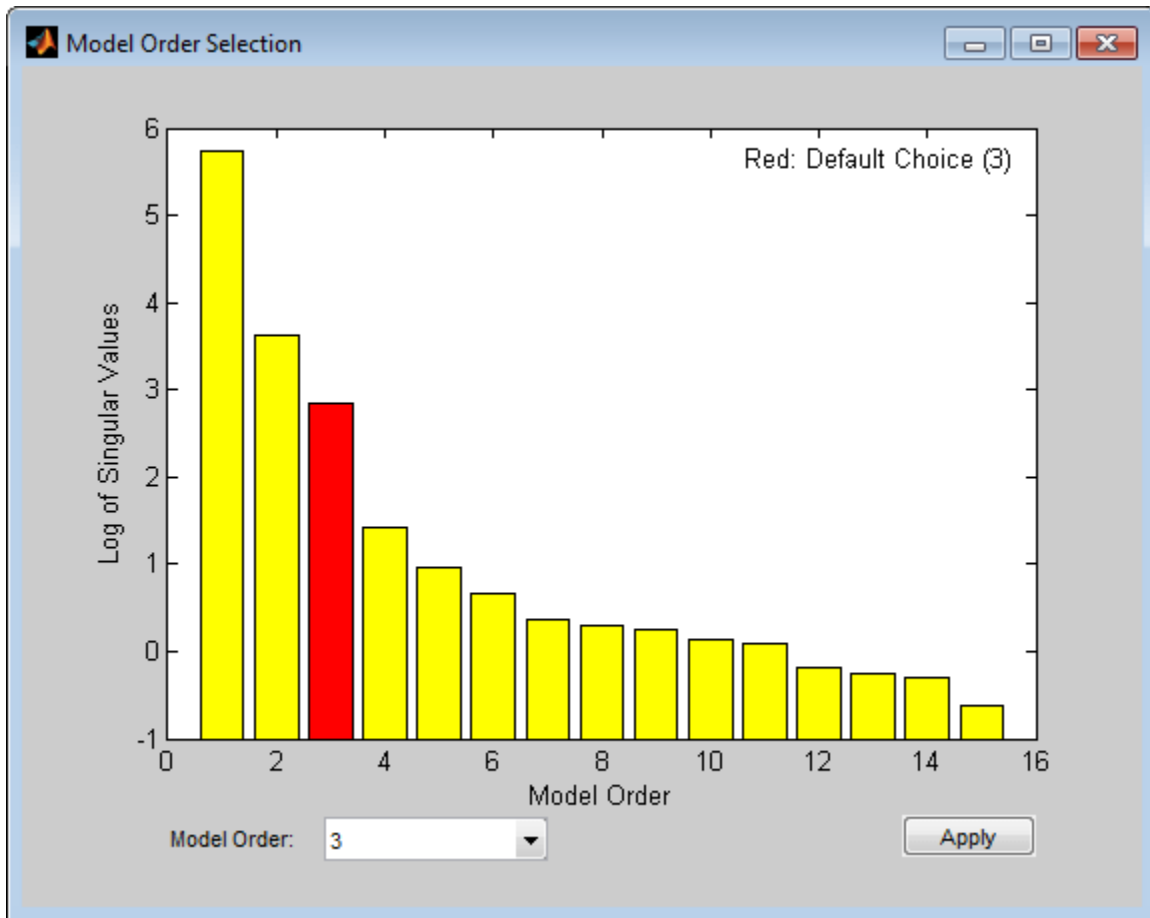
The "InputDelay" was set to 2 because by default `n4sid` estimates a model with no feedthrough (which accounts for one sample lag between input and output). The default order, indicated in the figure above, is 3, that is in good agreement with our earlier findings. Finally, we compare how the state-space model `ms` and the ARX model `th2` compare in reproducing the measured output of the validation data:

```
ms = n4sid(ze,3,'InputDelay',2);
compare(zr,ms,th2)
```
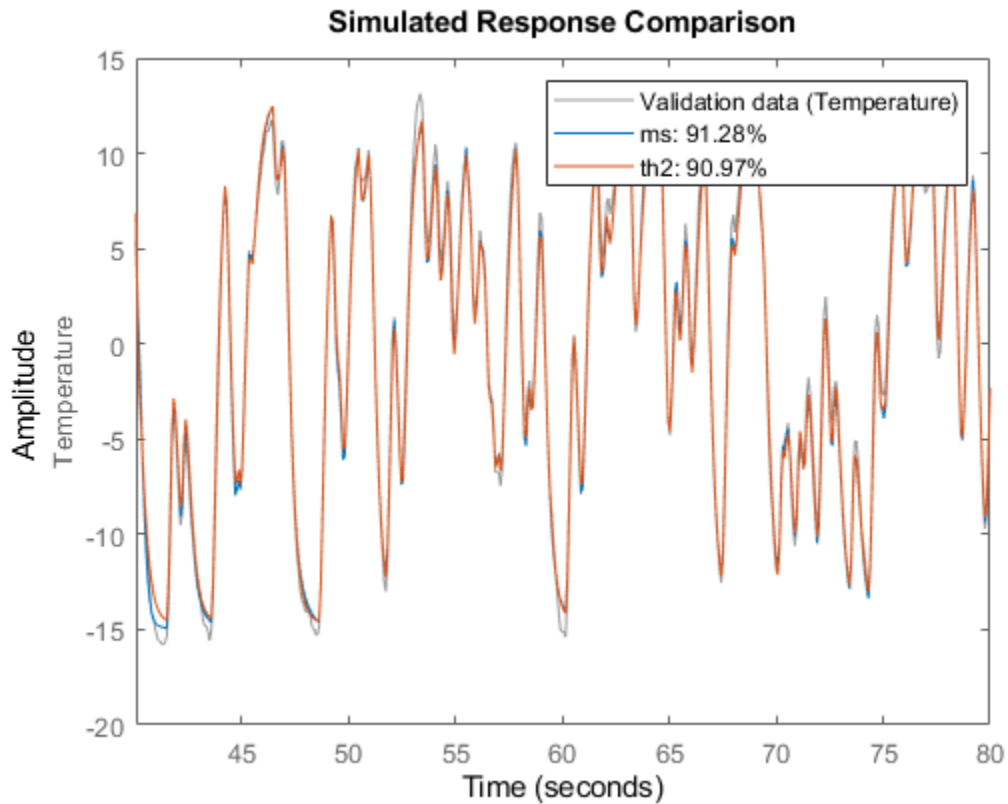
The comparison plot indicates that the two models are practically identical.

Conclusions

This example described some options for choosing a reasonable model order. Determining delay in advance can simplify the task of choosing orders. With ARX and state-space structures, we have some special tools (`arx` and `n4sid` estimators) for automatically evaluating a whole set of model orders, and choosing the best one among them. The information revealed by this exercise (using utilities such as `arxstruc`, `selstruc`, `n4sid` and `delayest`) could be used as a starting point when estimating models of other structures, such as BJ and ARMAX.

Frequency Domain Identification: Estimating Models Using Frequency Domain Data

This example shows how to estimate models using frequency domain data. The estimation and validation of models using frequency domain data work the same way as they do with time domain data. This provides a great amount of flexibility in estimation and analysis of models using time and frequency domain as well as spectral (FRF) data. You may simultaneously estimate models using data in both domains, compare and combine these models. A model estimated using time domain data may be validated using spectral data or vice-versa.

Frequency domain data cannot be used for estimation or validation of nonlinear models.

Introduction

Frequency domain experimental data are common in many applications. It could be that the data was collected as frequency response data (frequency functions: FRF) from the process using a frequency analyzer. It could also be that it is more practical to work with the input's and output's Fourier transforms (FFT of time-domain data), for example to handle periodic or band-limited data. (A band-limited continuous time signal has no frequency components above the Nyquist frequency). In System Identification Toolbox, frequency domain I/O data are represented the same way as time-domain data, i.e., using `iddata` objects. The 'Domain' property of the object must be set to 'Frequency'. Frequency response data are represented as complex vectors or as magnitude/phase vectors as a function of frequency. IDFRD objects in the toolbox are used to encapsulate FRFs, where a user specifies the complex response data and a frequency vector. Such IDDATA or IDFRD objects (and also FRD objects of Control System Toolbox) may be used seamlessly with any estimation routine (such as `procest`, `tfest` etc).

Inspecting Frequency Domain Data

Let us begin by loading some frequency domain data:

```
load demofr
```

This MAT-file contains frequency response data at frequencies W, with the amplitude response AMP and the phase response PHA. Let us first have a look at the data:

```
subplot(211), loglog(W,AMP),title('Amplitude Response')
subplot(212), semilogx(W,PHA),title('Phase Response')
```
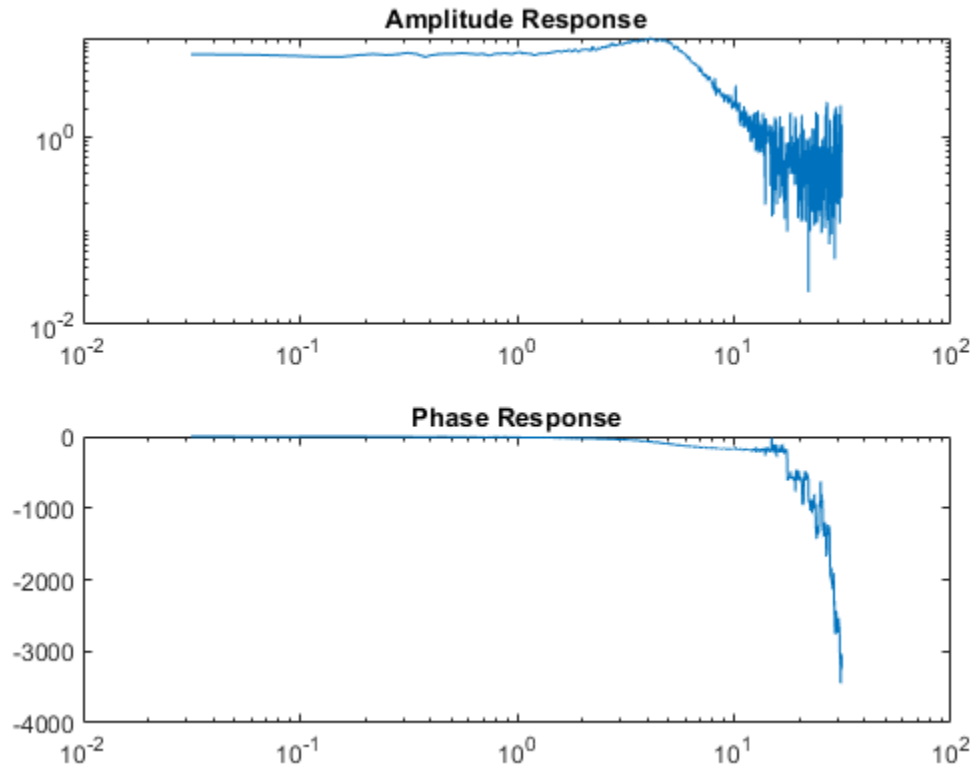
This experimental data will now be stored as an IDFRD object. First transform amplitude and phase to a complex valued response:

```
zfr = AMP.*exp(1i*PHA*pi/180);
Ts = 0.1;
gfr = idfrd(zfr,W,Ts);
```

Ts is the sample time of the underlying data. If the data corresponds to continuous time, for example since the input has been band-limited, use Ts = 0.

Note: If you have the Control System Toolbox™, you could use an FRD object instead of the IDFRD object. IDFRD has options for more information, like disturbance spectra and uncertainty measures which are not available in FRD objects.

The IDFRD object gfr now contains the data, and it can be plotted and analyzed in different ways. To view the data, we may use plot or bode:

```
clf
bode(gfr), legend('gfr')
```
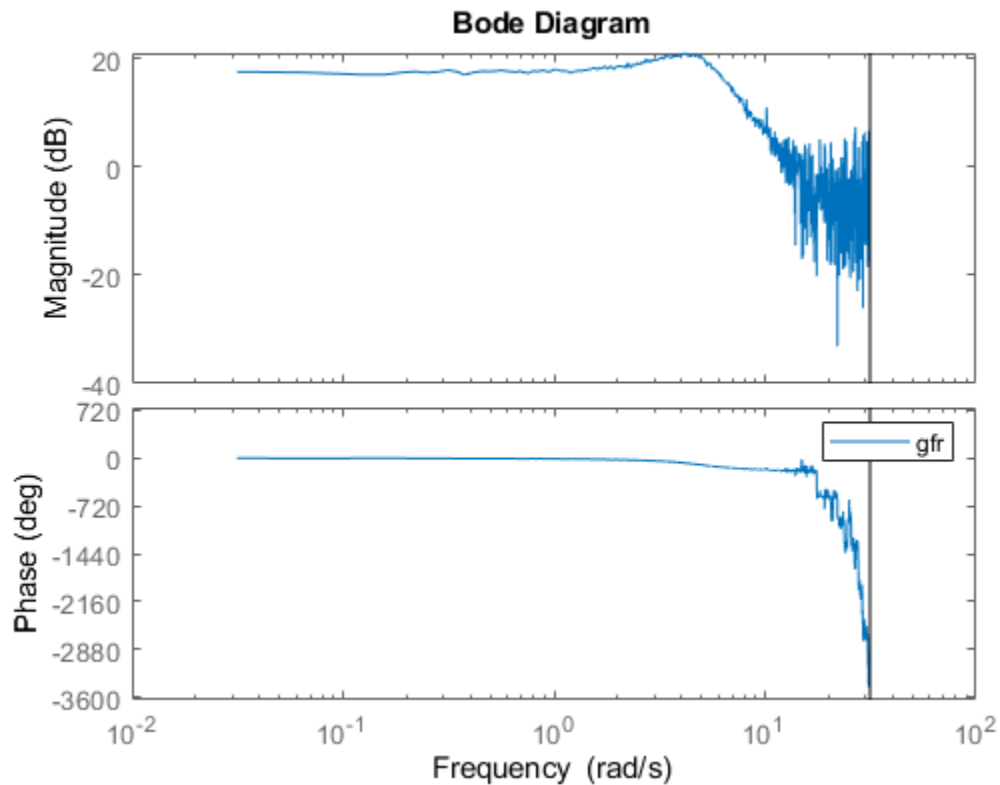
Estimating Models Using Frequency Response (FRF) Data

To estimate models, you can now use `gfr` as a data set with all the commands of the toolbox in a transparent fashion. The only restriction is that noise models cannot be built. This means that for polynomial models only OE (output-error models) apply, and for state-space models, you have to fix K = 0.

```
m1 = oe(gfr,[2 2 1]) % Discrete-time Output error (transfer function) model
ms = ssest(gfr) % Continuous-time state-space model with default choice of order
mproc = procest(gfr,'P2UDZ') % 2nd-order, continuous-time model with underdamped poles
compare(gfr,m1,ms,mproc)
L = findobj(gcf,'type','legend');
L.Location = 'southwest'; % move legend to non-overlapping location
```

```
m1 =
Discrete-time OE model: y(t) = [B(z)/F(z)]u(t) + e(t)
  B(z) = 0.9986 z^-1 + 0.4968 z^-2

  F(z) = 1 - 1.499 z^-1 + 0.6998 z^-2

Sample time: 0.1 seconds

Parameterization:
   Polynomial orders:   nb=2   nf=2   nk=1
   Number of free coefficients: 4
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.
```

```
Status:
Estimated using OE on frequency response data "gfr".
Fit to estimation data: 88.04%
FPE: 0.2512, MSE: 0.2492


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   -1.785    3.097
   x2   -6.835   -1.785

  B =
            u1
   x1    -4.15
   x2    27.17

  C =
            x1       x2
   y1     1.97   0.3947

  D =
       u1
   y1   0

  K =
       y1
   x1   0
   x2   0

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 8
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on frequency response data "gfr".
Fit to estimation data: 88.04%
FPE: 0.2512, MSE: 0.2492


mproc =

Process model with transfer function:
                      1+Tz*s
  G(s) = Kp * --------------------- * exp(-Td*s)
              1+2*Zeta*Tw*s+(Tw*s)^2

          Kp = 7.4619
          Tw = 0.20245
        Zeta = 0.36242
```

```
        Td = 0
        Tz = 0.013617

Parameterization:
    {'P2DUZ'}
  Number of free coefficients: 5
  Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on frequency response data "gfr".
Fit to estimation data: 88.03%
FPE: 0.2517, MSE: 0.2492
```
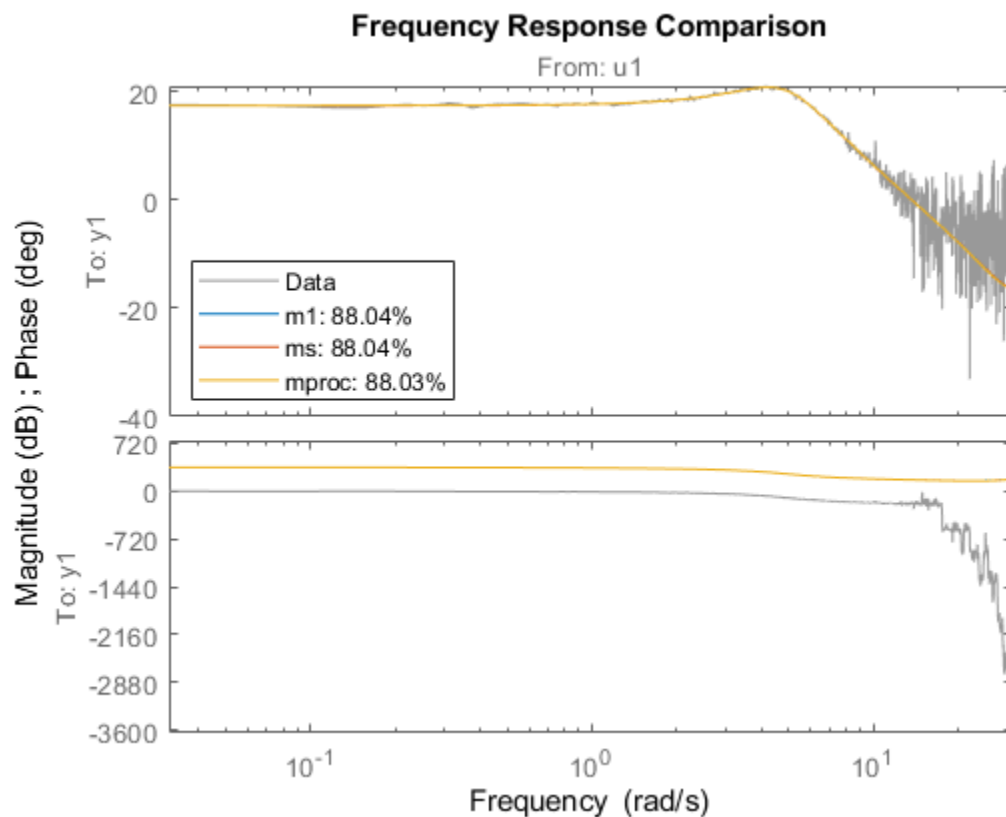

As shown above a variety of linear model types may be estimated in both continuous and discrete time domains, using spectral data. These models may be validated using, time-domain data. The time-domain I/O data set `ztime`, for example, is collected from the same system, and can be used for validation of `m1`, `ms` and `mproc`:

```
compare(ztime,m1,ms,mproc) %validation in a different domain
```
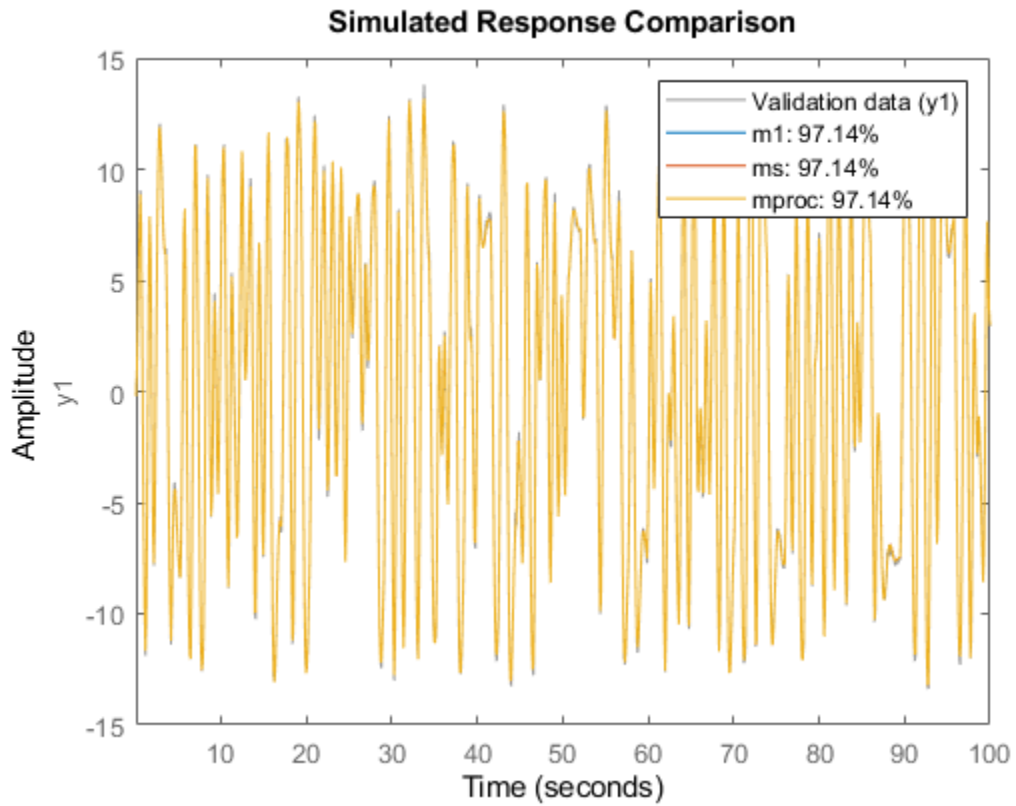
We may also look at the residuals to affirm the quality of the model using the validation data `ztime`. As observed, the residuals are almost white:

```
resid(ztime,mproc) % Residuals plot
```

Residue Correlation
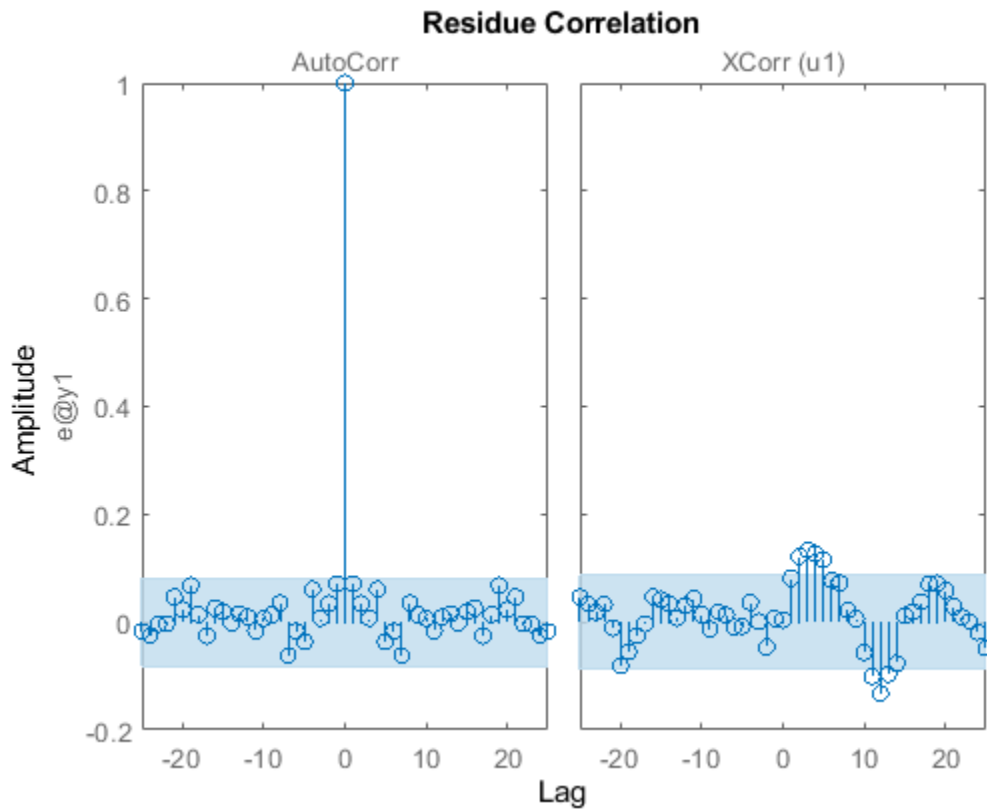
Condensing Data Using SPAFDR

An important reason to work with frequency response data is that it is easy to condense the information with little loss. The command SPAFDR allows you to compute smoothed response data over limited frequencies, for example with logarithmic spacing. Here is an example where the `gfr` data is condensed to 100 logarithmically spaced frequency values. With a similar technique, also the original time domain data can be condensed:

```
sgfr = spafdr(gfr) % spectral estimation with frequency-dependent resolution
sz = spafdr(ztime); % spectral estimation using time-domain data
clf
bode(gfr,sgfr,sz)
axis([pi/100 10*pi, -272 105])
legend('gfr (raw data)','sgfr','sz','location','southwest')


sgfr =

IDFRD model.
Contains Frequency Response Data for 1 output(s) and 1 input(s), and the spectra for disturbances
Response data and disturbance spectra are available at 100 frequency points, ranging from 0.0314

Sample time: 0.1 seconds
Output channels: 'y1'
Input channels: 'u1'
Status:
```

Estimated using SPAFDR on frequency response data "gfr".

Bode Diagram

From: u1 To: y1
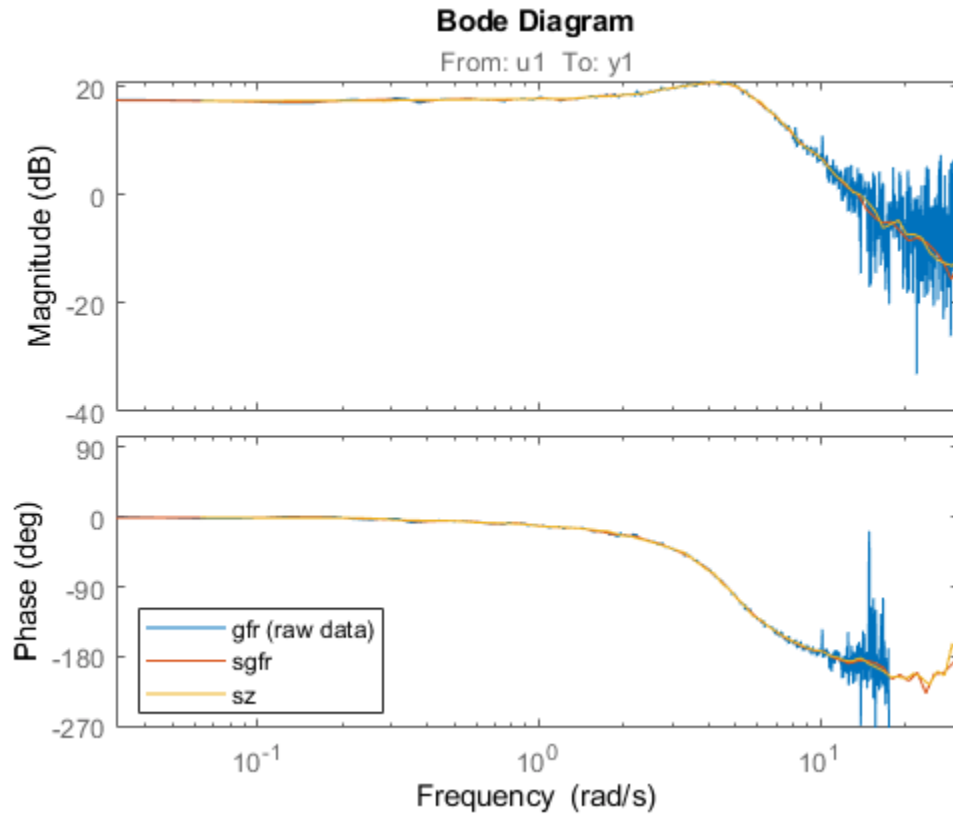
The Bode plots show that the information in the smoothed data has been taken well care of. Now, these data records with 100 points can very well be used for model estimation. For example:

```
msm = oe(sgfr,[2 2 1]);
compare(ztime,msm,m1) % msm has the same accuracy as M1 (based on 1000 points)
```

Simulated Response Comparison
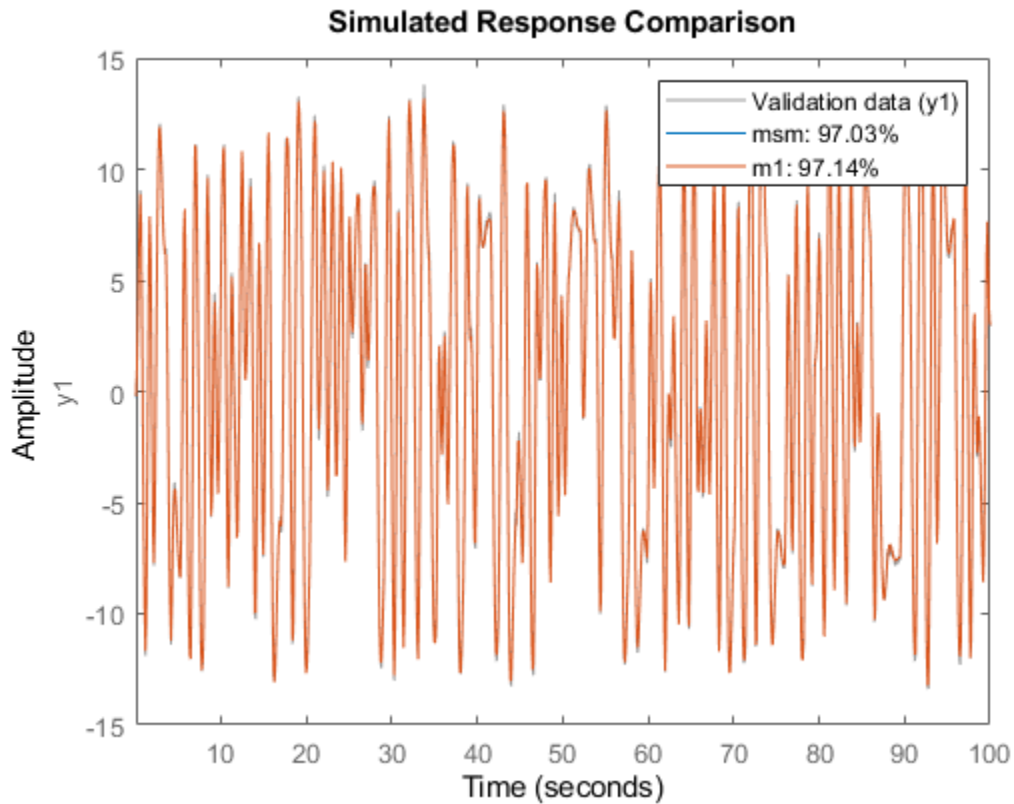
Estimation Using Frequency-Domain I/O Data

It may be that the measurements are available as Fourier transforms of inputs and output. Such frequency domain data from the system are given as the signals Y and U. In loglog plots they look like

```
Wfd = (0:500)'*10*pi/500;
subplot(211),loglog(Wfd,abs(Y)),title('The amplitude of the output')
subplot(212),loglog(Wfd,abs(U)),title('The amplitude of the input')
```
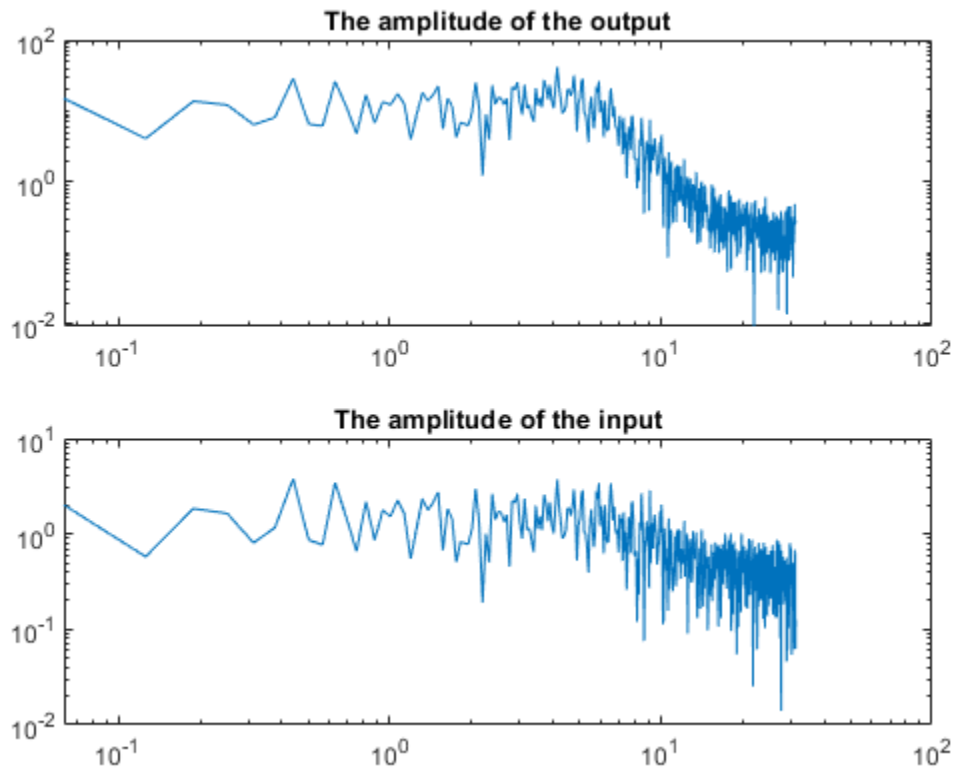
The frequency response data is essentially the ratio between Y and U. To collect the frequency domain data as an IDDATA object, do as follows:

```
ZFD = iddata(Y, U, 'Ts', 0.1, 'Frequency', Wfd)
```

```
ZFD =

Frequency domain data set with responses at 501 frequencies.
Frequency range: 0 to 31.416 rad/seconds
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

Now, again the frequency domain data set ZFD can be used as data in all estimation routines, just as time domain data and frequency response data:

```
mf = ssest(ZFD)    % SSEST picks best order in 1:10 range when called this way
mfr = ssregest(ZFD) % an alternative regularized reduction based state-space estimator
clf
compare(ztime,mf,mfr,m1)
```

```
mf =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
          x1       x2
   x1   -1.78    3.095
   x2  -6.812    -1.78

  B =
          u1
   x1   1.32
   x2  28.61

  C =
           x1         x2
   y1       2   6.416e-08

  D =
      u1
   y1   0

  K =
      y1
   x1   0
   x2   0

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 8
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on frequency domain data "ZFD".
Fit to estimation data: 97.21%
FPE: 0.04288, MSE: 0.04186


mfr =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
               x1          x2          x3          x4          x5          x6
   x1        0.701    -0.05307     -0.4345    -0.006642    -0.08085      0.1158
   x2      -0.4539    -0.09623     -0.3629      0.2113     -0.2219     -0.3536
   x3       0.1719     -0.3996      0.4385    -0.01558      0.1768     -0.2467
   x4      -0.1821      0.3465      0.3292     -0.1357     -0.2578     -0.2483
   x5     -0.09939      -0.338      0.1236     -0.2537      -0.387    -0.05591
   x6      0.06004      -0.226    -0.04117      0.6891      0.0873      0.2818
   x7      -0.1056      0.1859     0.04223      -0.629      0.1968      0.7077
   x8      0.05337      0.1948     0.06355    -0.09052      0.4216     -0.3997
   x9     -0.01696    -0.05961    -0.04891    -0.01251    -0.03521      0.3876
```

```
    x10    -0.01727    -0.1232    -0.03586    0.1187    -0.1738    0.05051

                   x7          x8          x9         x10
    x1        -0.3087    0.007547     0.02011     -0.1469
    x2       -0.01728    -0.04967      0.1144     0.03883
    x3        0.07107     -0.2977      -0.129    -0.06179
    x4        0.08461     0.03541    -0.06711     -0.1759
    x5        -0.5324      0.1778     -0.1114  -2.119e-05
    x6         -0.155      0.5047      -0.285     -0.3976
    x7        -0.2406      0.5628    -0.09159     -0.4845
    x8        -0.5674     -0.3337       0.105      0.1995
    x9        -0.2024     -0.4718    -0.01861     -0.5838
    x10       0.01243      0.2968      0.6808     -0.7726

  B =
             u1
    x1    -0.09482
    x2     0.7665
    x3     -1.036
    x4      0.162
    x5    -0.2926
    x6    -0.0805
    x7    -0.1277
    x8    -0.02441
    x9     0.0288
    x10    0.03776

  C =
            x1       x2       x3       x4       x5       x6       x7
    y1    1.956   -0.4539   -1.805   -1.356   0.3662   -1.691   0.8489

           x8       x9      x10
    y1    0.148   -0.5203   -0.4013

  D =
       u1
    y1   0

  K =
              y1
    x1       0.1063
    x2     -0.007395
    x3       0.0426
    x4      0.004966
    x5      -0.01505
    x6      0.005483
    x7     -0.0004218
    x8     -0.001044
    x9      0.003066
    x10     0.002273

Sample time: 0.1 seconds

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: estimate
   Number of free coefficients: 130
```

```
    Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSREGEST on frequency domain data "ZFD".
Fit to estimation data: 76.61% (prediction focus)
FPE: 3.448, MSE: 2.938
```
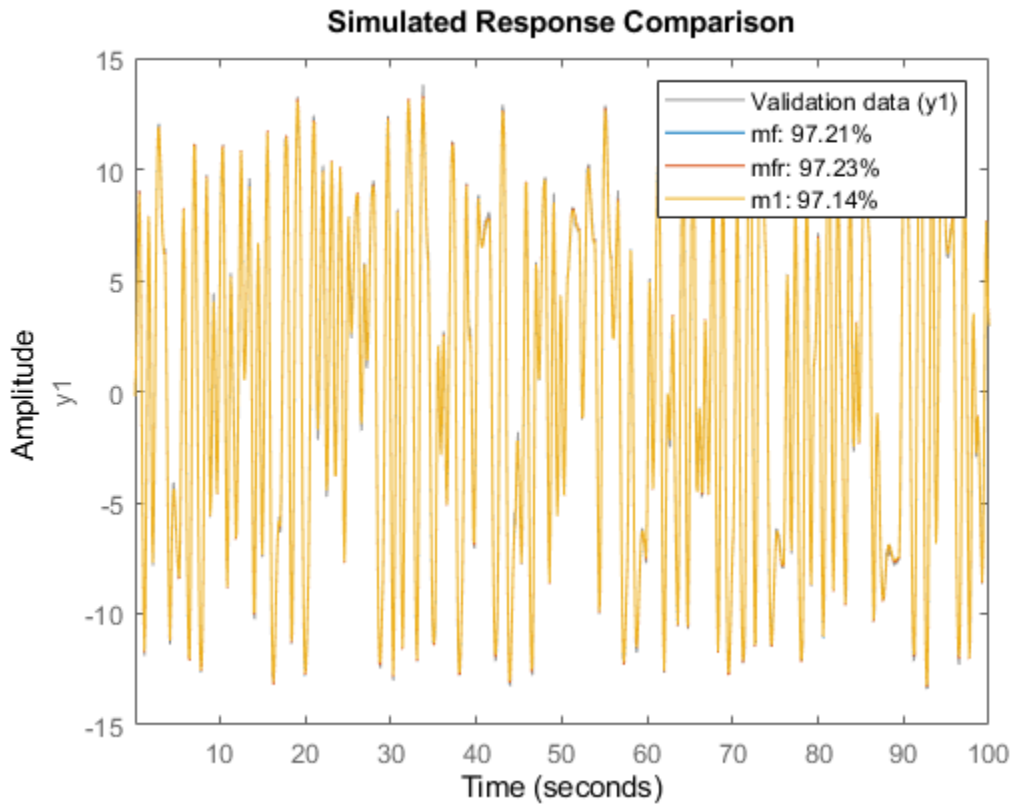

Transformations Between Data Representations (Time - Frequency)

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.
Frequency range: 0 to 31.416 rad/seconds
Sample time: 0.1 seconds

Outputs      Unit (if specified)
    y1

Inputs       Unit (if specified)
    u1
```

```
datat =

Time domain data set with 1000 samples.
Sample time: 0.1 seconds

Outputs        Unit (if specified)
    y1

Inputs         Unit (if specified)
    u1
```

Time and frequency domain input-output data can be transformed to frequency response data by SPAFDR, SPA and ETFE:

```
g1 = spafdr(ztime)
g2 = spafdr(ZFD);
clf;
bode(g1,g2)


g1 =

IDFRD model.
Contains Frequency Response Data for 1 output(s) and 1 input(s), and the spectra for disturbances
Response data and disturbance spectra are available at 100 frequency points, ranging from 0.0628

Sample time: 0.1 seconds
Output channels: 'y1'
Input channels: 'u1'
Status:
Estimated using SPAFDR on time domain data "ztime".
```
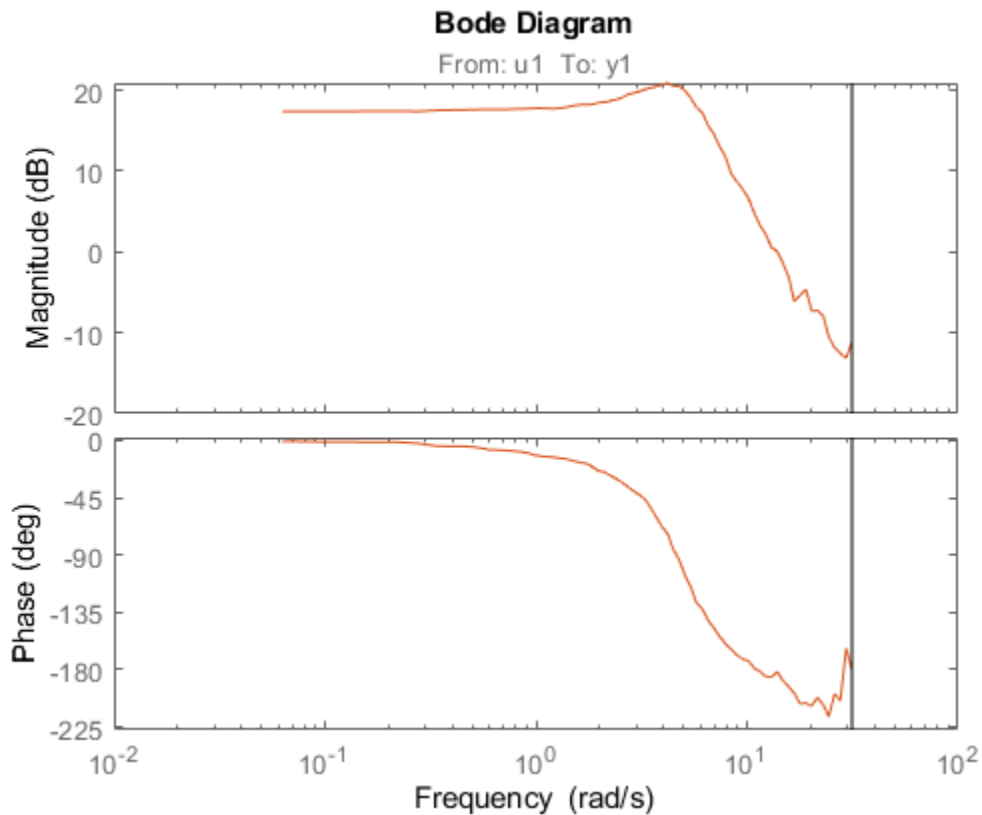
Frequency response data can also be transformed to more smoothed data (less resolution and less data) by SPAFDR and SPA;

```
g3 = spafdr(gfr);
```

Frequency response data can be transformed to frequency domain input-output signals by the command IDDATA:

```
gfd = iddata(g3)
plot(gfd)
```

```
gfd =

Frequency domain data set with responses at 100 frequencies.
Frequency range: 0.031416 to 31.416 rad/seconds
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```
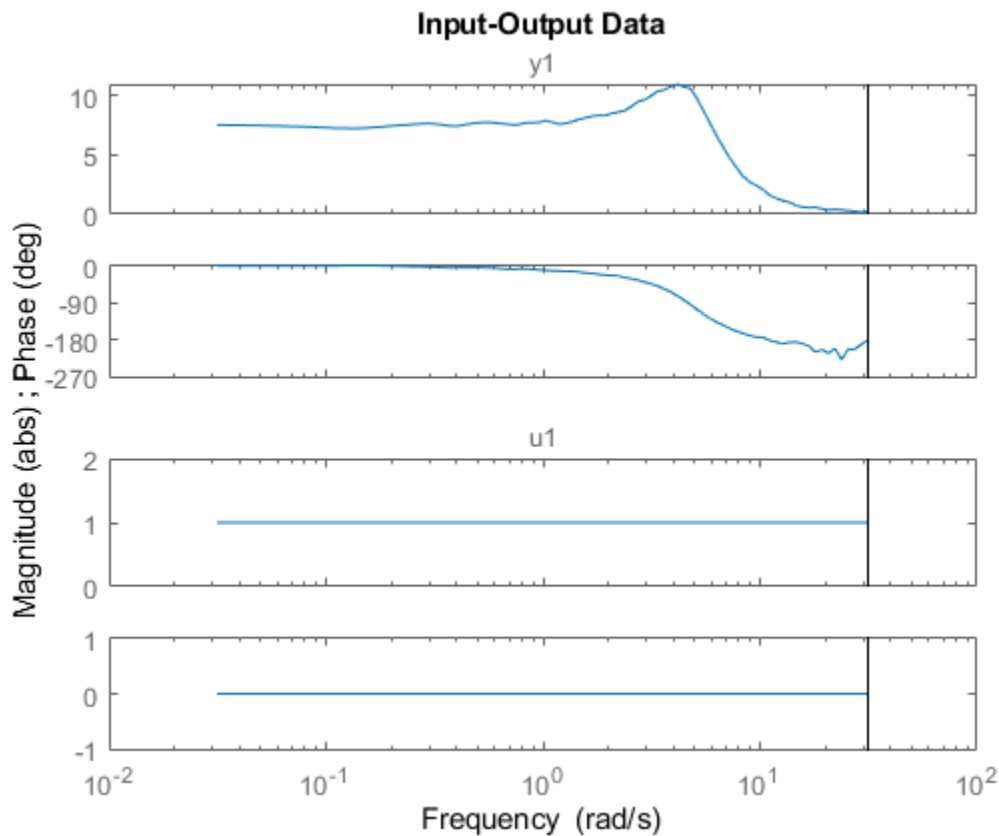
Using Continuous-time Frequency-domain Data to Estimate Continuous-time Models

Time domain data can naturally only be stored and dealt with as discrete-time, sampled data. Frequency domain data have the advantage that continuous time data can be represented correctly. Suppose that the underlying continuous time signals have no frequency information above the Nyquist frequency, e.g. because they are sampled fast, or the input has no frequency component above the Nyquist frequency and that the data has been collected from a steady-state experiment. Then the Discrete Fourier transforms (DFT) of the data also are the Fourier transforms of the continuous time signals, at the chosen frequencies. They can therefore be used to directly fit continuous time models.

This will be illustrated by the following example.

Consider the continuous time system:

$$G(s) = \frac{1}{s^2 + s + 1}$$

```
m0 = idpoly(1,1,1,1,[1 1 1],'ts',0)
```

```
m0 =
Continuous-time OE model: y(t) = [B(s)/F(s)]u(t) + e(t)
  B(s) = 1

  F(s) = s^2 + s + 1
```

```
Parameterization:
   Polynomial orders:    nb=1    nf=2    nk=0
   Number of free coefficients: 3
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

Load data that comes from steady-state simulation of this system using periodic inputs. The collected data was converted into frequency domain and saved in CTFDData.mat file.

```
load CTFDData.mat dataf % load continuous-time frequency-domain data.
```

Look at the data:

```
plot(dataf)
set(gca,'XLim',[0.1 10])
```
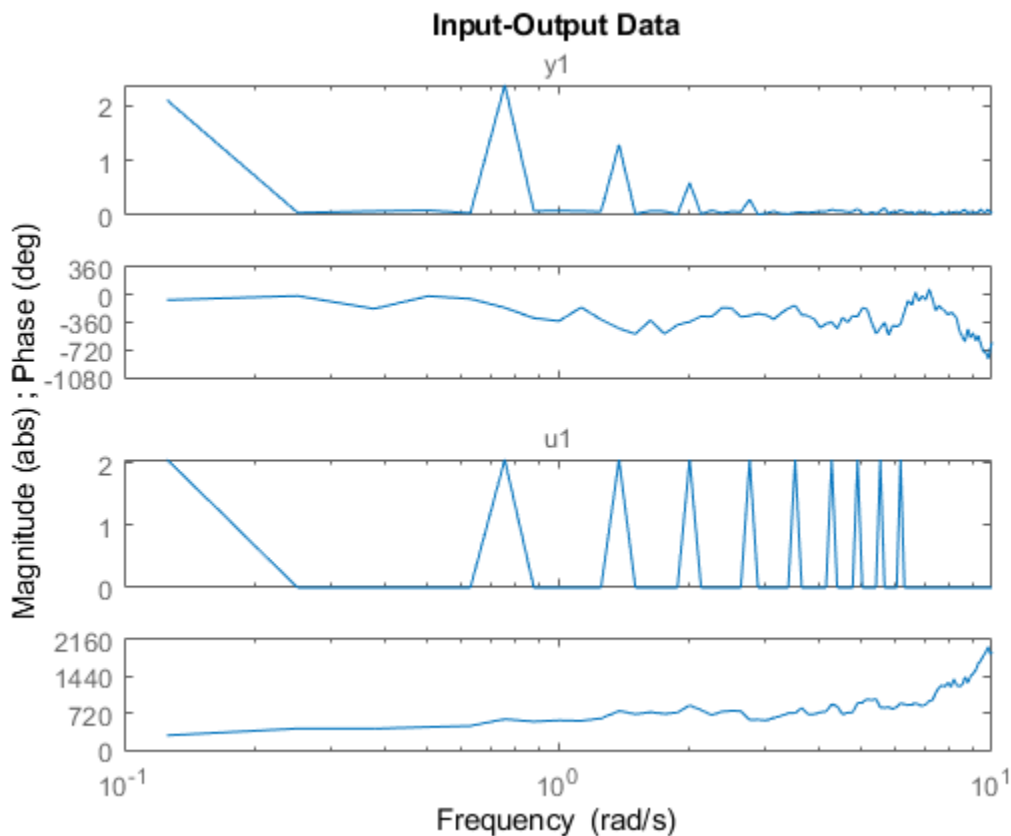

Using `dataf` for estimation will by default give continuous time models: State-space:

```
m4 = ssest(dataf,2); %Second order continuous-time model
```

For a polynomial model with `nb = 2` numerator coefficient and `nf = 2` estimated denominator coefficients use:

```
nb = 2;
nf = 2;
m5 = oe(dataf,[nb nf])
```

```
m5 =
Continuous-time OE model: y(t) = [B(s)/F(s)]u(t) + e(t)
  B(s) = -0.01831 s + 1.009

  F(s) = s^2 + 1.001 s + 0.997

Parameterization:
   Polynomial orders:    nb=2    nf=2    nk=0
   Number of free coefficients: 4
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using OE on frequency domain data "dataf".
Fit to estimation data: 70.15%
FPE: 0.00491, MSE: 0.00468
```

Compare step responses with uncertainty of the true system m0 and the models m4 and m5. The confidence intervals are shown with patches in the plot.

```
clf
h = stepplot(m0,m4,m5);
showConfidence(h,1)
legend('show','location','southeast')
```
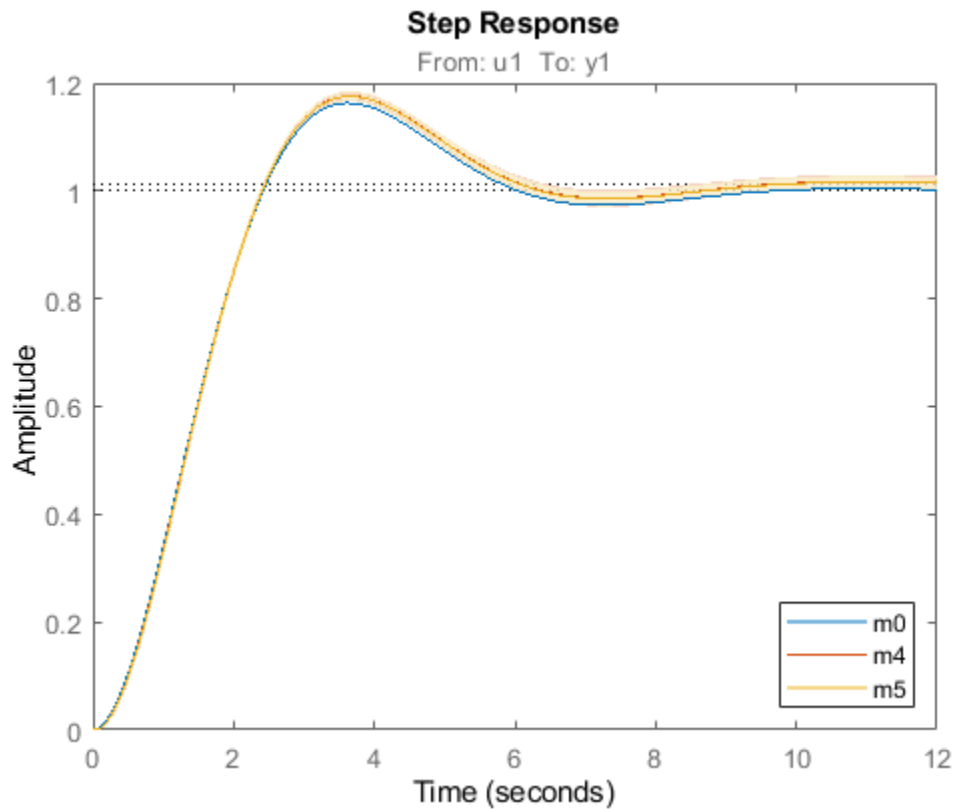
Although it was not necessary in this case, it is generally advised to focus the fit to a limited frequency band (low pass filter the data) when estimating using continuous time data. The system has a bandwidth of about 3 rad/s, and was excited by sinusoids up to 6.2 rad/s. A reasonable frequency range to focus the fit to is then [0 7] rad/s:

```
m6 = ssest(dataf,2,ssestOptions('WeightingFilter',[0 7])) % state space model
```

```
m6 =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
            x1         x2
    x1   -0.5011     1.001
    x2   -0.7446   -0.5011

  B =
             u1
    x1   -0.01706
    x2     1.016

  C =
             x1           x2
    y1       1.001   -0.0005347
```

```
   D =
         u1
    y1    0

   K =
         y1
    x1    0
    x2    0

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 8
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on frequency domain data "dataf".
Fit to estimation data: 87.03% (data prefiltered)
FPE: 0.004832, MSE: 0.003631
```

```
m7 = oe(dataf,[1 2],oeOptions('WeightingFilter',[0 7])) % polynomial model of Output Error struct
```

```
m7 =
Continuous-time OE model: y(t) = [B(s)/F(s)]u(t) + e(t)
  B(s) = 0.9861

  F(s) = s^2 + 0.9498 s + 0.9704

Parameterization:
   Polynomial orders:   nb=1   nf=2   nk=0
   Number of free coefficients: 3
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using OE on frequency domain data "dataf".
Fit to estimation data: 86.81% (data prefiltered)
FPE: 0.004902, MSE: 0.003752
```

```
opt = procestOptions('SearchMethod','lsqnonlin',...
    'WeightingFilter',[0 7]); % Requires Optimization Toolbox(TM)
m8 = procest(dataf,'P2UZ',opt)  % process model with underdamped poles
```

```
m8 =

Process model with transfer function:
                    1+Tz*s
  G(s) = Kp * ----------------------
              1+2*Zeta*Tw*s+(Tw*s)^2

         Kp = 1.0124
         Tw = 1.0019
       Zeta = 0.5021
         Tz = -0.017474
```

```
Parameterization:
    {'P2UZ'}
   Number of free coefficients: 4
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on frequency domain data "dataf".
Fit to estimation data: 87.03% (data prefiltered)
FPE: 0.004832, MSE: 0.003631
```

```matlab
opt = tfestOptions('SearchMethod','lsqnonlin',...
    'WeightingFilter',[0 7]); % Requires Optimization Toolbox
m9 = tfest(dataf,2,opt) % transfer function with 2 poles
```

```
m9 =
  From input "u1" to output "y1":
  -0.01662 s + 1.007
  ------------------
   s^2 + s + 0.995

Continuous-time identified transfer function.

Parameterization:
   Number of poles: 2    Number of zeros: 1
   Number of free coefficients: 4
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using TFEST on frequency domain data "dataf".
Fit to estimation data: 87.03% (data prefiltered)
FPE: 0.00491, MSE: 0.003629
```

```matlab
h = stepplot(m0,m6,m7,m8,m9);
showConfidence(h,1)
legend('show')
```

Step Response

From: u1 To: y1
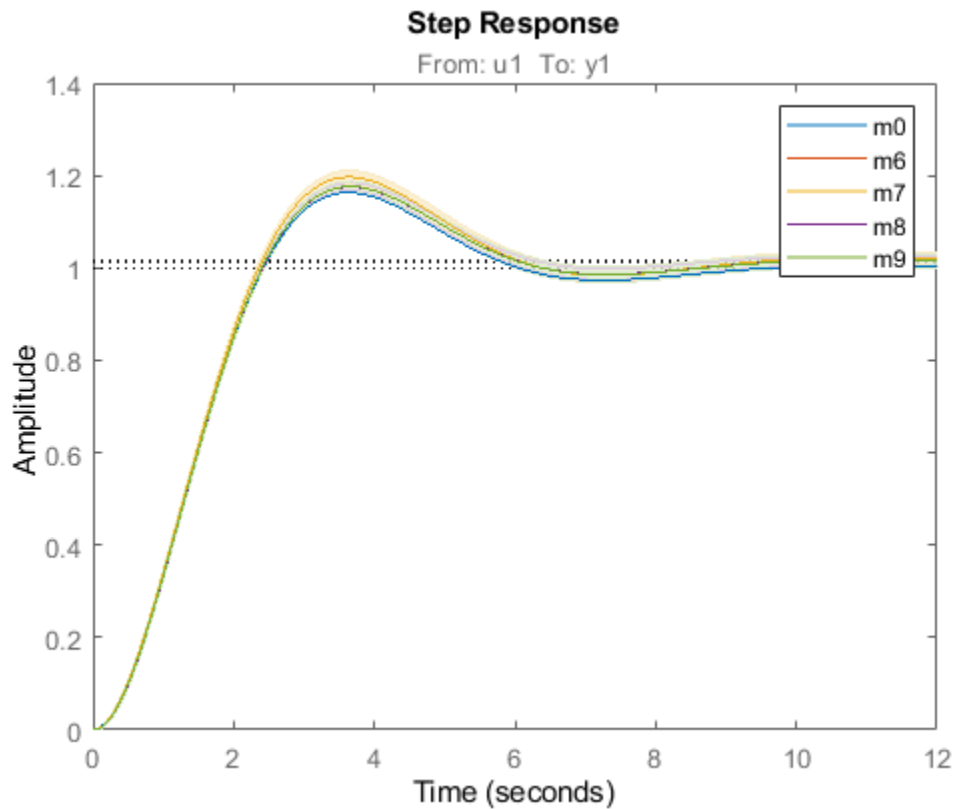
Conclusions

We saw how time, frequency and spectral data can seamlessly be used to estimate a variety of linear models in both continuous and discrete time domains. The models may be validated and compared in domains different from the ones they were estimated in. The data formats (time, frequency and spectrum) are interconvertible, using methods such as `fft`, `ifft`, `spafdr` and `spa`. Furthermore, direct, continuous-time estimation is achievable by using `tfest`, `ssest` and `procest` estimation routines. The seamless use of data in any domain for estimation and analysis is an important feature of System Identification Toolbox.

See Also

`tfest` | `ssest` | `procest` | `oe`

More About

- "Estimating Models Using Frequency-Domain Data"

Building Structured and User-Defined Models Using System Identification Toolbox

This example shows how to estimate parameters in user-defined model structures. Such structures are specified by IDGREY (linear state-space) or IDNLGREY (nonlinear state-space) models. We shall investigate how to assign structure, fix parameters and create dependencies among them.

Experiment Data

We shall investigate data produced by a (simulated) dc-motor. We first load the data:

```
load dcmdata y u
```

The matrix y contains the two outputs: y1 is the angular position of the motor shaft and y2 is the angular velocity. There are 400 data samples and the sample time is 0.1 seconds. The input is contained in the vector u. It is the input voltage to the motor.

```
z = iddata(y,u,0.1); % The IDDATA object
z.InputName = 'Voltage';
z.OutputName = {'Angle';'AngVel'};
plot(z(:,1,:))
```


Figure: Measurement Data: Voltage to Angle

```
plot(z(:,2,:))
```

Figure: Measurement Data: Voltage to Angular Velocity

Model Structure Selection

```
d/dt x = A x + B u + K e
    y   = C x + D u + e
```

We shall build a model of the dc-motor. The dynamics of the motor is well known. If we choose x1 as the angular position and x2 as the angular velocity it is easy to set up a state-space model of the following character neglecting disturbances: (see Example 4.1 in Ljung(1999):

```
          | 0     1  |       | 0   |
d/dt x =  |          | x  +  |     | u
          | 0   -th1 |       | th2 |

       | 1  0 |
  y =  |      | x
       | 0  1 |
```

The parameter `th1` is here the inverse time-constant of the motor and `th2` is such that `th2/th1` is the static gain from input to the angular velocity. (See Ljung(1987) for how `th1` and `th2` relate to the physical parameters of the motor). We shall estimate these two parameters from the observed data. The model structure (parameterized state space) described above can be represented in MATLAB® using IDSS and IDGREY models. These models let you perform estimation of parameters using experimental data.

Specification of a Nominal (Initial) Model

If we guess that th1=1 and th2 = 0.28 we obtain the nominal or initial model

```
A = [0 1; 0 -1]; % initial guess for A(2,2) is -1
B = [0; 0.28]; % initial guess for B(2) is 0.28
C = eye(2);
D = zeros(2,1);
```

and we package this into an IDSS model object:

```
ms = idss(A,B,C,D);
```

The model is characterized by its matrices, their values, which elements are free (to be estimated) and upper and lower limits of those:

```
ms.Structure.a
```

```
ans =

        Name: 'A'
       Value: [2x2 double]
     Minimum: [2x2 double]
     Maximum: [2x2 double]
        Free: [2x2 logical]
       Scale: [2x2 double]
        Info: [2x2 struct]


1x1 param.Continuous
```

```
ms.Structure.a.Value
ms.Structure.a.Free
```

```
ans =

     0     1
     0    -1


ans =

  2x2 logical array

   1   1
   1   1
```

Specification of Free (Independent) Parameters Using IDSS Models

So we should now mark that it is only A(2,2) and B(2,1) that are free parameters to be estimated.

```
ms.Structure.a.Free = [0 0; 0 1];
ms.Structure.b.Free = [0; 1];
ms.Structure.c.Free = 0; % scalar expansion used
```

```
ms.Structure.d.Free = 0;
ms.Ts = 0;  % This defines the model to be continuous
```

The Initial Model

```
ms % Initial model


ms =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
       x1   x2
   x1   0    1
   x2   0   -1

  B =
         u1
   x1     0
   x2  0.28

  C =
       x1   x2
   y1   1    0
   y2   0    1

  D =
         u1
   y1    0
   y2    0

  K =
       y1   y2
   x1   0    0
   x2   0    0

Parameterization:
   STRUCTURED form (some fixed coefficients in  A, B, C).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 2
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

Estimation of Free Parameters of the IDSS Model

The prediction error (maximum likelihood) estimate of the parameters is now computed by:

```
dcmodel = ssest(z,ms,ssestOptions('Display','on'));
dcmodel


dcmodel =
  Continuous-time identified state-space model:
```

```
        dx/dt = A x(t) + B u(t) + K e(t)
         y(t) = C x(t) + D u(t) + e(t)

   A =
            x1        x2
    x1       0         1
    x2       0  -4.013

   B =
        Voltage
    x1       0
    x2    1.002

   C =
            x1  x2
    Angle    1   0
    AngVel   0   1

   D =
           Voltage
    Angle        0
    AngVel       0

   K =
        Angle   AngVel
    x1       0        0
    x2       0        0

Parameterization:
   STRUCTURED form (some fixed coefficients in  A, B, C).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 2
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on time domain data "z".
Fit to estimation data: [98.35;84.42]%
FPE: 0.001071, MSE: 0.1192
```

```
Estimation data: Time domain data z
Data has 2 outputs, 1 inputs and 400 samples.
Number of states: 2
```

ESTIMATION PROGRESS

```
Algorithm: Nonlinear least squares with automatically chosen line search method
--------------------------------------------------------------------------------
-------
                        Norm of      First-order   Improvement (%)
  Iteration     Cost      step        optimality    Expected  Achieved   Bisections
--------------------------------------------------------------------------------
-------
      0       1.89282        -        3.09e+03       145          -          -
      1       0.139814     0.862      2.13e+03       145        92.6         0
      2       0.00639664   1.15       1.73e+03       158        95.4         0
      3       0.0011775    0.848        385          109        81.6         0
      4       0.00106111   0.227         7.9         10.1        9.88        0
      5       0.00106082   0.0126        0.158       0.027      0.0272       0
      6       0.00106082   0.000154      0.00106     4.03e-06   4.07e-06     0
--------------------------------------------------------------------------------
```

RESULT

```
Termination condition: Near (local) minimum, (norm(g) < tol)..
Number of iterations: 6, Number of function evaluations: 13

Status: Estimated using SSEST
Fit to estimation data: [98.35;84.42]%, FPE: 0.00107149
```

■ Stop   Close

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);
```

The resulting model is

```
dcmodel2
```

```
dcmodel2 =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
            x1        x2
    x1       0    0.9975
    x2       0    -4.011

  B =
        Voltage
    x1        0
    x2    1.004
```

```
C =
          x1  x2
  Angle    1   0
  AngVel   0   1

D =
          Voltage
  Angle         0
  AngVel        0

K =
      Angle   AngVel
  x1      0        0
  x2      0        0

Parameterization:
   STRUCTURED form (some fixed coefficients in  A, B, C).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 3
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PEM on time domain data "z".
Fit to estimation data: [98.35;84.42]%
FPE: 0.001077, MSE: 0.1192
```

We find that the estimated `A(1,2)` is close to 1. To compare the two model we use the `compare` command:

`compare(z,dcmodel,dcmodel2)`

Specification of Models with Coupled Parameters Using IDGREY Objects

Suppose that we accurately know the static gain of the dc-motor (from input voltage to angular velocity, e.g. from a previous step-response experiment. If the static gain is G, and the time constant of the motor is t, then the state-space model becomes

```
          |0      1|     |  0  |
d/dt x =  |        |x +  |     |  u
          |0   -1/t|     | G/t |

          |1    0|
   y   =  |      |  x
          |0    1|
```

With G known, there is a dependence between the entries in the different matrices. In order to describe that, the earlier used way with "Free" parameters will not be sufficient. We thus have to write a MATLAB file which produces the A, B, C, and D, and optionally also the K and X0 matrices as outputs, for each given parameter vector as input. It also takes auxiliary arguments as inputs, so that the user can change certain things in the model structure, without having to edit the file. In this case we let the known static gain G be entered as such an argument. The file that has been written has the name 'motorDynamics.m'.

```
type motorDynamics
```

```
function [A,B,C,D,K,X0] = motorDynamics(par,ts,aux)
%MOTORDYNAMICS ODE file representing the dynamics of a motor.
```

```
%
%   [A,B,C,D,K,X0] = motorDynamics(Tau,Ts,G)
%   returns the State Space matrices of the DC-motor with
%   time-constant Tau (Tau = par) and known static gain G. The sample
%   time is Ts.
%
%   This file returns continuous-time representation if input argument Ts
%   is zero. If Ts>0, a discrete-time representation is returned. To make
%   the IDGREY model that uses this file aware of this flexibility, set the
%   value of Structure.FcnType property to 'cd'. This flexibility is useful
%   for conversion between continuous and discrete domains required for
%   estimation and simulation.
%
%   See also IDGREY, IDDEMO7.

%   L. Ljung
%   Copyright 1986-2015 The MathWorks, Inc.

t = par(1);
G = aux(1);

A = [0 1;0 -1/t];
B = [0;G/t];
C = eye(2);
D = [0;0];
K = zeros(2);
X0 = [0;0];
if ts>0 % Sample the model with sample time Ts
   s = expm([[A B]*ts; zeros(1,3)]);
   A = s(1:2,1:2);
   B = s(1:2,3);
end
```

We now create an IDGREY model object corresponding to this model structure: The assumed time constant will be

```
par_guess = 1;
```

We also give the value 0.25 to the auxiliary variable G (gain) and sample time.

```
aux = 0.25;
dcmm = idgrey('motorDynamics',par_guess,'cd',aux,0);
```

The time constant is now estimated by

```
dcmm = greyest(z,dcmm,greyestOptions('Display','on'));
```

We have thus now estimated the time constant of the motor directly. Its value is in good agreement with the previous estimate.

```
dcmm
```

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

```
 A =
           x1       x2
  x1        0        1
  x2        0   -4.006

 B =
      Voltage
  x1        0
  x2    1.001

 C =
           x1  x2
  Angle     1   0
  AngVel    0   1

 D =
         Voltage
  Angle        0
  AngVel       0

 K =
       Angle  AngVel
  x1       0       0
  x2       0       0

 Model parameters:
  Par1 = 0.2496

Parameterization:
   ODE Function: motorDynamics
   (parametrizes both continuous- and discrete-time equations)
   Disturbance component: parameterized by the ODE function
   Initial state: parameterized by the ODE function
   Number of free coefficients: 1
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using GREYEST on time domain data "z".
Fit to estimation data: [98.35;84.42]%
FPE: 0.00107, MSE: 0.1193
```

With this model we can now proceed to test various aspects as before. The syntax of all the commands is identical to the previous case. For example, we can compare the idgrey model with the other state-space model:

```
compare(z,dcmm,dcmodel)
```

Simulated Response Comparison

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:

$$a_{kl}(q) = 1 + a_1 q^{-1} + \dots + a_{nakl} q^{-nakl} q$$

It is thus a polynomial in `1/q` of degree `nakl`.

Similarly B(q) is a ny | nu matrix, whose kj-element is:

$$b_{kj}(q) = b_0 q^{-nkk} + b_1 q^{-nkkj-1} + \dots + b_{nbkj} q^{-nkkj-nbkj}$$

There is thus a delay of `nkkj` from input number `j` to output number `k`. The most common way to create those would be to use the ARX-command. The orders are specified as: `nn = [na nb nk]` with `na` being a `ny-by-ny` matrix whose `kj`-entry is `nakj`; `nb` and `nk` are defined similarly.

Let's test some ARX-models on the dc-data. First we could simply build a general second order model:

```
dcarx1 = arx(z,'na',[2,2;2,2],'nb',[2;2],'nk',[1;1])
```

```
dcarx1 =
Discrete-time ARX model:
  Model for output "Angle": A(z)y_1(t) = - A_i(z)y_i(t) + B(z)u(t) + e_1(t)
    A(z) = 1 - 0.5545 z^-1 - 0.4454 z^-2

    A_2(z) = -0.03548 z^-1 - 0.06405 z^-2

    B(z) = 0.004243 z^-1 + 0.006589 z^-2

  Model for output "AngVel": A(z)y_2(t) = - A_i(z)y_i(t) + B(z)u(t) + e_2(t)
    A(z) = 1 - 0.2005 z^-1 - 0.2924 z^-2

    A_1(z) = 0.01849 z^-1 - 0.01937 z^-2

    B(z) = 0.08642 z^-1 + 0.03877 z^-2

Sample time: 0.1 seconds

Parameterization:
   Polynomial orders:   na=[2 2;2 2]   nb=[2;2]   nk=[1;1]
   Number of free coefficients: 12
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using ARX on time domain data "z".
Fit to estimation data: [97.87;83.44]% (prediction focus)
FPE: 0.002157, MSE: 0.1398
```

The result, `dcarx1`, is stored as an IDPOLY model, and all previous commands apply. We could for example explicitly list the ARX-polynomials by:

```
dcarx1.a
```

```
ans =

  2x2 cell array

    {[1 -0.5545 -0.4454]}    {[0 -0.0355 -0.0640]}
    {[ 0 0.0185 -0.0194]}    {[1 -0.2005 -0.2924]}
```

as cell arrays where e.g. the {1,2} element of dcarx1.a is the polynomial A(1,2) described earlier, relating y2 to y1.

We could also test a structure, where we know that `y1` is obtained by filtering `y2` through a first order filter. (The angle is the integral of the angular velocity). We could then also postulate a first order dynamics from input to output number 2:

```
na = [1 1; 0 1];
nb = [0 ; 1];
nk = [1 ; 1];
dcarx2 = arx(z,[na nb nk])


dcarx2 =
Discrete-time ARX model:
  Model for output "Angle": A(z)y_1(t) = - A_i(z)y_i(t) + B(z)u(t) + e_1(t)
    A(z) = 1 - 0.9992 z^-1

    A_2(z) = -0.09595 z^-1

    B(z) = 0

  Model for output "AngVel": A(z)y_2(t) = B(z)u(t) + e_2(t)
    A(z) = 1 - 0.6254 z^-1

    B(z) = 0.08973 z^-1

Sample time: 0.1 seconds

Parameterization:
   Polynomial orders:   na=[1 1;0 1]   nb=[0;1]   nk=[1;1]
   Number of free coefficients: 4
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using ARX on time domain data "z".
Fit to estimation data: [97.52;81.46]% (prediction focus)
FPE: 0.003452, MSE: 0.177
```

To compare the different models obtained we use

```
compare(z,dcmodel,dcmm,dcarx2)
```

Simulated Response Comparison

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')
```

Bode Diagram

From: Voltage To: Angle

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.

Estimating Simple Models from Real Laboratory Process Data

This example shows how to develop and analyze simple models from a real laboratory process data. We start with a small description of the process, learn how to import the data to the toolbox and preprocess/condition it and then proceed systematically to estimate parametric and nonparametric models. Once the models have been identified we compare the estimated models and also validate the model to the actual output data from the experiment.

System Description

This case study concerns data collected from a laboratory scale "hairdryer". (Feedback's Process Trainer PT326; See also page 525 in Ljung, 1999). The process works as follows: Air is fanned through a tube and heated at the inlet. The air temperature is measured by a thermocouple at the outlet. The input is the voltage over the heating device, which is just a mesh of resistor wires. The output is the outlet air temperature represented by the measured thermocouple voltage.

Setting up Data for Analysis

First we load the input-output data to the MATLAB® Workspace.

```
load dryer2;
```

Vector y2, the output, contains 1000 measurements of the thermocouple voltage which is proportional to the temperature in the outlet airstream. Vector u2 contains 1000 input data points consisting of the voltage applied to the heater. The input was generated as a binary random sequence that switches from one level to the other with probability 0.2. The sample time is 0.08 seconds.

The next step is to set up the data as an iddata object

```
dry = iddata(y2,u2,0.08);
```

To get information about the data, just type the name of the `iddata` object at the MATLAB command window:

```
dry
```

```
dry =

Time domain data set with 1000 samples.
Sample time: 0.08 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

To inspect the properties of the above iddata object, use the `get` command:

```
get(dry)
```

```
ans =
```

```
struct with fields:

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

For better book-keeping, it is good practice to give names to the input and output channels and Time units. These names would be propagated throughout the analysis of this iddata object:

```
dry.InputName = 'Heater Voltage';
dry.OutputName = 'Thermocouple Voltage';
dry.TimeUnit = 'seconds';
dry.InputUnit = 'V';
dry.OutputUnit = 'V';
```

Now that we have the data set ready, we choose the first 300 data points for model estimation.

```
ze = dry(1:300)
```

```
ze =

Time domain data set with 300 samples.
Sample time: 0.08 seconds

Outputs                  Unit (if specified)
   Thermocouple Voltage      V

Inputs                   Unit (if specified)
   Heater Voltage            V
```

Preprocessing the Data

Plot the interval from sample 200 to 300:

```
plot(ze(200:300));
```
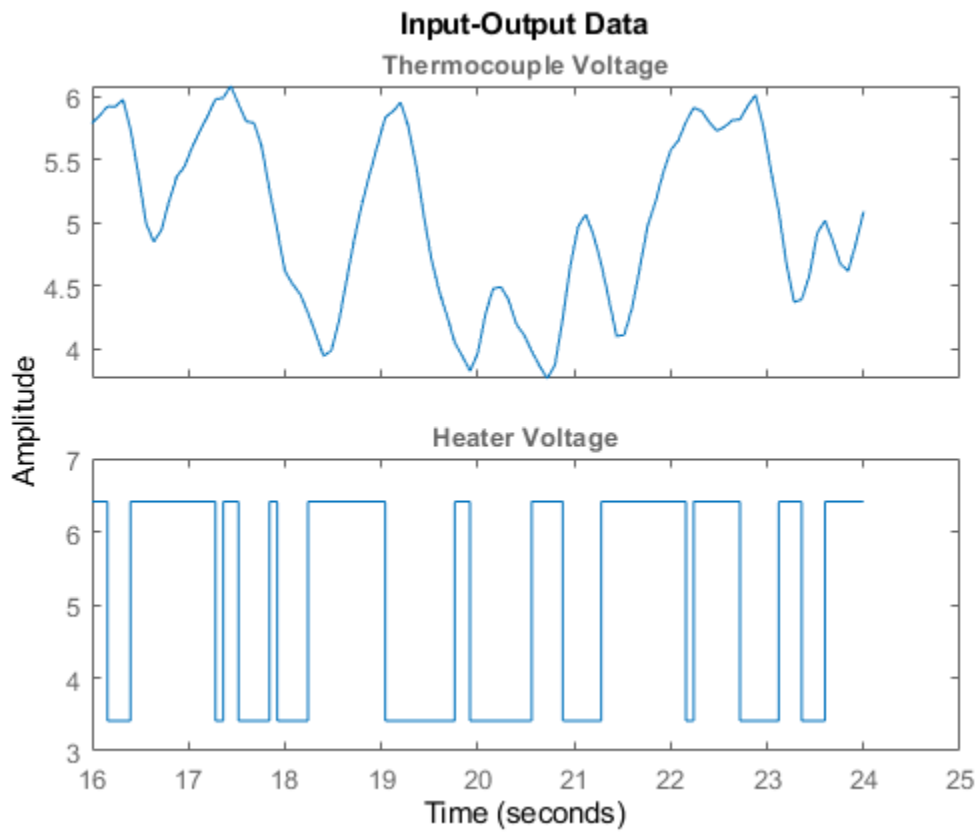
Figure 1: A snapshot of the measured hair-dryer data.

From the above plot, it can be observed that the data is not zero mean. So let us remove the constant levels and make the data zero mean.

```
ze = detrend(ze);
```

The same data set after it has been detrended:

```
plot(ze(200:300)) %show samples from 200 to 300 of detrended data
```
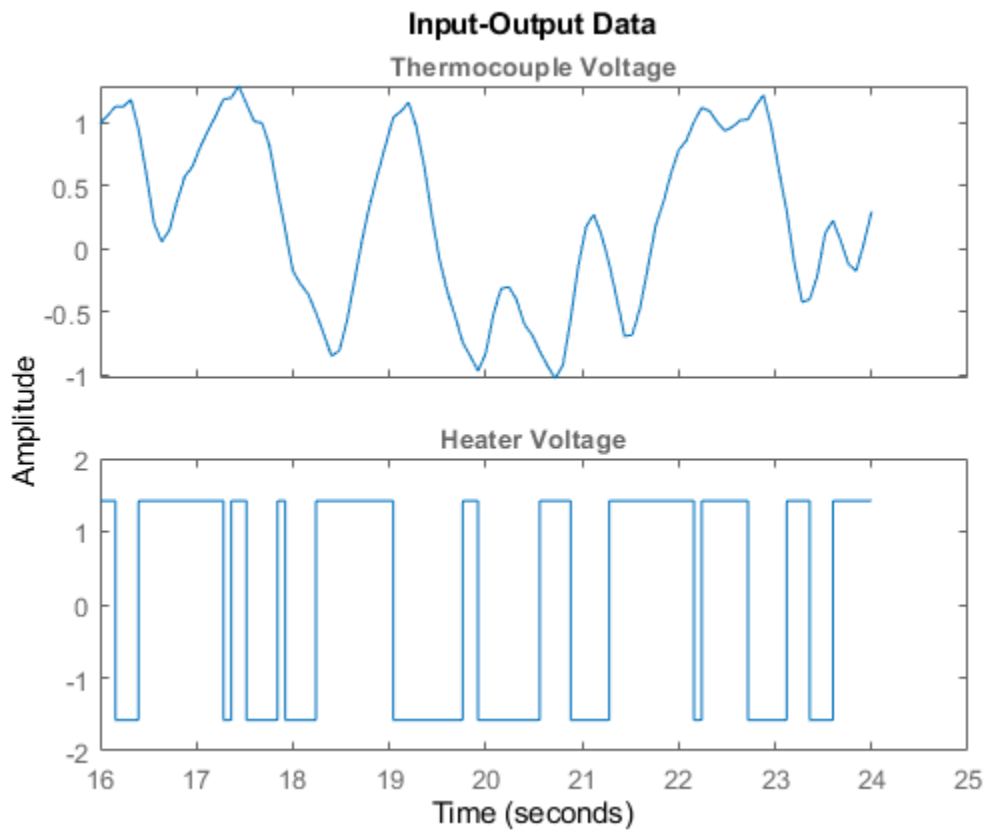
Figure 2: Detrended estimation data.

Estimating Nonparametric and Parametric Models

Now that the dataset has been detrended and there are no obvious outliers, let us first estimate the impulse response of the system by correlation analysis to get some idea of time constants and the like:

```
clf
mi = impulseest(ze); % non-parametric (FIR) model
showConfidence(impulseplot(mi),3); %impulse response with 3 standard
                                   %deviations confidence region
```
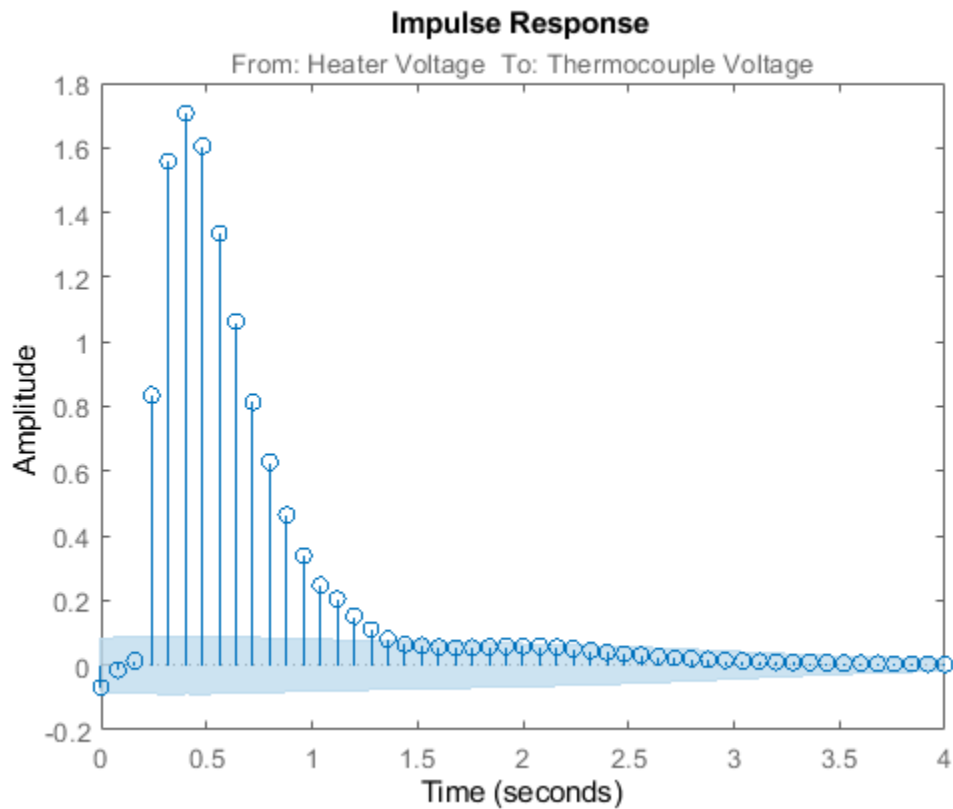
Figure 3: Impulse response of the FIR model estimated using `ze`.

The shaded region marks a 99.7% confidence interval. There is a time delay (dead-time) of 3 samples before the output responds to the input (significant output outside the confidence interval).

The simplest way to get started on a parametric estimation routine is to build a state-space model where the model-order is automatically determined, using a prediction error method. Let us estimate a model using the `ssest` estimation technique:

```
m1 = ssest(ze);
```

`m1` is a continuous-time identified state-space model, represented by an `idss` object. The estimation algorithm chooses 3 as the optimal order of the model. To inspect the properties of the estimated model, just enter the model name at the command window:

```
m1
```

```
m1 =
  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         x3
     x1  -0.4839    -2.011      2.092
     x2   3.321     -1.913      5.998
```

```
    x3     1.623    -17.01    -15.61

  B =
        Heater Volta
    x1      -0.05753
    x2       0.02004
    x3         1.377

  C =
                      x1       x2        x3
    Thermocouple   -14.07  0.07729   0.04252

  D =
                  Heater Volta
    Thermocouple           0

  K =
        Thermocouple
    x1        -0.9457
    x2        -0.02097
    x3          2.102

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: estimate
   Number of free coefficients: 18
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on time domain data "ze".
Fit to estimation data: 95.32% (prediction focus)
FPE: 0.001621, MSE: 0.001526
```

The display suggests that the model is free-form (all entries of A, B and C matrices were treated as free parameters) and that the estimated model fits the data pretty well (over 90% fit). To retrieve the properties of this model, for example to obtain the A matrix of the discrete state-space object generated above, we can use the dot operator:

```
    A = m1.a;
```

See the "Data and Model Objects in System Identification Toolbox" example for more information regarding model objects. To find out which properties of the model object can be retrieved, use `get` command:

`get(m1)`

```
               A: [3x3 double]
               B: [3x1 double]
               C: [-14.0706 0.0773 0.0425]
               D: 0
               K: [3x1 double]
       StateName: {3x1 cell}
       StateUnit: {3x1 cell}
       Structure: [1x1 pmodel.ss]
   NoiseVariance: 1.2587e-04
      InputDelay: 0
```

```
        OutputDelay: 0
          InputName: {'Heater Voltage'}
          InputUnit: {'V'}
         InputGroup: [1x1 struct]
         OutputName: {'Thermocouple Voltage'}
         OutputUnit: {'V'}
        OutputGroup: [1x1 struct]
              Notes: [0x1 string]
           UserData: []
               Name: ''
                 Ts: 0
           TimeUnit: 'seconds'
       SamplingGrid: [1x1 struct]
             Report: [1x1 idresults.ssest]
```

To fetch the values of the state-space matrices and their 1 standard deviation uncertainties, use the `idssdata` command:

```
[A,B,C,D,K,~,dA,dB,dC,dD,dK] = idssdata(m1)
```

```
A =

    -0.4839    -2.0112     2.0917
     3.3205    -1.9135     5.9981
     1.6235   -17.0096   -15.6070


B =

    -0.0575
     0.0200
     1.3770


C =

   -14.0706     0.0773     0.0425


D =

      0


K =

    -0.9457
    -0.0210
     2.1019


dA =

   1.0e+14 *

     1.4768     1.0316     0.7123
```

```
   3.2030    1.9017    1.8636
   9.8209    5.1241    1.9882


dB =

   1.0e+13 *

   0.3647
   0.9753
   4.0183


dC =

   1.0e+14 *

   2.0691    1.5828    0.3733


dD =

      0


dK =

   1.0e+13 *

   1.3848
   4.0568
   8.1785
```

The uncertainties are quite large even though the model fit the estimation data well. This is because the model is over-parameterized, that is, it has more free parameters than what could be uniquely identified. The variance of parameters in such cases is not well defined. However this does not imply that the model is unreliable. We can plot the time- and frequency-response of this plot and view the variance as confidence regions as discussed next.

Analyzing the Estimated Model

The Bode plot of the generated model can be obtained using the bode function as shown below:

```
h = bodeplot(m1);
```

Bode Diagram

From: Heater Voltage To: Thermocouple Voltage
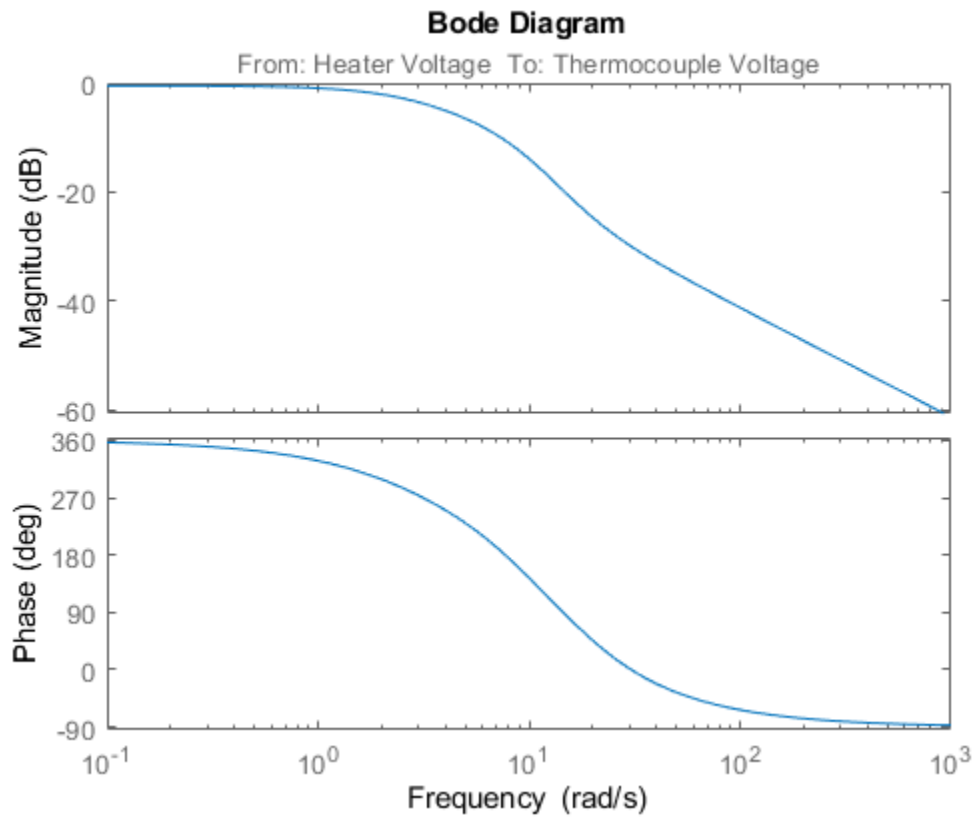
Figure 4: Bode plot of estimated model.

Right-click on the plot and pick Characteristics->Confidence Region. Or, use the `showConfidence` command to view the variance of the response.

```
showConfidence(h,3) % 3 standard deviation (99.7%) confidence region
```
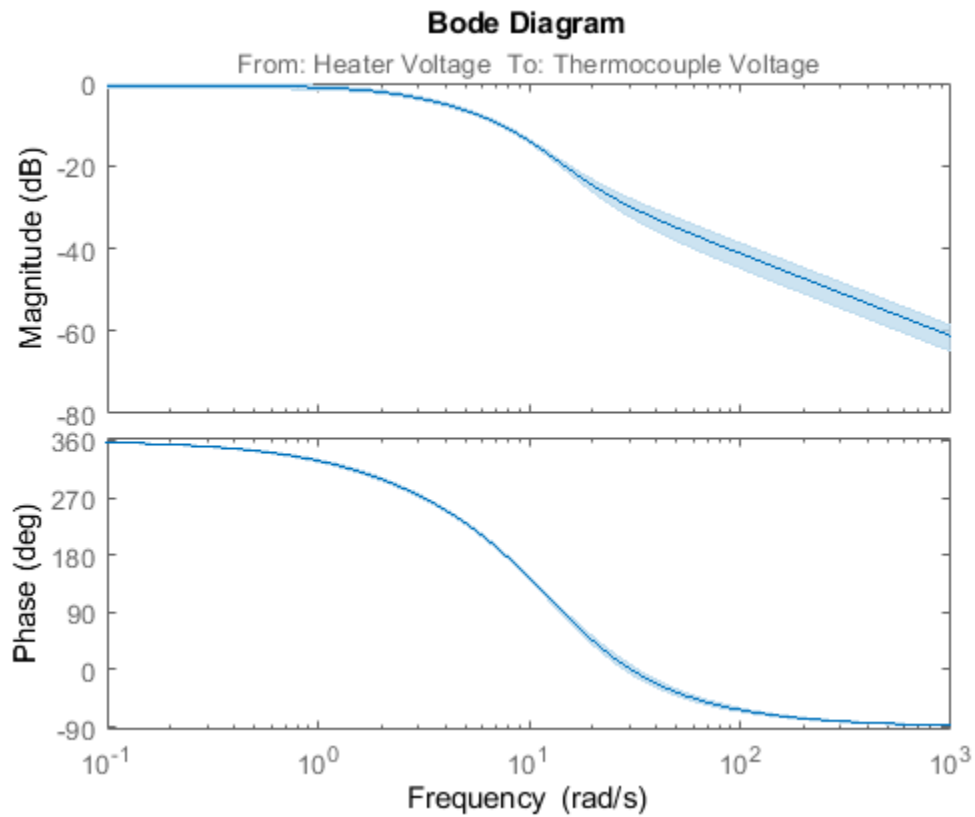
Figure 5: Bode plot with 3 standard deviation confidence region.

Similarly, we can generate the step plot and its associated 3 standard deviation confidence region. We can compare the responses and associated variances of the parametric model `m1` with that of the nonparametric model `mi`:

```
showConfidence(stepplot(m1,'b',mi,'r',3),3)
```
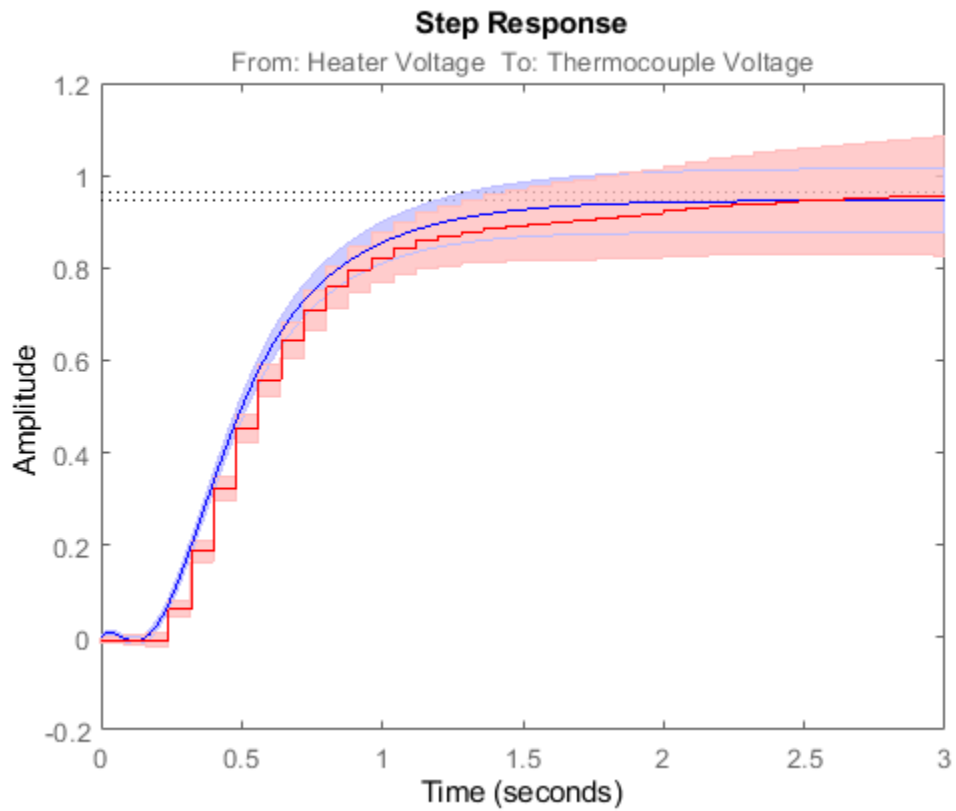
Figure 6: Step plot of models `m1` and `mi` with confidence regions.

We can also consider the Nyquist plot, and mark uncertainty regions at certain frequencies with ellipses, corresponding to 3 standard deviations:

```
Opt = nyquistoptions;
Opt.ShowFullContour = 'off';
showConfidence(nyquistplot(m1,Opt),3)
```
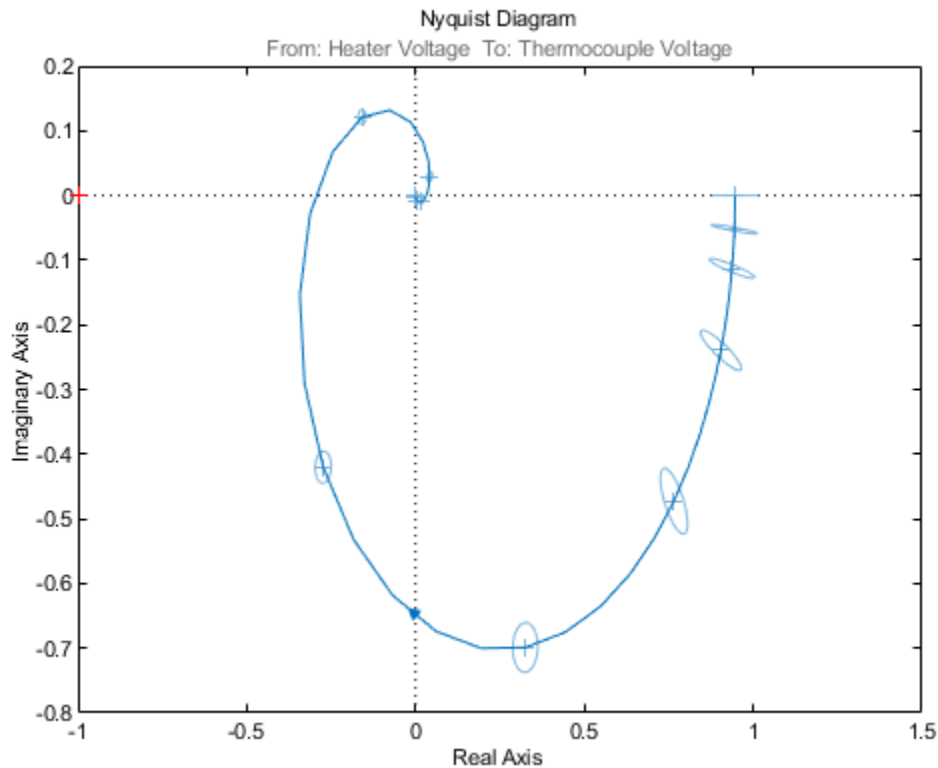
Figure 7: Nyquist plot of estimated model showing the uncertainty regions at certain frequencies.

The response plots show that the estimated model `m1` is quite reliable.

Estimating Models with a Prescribed Structure

System Identification Toolbox can also be used to obtain a model with a prescribed structure. For example, a difference equation model with 2 poles, 1 zero and 3 sample delays can be obtained using the `arx` function as shown below:

```
m2 = arx(ze,[2 2 3]);
```

To look at the model, enter the model name at the command window.

```
m2
```

```
m2 =
Discrete-time ARX model: A(z)y(t) = B(z)u(t) + e(t)
  A(z) = 1 - 1.274 z^-1 + 0.3942 z^-2

  B(z) = 0.06679 z^-3 + 0.04429 z^-4

Sample time: 0.08 seconds

Parameterization:
   Polynomial orders:    na=2    nb=2    nk=3
   Number of free coefficients: 4
```

```
    Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using ARX on time domain data "ze".
Fit to estimation data: 95.08% (prediction focus)
FPE: 0.001756, MSE: 0.001687
```

A continuous time transfer function with 2 poles, one zero and 0.2 second transport delay can be estimated using the `tfest` command:

```
m3 = tfest(ze, 2, 1, 0.2)


m3 =
  From input "Heater Voltage" to output "Thermocouple Voltage":
                1.183 s + 26.55
  exp(-0.2*s) * --------------------
                s^2 + 11.61 s + 28.63

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 time domain data "ze".
Fit to estimation data: 88.79%
FPE: 0.009126, MSE: 0.008768
```

Validating the Estimated Model to Experimental Output

How good is an estimated model? One way to find out is to simulate it and compare the model output with measured output. Select a portion of the original data that was not used in building the model, say from samples 800 to 900. Once the validation data has been preprocessed, we use the `compare` function as shown below to view the quality of prediction:

```
zv = dry(800:900);   % select an independent data set for validation
zv = detrend(zv);    % preprocess the validation data
set(gcf,'DefaultLegendLocation','best')
compare(zv,m1);      % perform comparison of simulated output
```
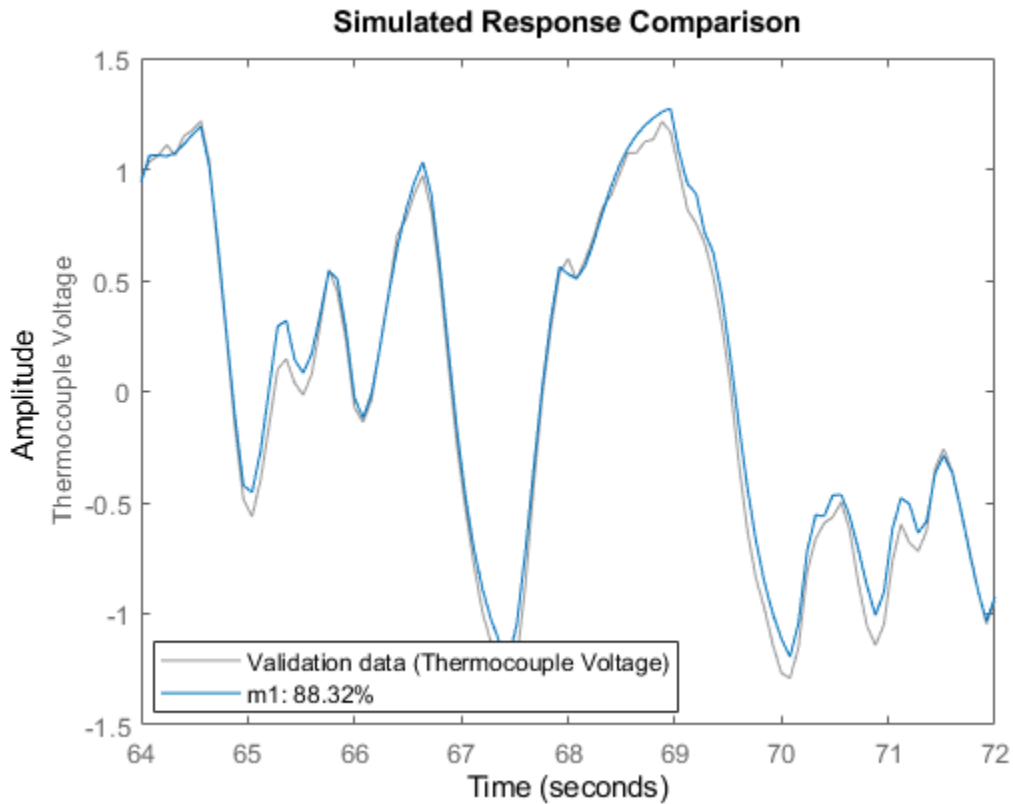
Figure 8: Model's simulated response vs. validation data output.

It can be observed here that the agreement is very good. The "Fit" value shown is calculated as:

```
Fit = 100*(1 - norm(yh - y)/norm(y-mean(y)))
```

where `y` is the measured output (=|zv.y|), and `yh` is the output of the model `m1`.

Comparing Estimated Models

To compare the performance of the models that we have estimated, for example `m1`, `m2` and `m3` with the validation data `zv`, we can again use the `compare` command:

```
compare(zv,m1,'b',m2,'r',m3,'c');
```
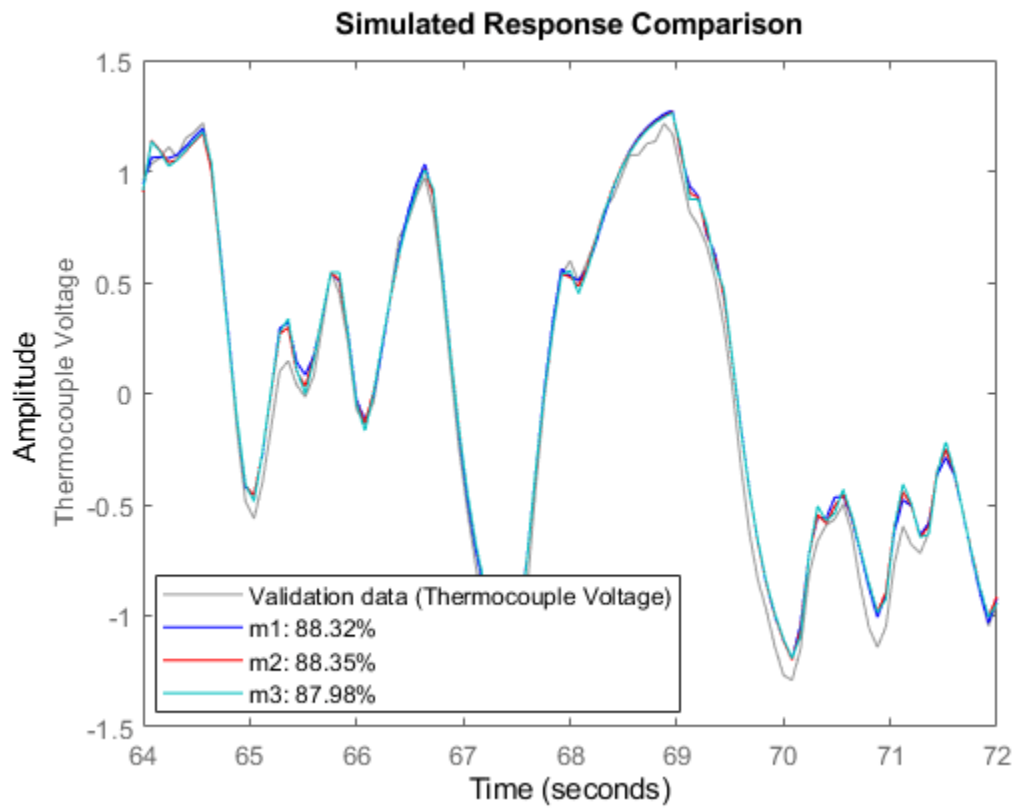
Figure 9: Comparing the responses of models m1, m2, m3 on validation data set ze.

The pole-zero plots for the models can be obtained using `iopzplot`:

```
h = iopzplot(m1,'b',m2,'r',m3,'c');
```
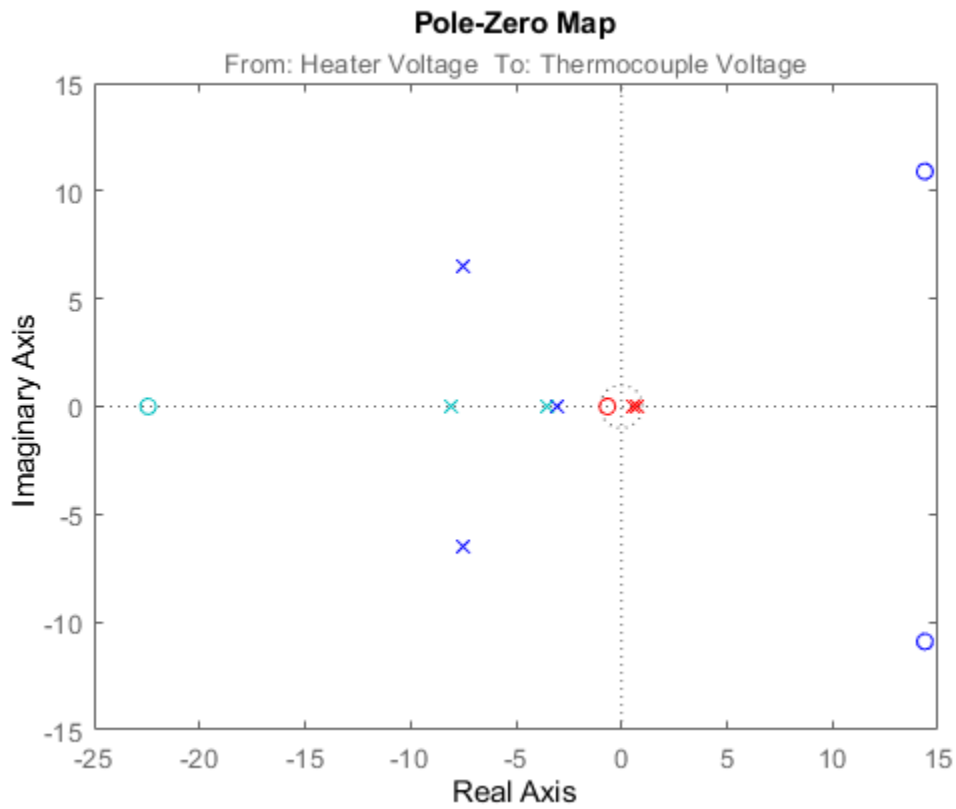
Figure 10: Poles and zeros of the models m1, m2 and m3.

The uncertainties in the poles and zeroes can also be obtained. In the following statement, '3' refers to the number of standard deviations.

```
showConfidence(h,3);
```
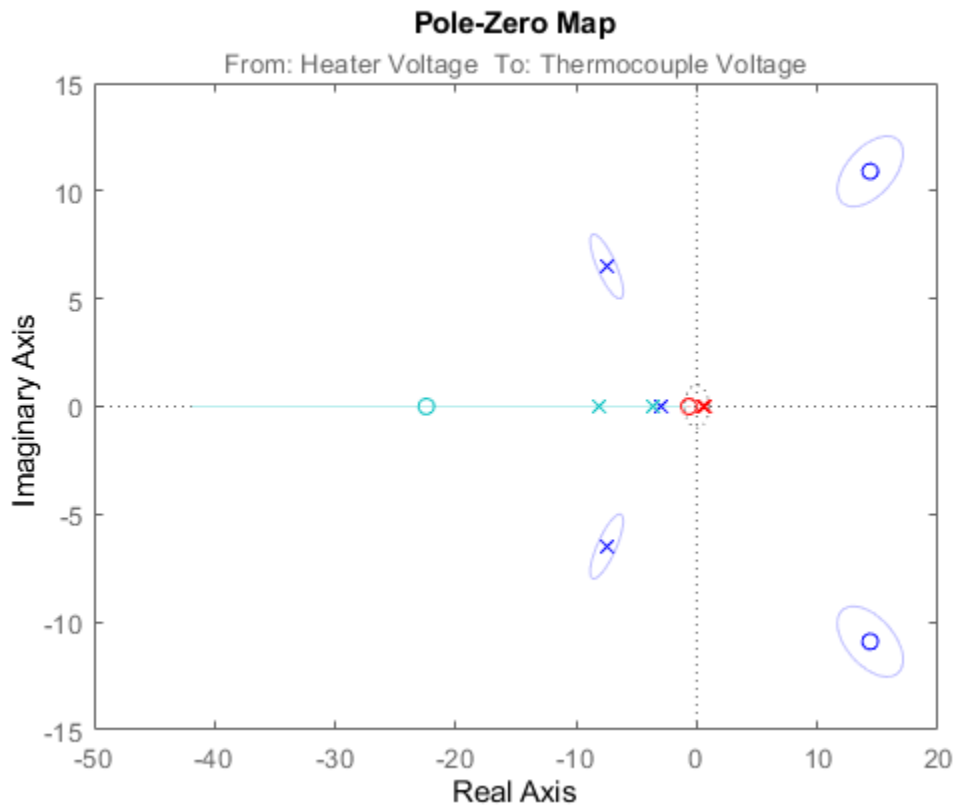
Figure 11: Pole-zero map with uncertainty regions.

The frequency functions above that are obtained from the models can be compared with one that is obtained using a non-parametric spectral analysis method (`spa`):

```
gs = spa(ze);
```

The `spa` command produces an IDFRD model. The `bode` function can again be used for a comparison with the transfer functions of the models obtained.

```
w = linspace(0.4,pi/m2.Ts,200);
opt = bodeoptions; opt.PhaseMatching = 'on';
bodeplot(m1,'b',m2,'r',m3,'c',gs,'g',w,opt);
legend('m1','m2','m3','gs')
```
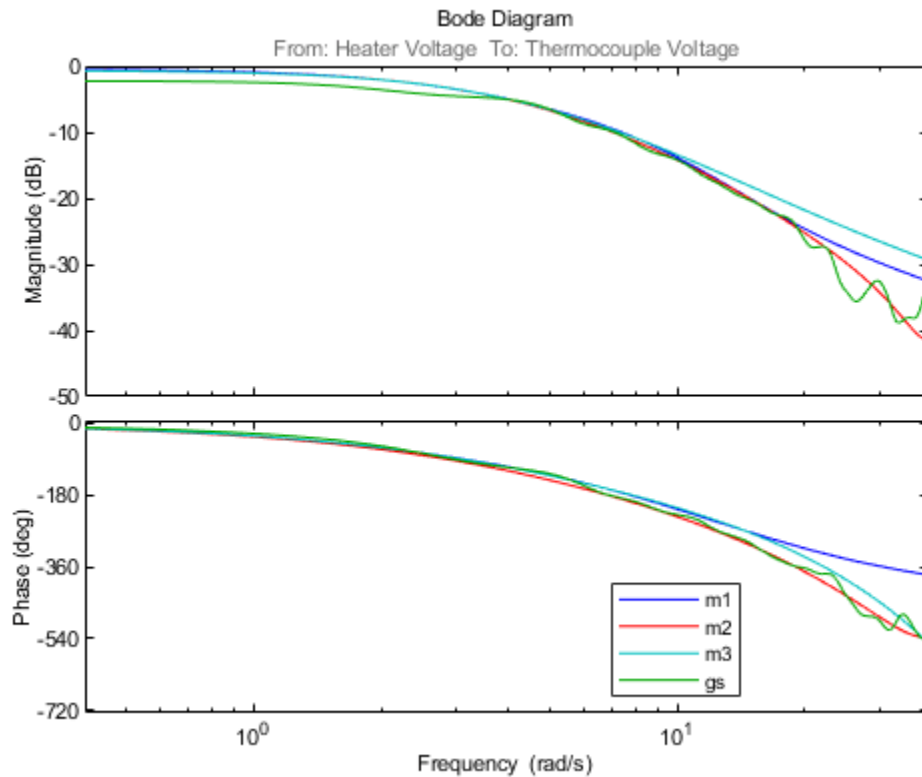
Figure 12: Bode responses of `m1`, `m2` and `m3` compared against the non-parametric spectral estimation model `gs`.

The frequency responses from the three models/methods are very close. This indicates that this response is reliable.

Also, a Nyquist plot can be analyzed with the uncertainty regions marked at certain frequencies:

```
showConfidence(nyquistplot(m1,'b',m2,'r',m3,'c',gs,'g'),3)
```
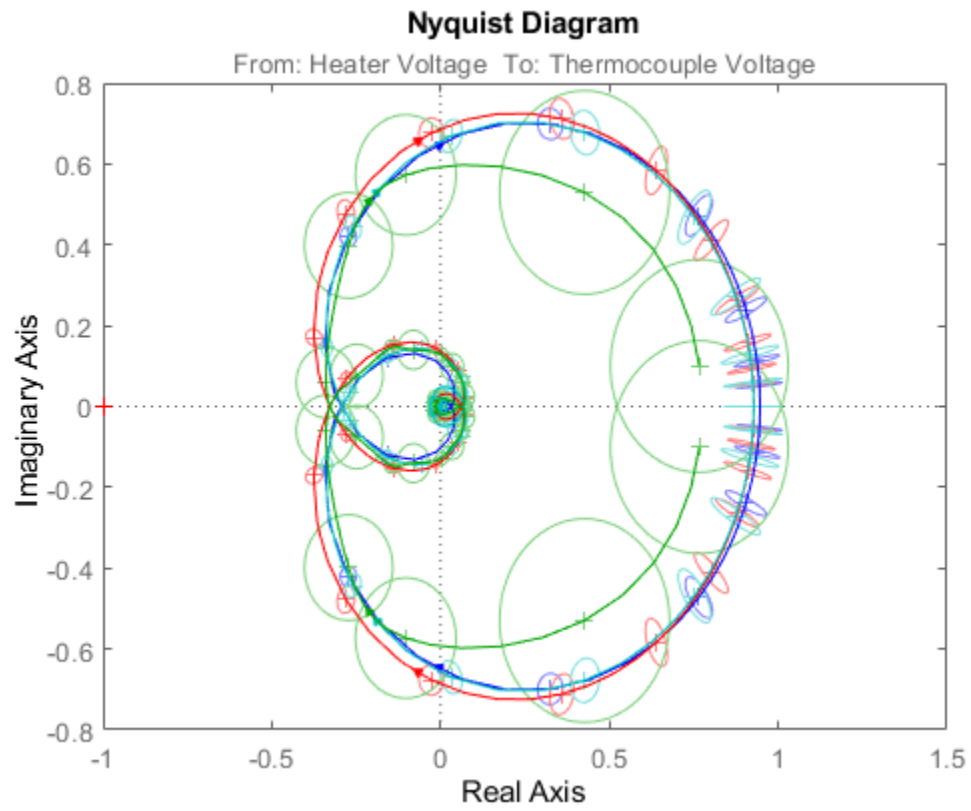
Figure 13: Nyquist plots of models m1, m2, m3 and gs.

The non-parametric model gs exhibits the most uncertainty in response.

Data and Model Objects in System Identification Toolbox

This example shows how to manage data and model objects available in the System Identification Toolbox™. System identification is about building models from data. A data set is characterized by several pieces of information: The input and output signals, the sample time, the variable names and units, etc. Similarly, the estimated models contain information of different kinds - estimated parameters, their covariance matrices, model structure and so on.

This means that it is suitable and desirable to package relevant information around data and models into objects. System Identification Toolbox contains a number of such objects, and the basic features of these are described in this example.

The IDDATA Object

First create some data:

```
u = sign(randn(200,2)); % 2 inputs
y = randn(200,1);        % 1 output
ts = 0.1;                % The sample time
```

To collect the input and the output in one object do

```
z = iddata(y,u,ts);
```

The data information is displayed by just typing its name:

```
z
```

```
z =

Time domain data set with 200 samples.
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
   u2
```

The data is plotted as iddata by the `plot` command, as in `plot(z)`. Press a key to continue and advance between the subplots. Here, we plot the channels separately:

```
plot(z(:,1,1)) % Data subset with Input 1 and Output 1.
```

Input-Output Data
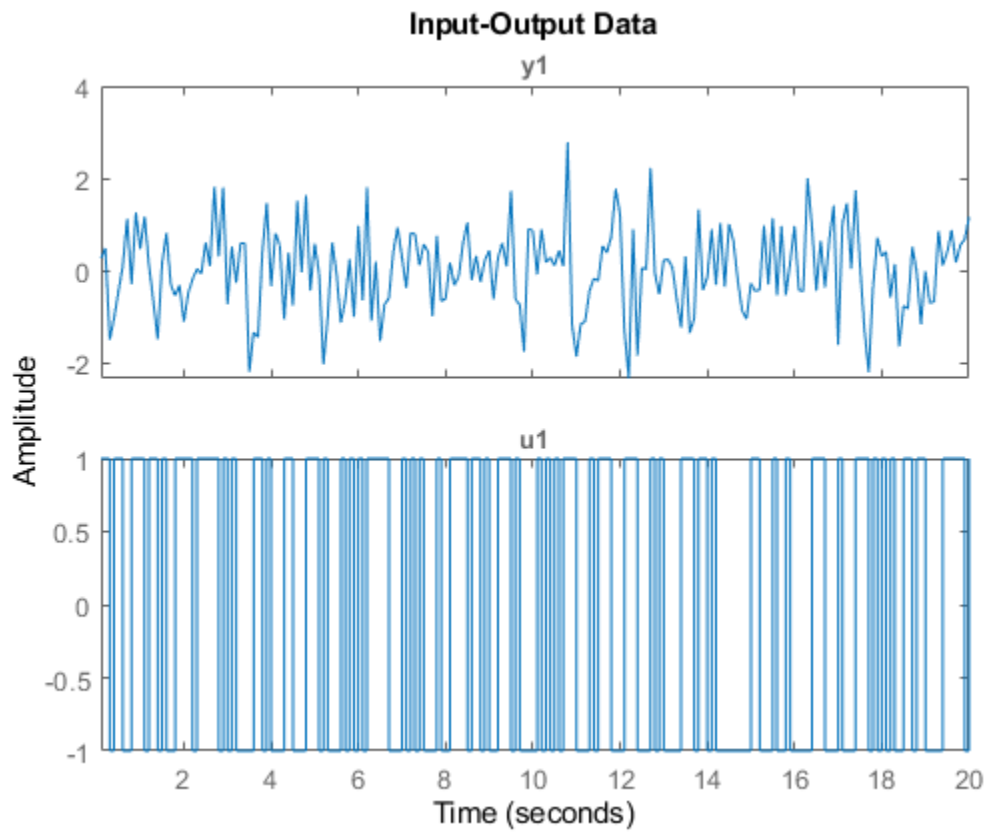
```
plot(z(:,1,2)) % Data subset with Input 2 and Output 1.
```
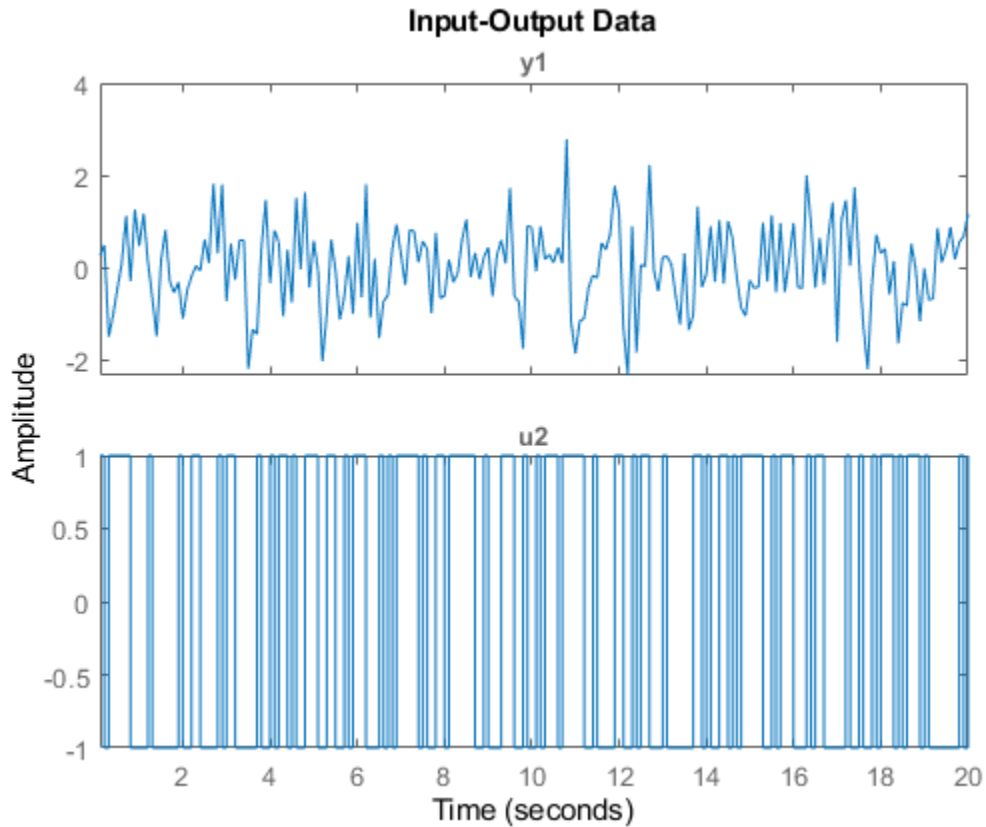
To retrieve the outputs and inputs, use

```
u = z.u;    % or, equivalently u = get(z,'u');
y = z.y;    % or, equivalently y = get(z,'y');
```

To select a portion of the data:

```
zp = z(48:79);
```

To select the first output and the second input:

```
zs = z(:,1,2);   % The ':' refers to all the data time points.
```

The sub-selections can be combined:

```
plot(z(45:54,1,2)) % samples 45 to 54 of response from second input to the first output.
```

Input-Output Data
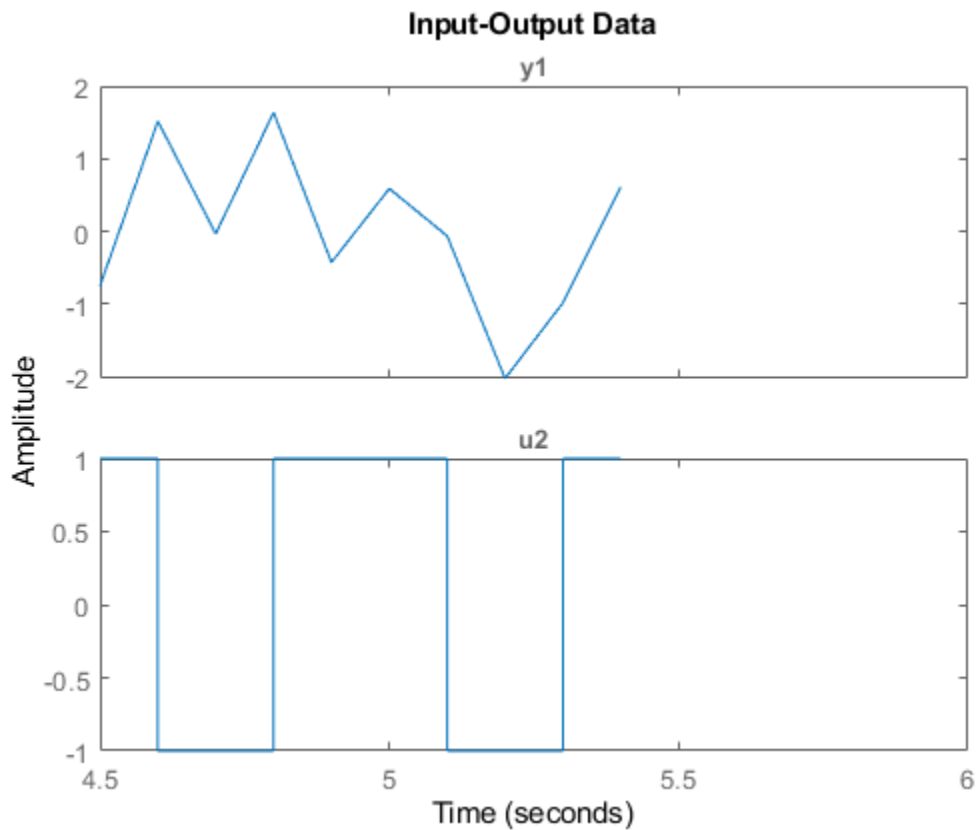
The channels are given default names 'y1', 'u2', etc. This can be changed to any values by

```
set(z,'InputName',{'Voltage';'Current'},'OutputName','Speed');
```

Equivalently we could write

```
z.inputn = {'Voltage';'Current'}; % Autofill is used for properties
z.outputn = 'Speed';     % Upper and lower cases are also ignored
```

For bookkeeping and plots, also units can be set:

```
z.InputUnit = {'Volt';'Ampere'};
z.OutputUnit = 'm/s';
z


z =

Time domain data set with 200 samples.
Sample time: 0.1 seconds

Outputs       Unit (if specified)
   Speed          m/s

Inputs        Unit (if specified)
   Voltage        Volt
   Current        Ampere
```

All current properties are (as for any object) obtained by get:

```
get(z)

ans =

  struct with fields:

             Domain: 'Time'
               Name: ''
         OutputData: [200x1 double]
                  y: 'Same as OutputData'
         OutputName: {'Speed'}
         OutputUnit: {'m/s'}
          InputData: [200x2 double]
                  u: 'Same as InputData'
          InputName: {2x1 cell}
          InputUnit: {2x1 cell}
             Period: [2x1 double]
        InterSample: {2x1 cell}
                 Ts: 0.1000
             Tstart: 0.1000
   SamplingInstants: [200x1 double]
           TimeUnit: 'seconds'
     ExperimentName: 'Exp1'
              Notes: {}
           UserData: []
```

In addition to the properties discussed so far, we have 'Period' which denotes the period of the input if periodic Period = inf means a non-periodic input:

```
z.Period

ans =

   Inf
   Inf
```

The intersample behavior of the input may be given as 'zoh' (zero-order-hold, i.e. piecewise constant) or 'foh' (first- order-hold, i.e., piecewise linear). The identification routines use this information to adjust the algorithms.

```
z.InterSample

ans =

  2x1 cell array

    {'zoh'}
    {'zoh'}
```

You can add channels (both input and output) by "horizontal concatenation", i.e. z = [z1 z2]:

4-115

```
z2 = iddata(rand(200,1),ones(200,1),0.1,'OutputName','New Output',...
    'InputName','New Input');
z3 = [z,z2]


z3 =

Time domain data set with 200 samples.
Sample time: 0.1 seconds

Outputs            Unit (if specified)
    Speed              m/s
    New Output

Inputs             Unit (if specified)
    Voltage            Volt
    Current            Ampere
    New Input
```

Let us plot some of the channels of z3:

```
plot(z3(:,1,1)) % Data subset with Input 2 and Output 1.
```


Input-Output Data
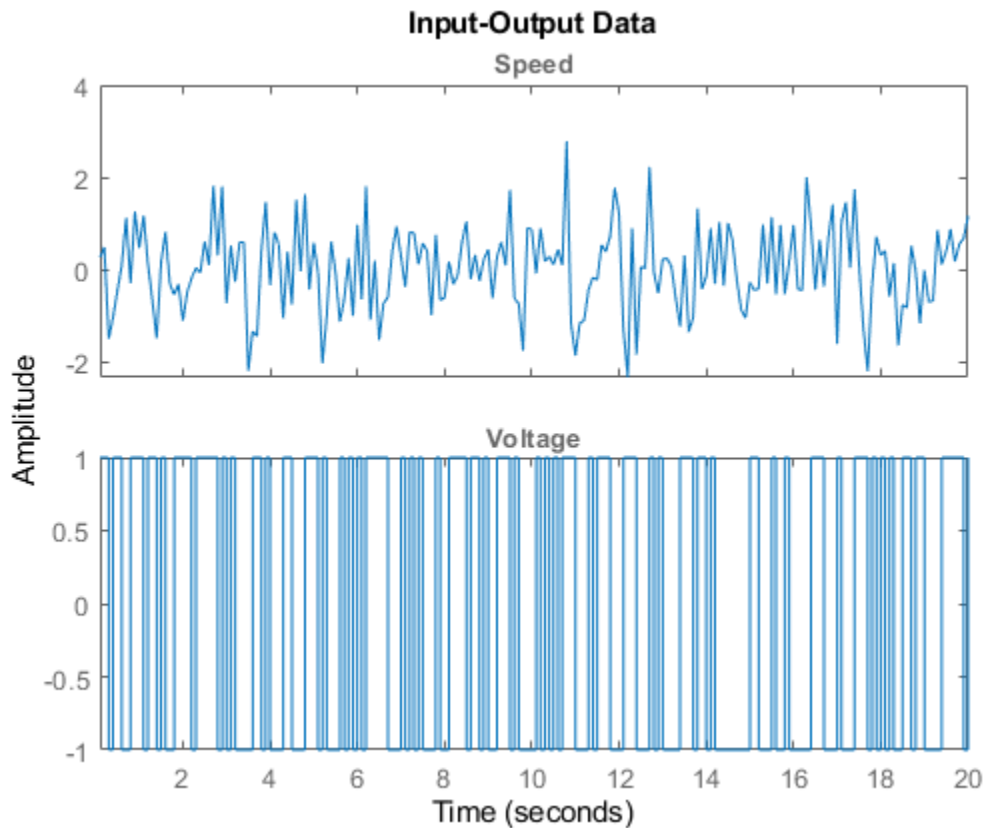
```
plot(z3(:,2,3)) % Data subset with Input 2 and Output 3.
```
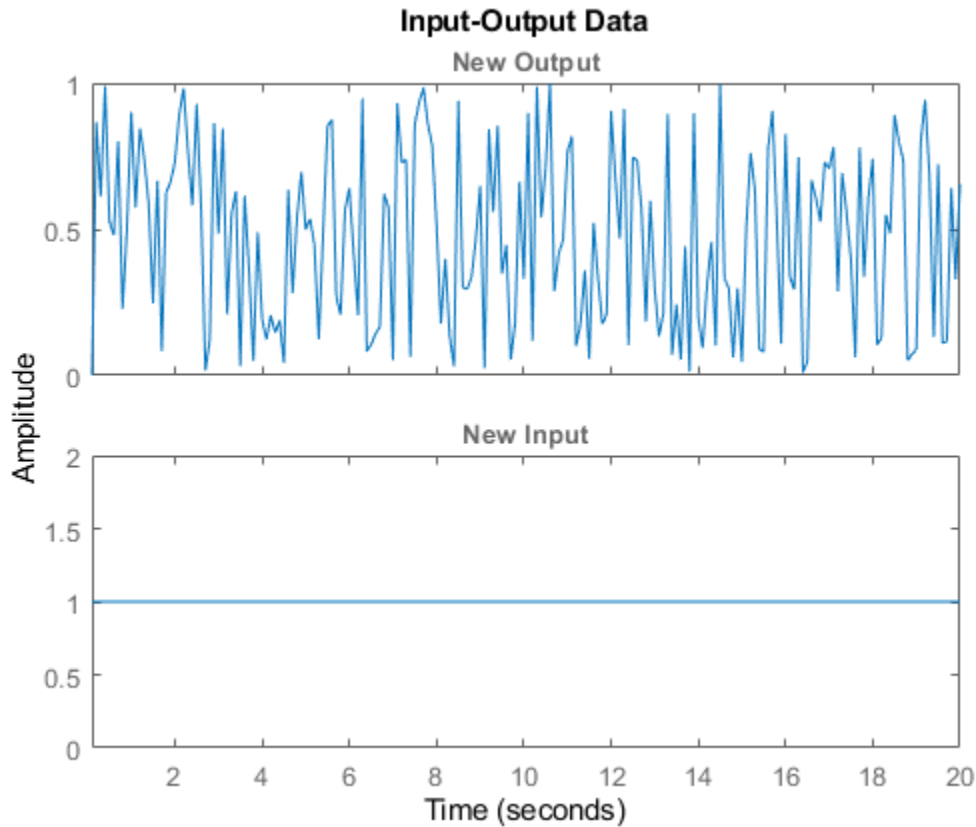
Generating Inputs

The command `idinput` generates typical input signals.

```
u = idinput([30 1 10],'sine'); % 10 periods of 30 samples
u = iddata([],u,1,'Period',30) % Making the input an IDDATA object.


u =

Time domain data set with 300 samples.
Sample time: 1 seconds

Inputs       Unit (if specified)
   u1
```

SIM applied to an iddata input delivers an iddata output. Let us use `sim` to obtain the response of an estimated model m using the input u. We also add noise to the model response in accordance with the noise dynamics of the model. We do this by using the "AddNoise" simulation option:

```
m = idpoly([1 -1.5 0.7],[0 1 0.5]);  % This creates a model; see below.
options = simOptions;
options.AddNoise = true;
y = sim(m,u,options) % simulated response produced as an iddata object


y =
```

```
Time domain data set with 300 samples.
Sample time: 1 seconds
Name: m

Outputs      Unit (if specified)
   y1
```

The simulation input u and the output y may be combined into a single `iddata` object as follows:

```
z5 = [y u] % The output-input iddata.


z5 =

Time domain data set with 300 samples.
Sample time: 1 seconds
Name: m

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

More about the `iddata` object is found under `help iddata`.

The Linear Model Objects

All models are delivered as MATLAB® objects. There are a few different objects depending on the type of model used, but this is mostly transparent.

```
load iddata1
m = armax(z1,[2 2 2 1]);  % This creates an ARMAX model, delivered as an IDPOLY object
```

All relevant properties of this model are packaged as one object (here, `idpoly`). To display it just type its name:

```
m


m =
Discrete-time ARMAX model: A(z)y(t) = B(z)u(t) + C(z)e(t)
  A(z) = 1 - 1.531 z^-1 + 0.7293 z^-2

  B(z) = 0.943 z^-1 + 0.5224 z^-2

  C(z) = 1 - 1.059 z^-1 + 0.1968 z^-2

Sample time: 0.1 seconds

Parameterization:
   Polynomial orders:   na=2   nb=2   nc=2   nk=1
   Number of free coefficients: 6
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
```

```
Estimated using ARMAX on time domain data "z1".
Fit to estimation data: 76.38% (prediction focus)
FPE: 1.127, MSE: 1.082
```

Many of the model properties are directly accessible

```
m.a     % The A-polynomial


ans =

   1.0000   -1.5312    0.7293
```

A list of properties is obtained by get:

```
get(m)
                 A: [1 -1.5312 0.7293]
                 B: [0 0.9430 0.5224]
                 C: [1 -1.0587 0.1968]
                 D: 1
                 F: 1
    IntegrateNoise: 0
          Variable: 'z^-1'
           IODelay: 0
         Structure: [1x1 pmodel.polynomial]
     NoiseVariance: 1.1045
        InputDelay: 0
       OutputDelay: 0
         InputName: {'u1'}
         InputUnit: {''}
        InputGroup: [1x1 struct]
        OutputName: {'y1'}
        OutputUnit: {''}
       OutputGroup: [1x1 struct]
             Notes: [0x1 string]
          UserData: []
              Name: ''
                Ts: 0.1000
          TimeUnit: 'seconds'
      SamplingGrid: [1x1 struct]
            Report: [1x1 idresults.polyest]
```

Use `present` to view the estimated parameter covariance as +/- 1 standard deviation uncertainty values on individual parameters:

```
present(m)


m =
Discrete-time ARMAX model: A(z)y(t) = B(z)u(t) + C(z)e(t)
  A(z) = 1 - 1.531 (+/- 0.01801) z^-1 + 0.7293 (+/- 0.01473) z^-2

  B(z) = 0.943 (+/- 0.06074) z^-1 + 0.5224 (+/- 0.07818) z^-2

  C(z) = 1 - 1.059 (+/- 0.06067) z^-1 + 0.1968 (+/- 0.05957) z^-2
```

```
Sample time: 0.1 seconds

Parameterization:
   Polynomial orders:    na=2    nb=2    nc=2    nk=1
   Number of free coefficients: 6
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Termination condition: Near (local) minimum, (norm(g) < tol)..
Number of iterations: 3, Number of function evaluations: 7

Estimated using ARMAX on time domain data "z1".
Fit to estimation data: 76.38% (prediction focus)
FPE: 1.127, MSE: 1.082
More information in model's "Report" property.
```

Use `getpvec` to fetch a flat list of all the model parameters, or just the free ones, and their uncertainties. Use `getcov` to fetch the entire covariance matrix.

```
[par, dpar] = getpvec(m, 'free')
CovFree = getcov(m,'value')


par =

   -1.5312
    0.7293
    0.9430
    0.5224
   -1.0587
    0.1968


dpar =

    0.0180
    0.0147
    0.0607
    0.0782
    0.0607
    0.0596


CovFree =

    0.0003   -0.0003    0.0000    0.0007    0.0004   -0.0003
   -0.0003    0.0002   -0.0000   -0.0004   -0.0003    0.0002
    0.0000   -0.0000    0.0037   -0.0034   -0.0000    0.0001
    0.0007   -0.0004   -0.0034    0.0061    0.0008   -0.0005
    0.0004   -0.0003   -0.0000    0.0008    0.0037   -0.0032
   -0.0003    0.0002    0.0001   -0.0005   -0.0032    0.0035
```

nf = 0, nd = 0 denote orders of a general linear model, of which the ARMAX model is a special case.

Report contains information about the estimation process:

```
m.Report
m.Report.DataUsed          % record of data used for estimation
m.Report.Fit               % quantitative measures of model quality
m.Report.Termination       % search termination conditions


ans =

              Status: 'Estimated using ARMAX with prediction focus'
              Method: 'ARMAX'
    InitialCondition: 'zero'
                 Fit: [1x1 struct]
          Parameters: [1x1 struct]
         OptionsUsed: [1x1 idoptions.polyest]
           RandState: [1x1 struct]
            DataUsed: [1x1 struct]
         Termination: [1x1 struct]


ans =

  struct with fields:

           Name: 'z1'
           Type: 'Time domain data'
         Length: 300
             Ts: 0.1000
    InterSample: 'zoh'
    InputOffset: []
   OutputOffset: []


ans =

  struct with fields:

    FitPercent: 76.3807
       LossFcn: 1.0824
           MSE: 1.0824
           FPE: 1.1266
           AIC: 887.1256
          AICc: 887.4123
          nAIC: 0.1192
           BIC: 909.3483


ans =

  struct with fields:

                 WhyStop: 'Near (local) minimum, (norm(g) < tol).'
              Iterations: 3
     FirstOrderOptimality: 7.2436
                FcnCount: 7
               UpdateNorm: 0.0067
         LastImprovement: 0.0067
```

To obtain on-line information about the minimization, use the 'Display' estimation option with possible values 'off', 'on', and 'full'. This launches a progress viewer that shows information on the model estimation progress.

```
Opt = armaxOptions('Display','on');
m1 = armax(z1,[2 2 2 1],Opt);
```

ARMAX Model Identification

Estimation data: Time domain data z1
Data has 1 outputs, 1 inputs and 300 samples.
Orders: na = 2, nb = 2, nc = 2, nk = 1

ESTIMATION PROGRESS

Initializing model parameters...
Initializing using 'arx' method...
Initialization complete.

Algorithm: Nonlinear least squares with automatically chosen line search method
----

| | | Norm of | First-order | Improvement (%) | | |
Iteration	Cost	step	optimality	Expected	Achieved	Bisections
0	1.10868	–	224	2.13	–	–
1	1.08367	0.119	161	2.13	2.26	0
2	1.0825	0.015	4.84	0.0994	0.108	0
3	1.08243	0.00373	7.24	0.00528	0.00673	0

RESULT

Termination condition: Near (local) minimum, (norm(g) < tol)..
Number of iterations: 3, Number of function evaluations: 7

Status: Estimated using ARMAX with prediction focus
Fit to estimation data: 76.38%, FPE: 1.12661

■ Stop   Close

Variants of Linear Models - IDTF, IDPOLY, IDPROC, IDSS and IDGREY

There are several types of linear models. The one above is an example of the `idpoly` version for polynomial type models. Different variants of polynomial-type models, such as Box-Jenkins models,

Output Error models, ARMAX models etc are obtained using the corresponding estimators - `bj, oe, armax, arx` etc. All of these are presented as `idpoly` objects.

Other variants are `idss` for state-space models; `idgrey` for user-defined structured state-space models; `idtf` for transfer function models, and `idproc` for process models (gain+delay+static gain).

The commands to evaluate the model: `bode, step, iopzmap, compare`, etc, all operate directly on the model objects, for example:

```
compare(z1,m1)
```
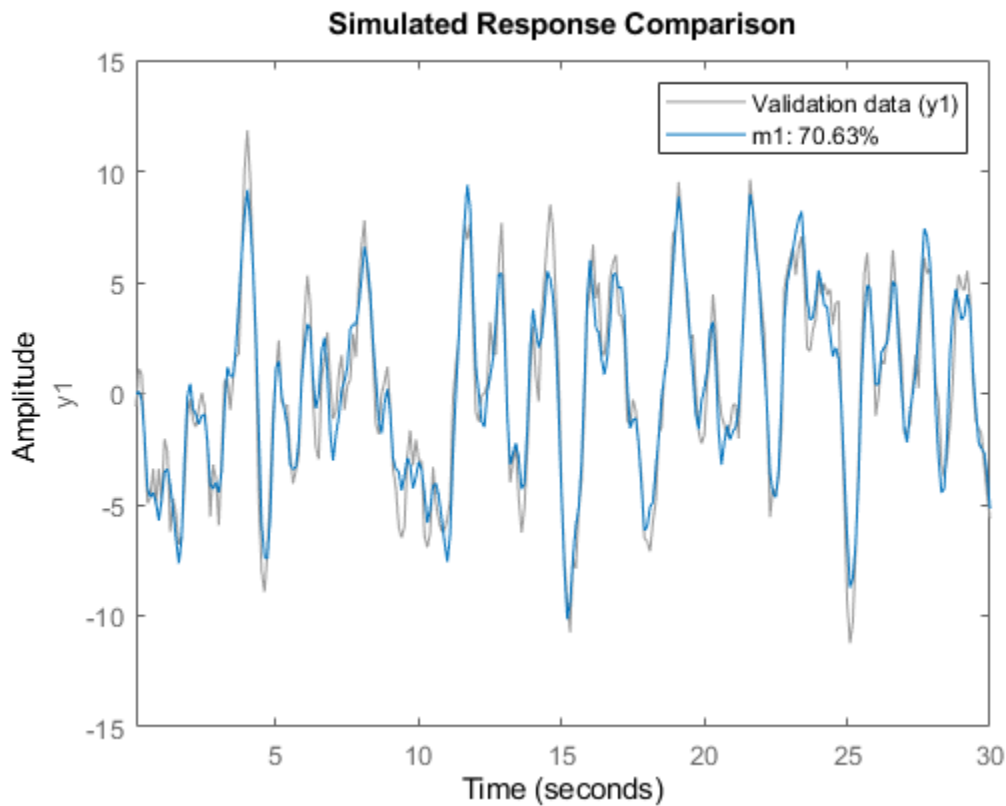

Transformations to state-space, transfer function and zeros/poles are obtained by `idssdata`, `tfdata` and `zpkdata`:

```
[num,den]  = tfdata(m1,'v')


num =

        0    0.9430    0.5224


den =

   1.0000   -1.5312    0.7293
```

The 'v' means that num and den are returned as vectors and not as cell arrays. The cell arrays are useful to handle multivariable systems. To also retrieve the 1 standard deviation uncertainties in the values of num and den use:

```
[num, den, ~, dnum, dden] = tfdata(m1,'v')


num =

        0     0.9430     0.5224


den =

    1.0000    -1.5312     0.7293


dnum =

        0     0.0607     0.0782


dden =

        0     0.0180     0.0147
```

Transforming Identified Models into Numeric LTIs of Control System Toolbox

The objects also connect directly to the Control System Toolbox™ model objects, like tf, ss, and zpk, and can be converted into these LTI objects, if Control System Toolbox is available. For example, tf converts an idpoly object into a tf object.

```
CSTBInstalled = exist('tf','class')==8;
if CSTBInstalled % check if Control System Toolbox is installed
    tfm = tf(m1) % convert IDPOLY model m1 into a TF object
end


tfm =

  From input "u1" to output "y1":
    0.943 z^-1 + 0.5224 z^-2
    ---------------------------
  1 - 1.531 z^-1 + 0.7293 z^-2

Sample time: 0.1 seconds
Discrete-time transfer function.
```

When converting an IDLTI model to the Control Systems Toolbox's LTI models, noise component is not retained. To also include the noise channels as regular inputs of the LTI model, use the 'augmented' flag:

```
if CSTBInstalled
    tfm2 = tf(m1,'augmented')
end
```

```
tfm2 =

  From input "u1" to output "y1":
    0.943 z^-1 + 0.5224 z^-2
  ----------------------------
  1 - 1.531 z^-1 + 0.7293 z^-2

  From input "v@y1" to output "y1":
  1.051 - 1.113 z^-1 + 0.2069 z^-2
  --------------------------------
    1 - 1.531 z^-1 + 0.7293 z^-2

Input groups:
      Name       Channels
    Measured        1
     Noise          2

Sample time: 0.1 seconds
Discrete-time transfer function.
```

The noise channel is named `v@y1` in the model `tfm2`.

Comparison of Various Model Identification Methods

This example shows several identification methods available in System Identification Toolbox™. We begin by simulating experimental data and use several estimation techniques to estimate models from the data. The following estimation routines are illustrated in this example: `spa`, `ssest`, `tfest`, `arx`, `oe`, `armax` and `bj`.

System Description

A noise corrupted linear system can be described by:

```
y = G u + H e
```

where `y` is the output and `u` is the input, while `e` denotes the unmeasured (white) noise source. `G` is the system's transfer function and `H` gives the description of the noise disturbance.

There are many ways to estimate `G` and `H`. Essentially they correspond to different ways of parameterizing these functions.

Defining a Model

System Identification Toolbox provides users with the option of simulating data as would have been obtained from a physical process. Let us simulate the data from an IDPOLY model with a given set of coefficients.

```
B = [0 1 0.5];
A = [1 -1.5 0.7];
m0 = idpoly(A,B,[1 -1 0.2],'Ts',0.25,'Variable','q^-1'); % The sample time is 0.25 s.
```

The third argument `[1 -1 0.2]` gives a characterization of the disturbances that act on the system. Execution of these commands produces the following discrete-time idpoly model:

```
m0
```

```
m0 =
Discrete-time ARMAX model: A(q)y(t) = B(q)u(t) + C(q)e(t)
  A(q) = 1 - 1.5 q^-1 + 0.7 q^-2

  B(q) = q^-1 + 0.5 q^-2

  C(q) = 1 - q^-1 + 0.2 q^-2

Sample time: 0.25 seconds

Parameterization:
   Polynomial orders:   na=2   nb=2   nc=2   nk=1
   Number of free coefficients: 6
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

Here `q` denotes the shift operator so that A(q)y(t) is really short for y(t) - 1.5 y(t-1) + 0.7 y(t-2). The display of model `m0` shows that it is an ARMAX model.

This particular model structure is known as an ARMAX model, where AR (Autoregressive) refers to the A-polynomial, MA (Moving average) to the noise C-polynomial and X to the "eXtra" input B(q)u(t).

In terms of the general transfer functions G and H, this model corresponds to a parameterization:

$G(q) = B(q)/A(q)$ and $H(q) = C(q)/A(q)$, with common denominators

Simulating the Model

We generate an input signal u and simulate the response of the model to these inputs. The `idinput` command can be used to create an input signal to the model and the `iddata` command will package the signal into a suitable format. The `sim` command can then be used to simulate the output as shown below:

```
prevRng = rng(12,'v5normal');
u = idinput(350,'rbs');  %Generates a random binary signal of length 350
u = iddata([],u,0.25);   %Creates an IDDATA object. The sample time is 0.25 sec.
y = sim(m0,u,'noise')    %Simulates the model's response for this

y =

Time domain data set with 350 samples.
Sample time: 0.25 seconds
Name: m0

Outputs      Unit (if specified)
   y1


                         %input with added white Gaussian noise e
                         %according to the model description m0
rng(prevRng);
```

One aspect to note is that the signals u and y are both IDDATA objects. Next we collect this input-output data to form a single iddata object.

```
z = [y,u];
```

To get information on the data object (which now incorporates both the input and output data samples), just enter its name at the MATLAB® command window:

```
z

z =

Time domain data set with 350 samples.
Sample time: 0.25 seconds
Name: m0

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

To plot the first 100 values of the input u and output y, use the `plot` command on the iddata object:

```
h = gcf; set(h,'DefaultLegendLocation','best')
h.Position = [100 100 780 520];
plot(z(1:100));
```

Input-Output Data
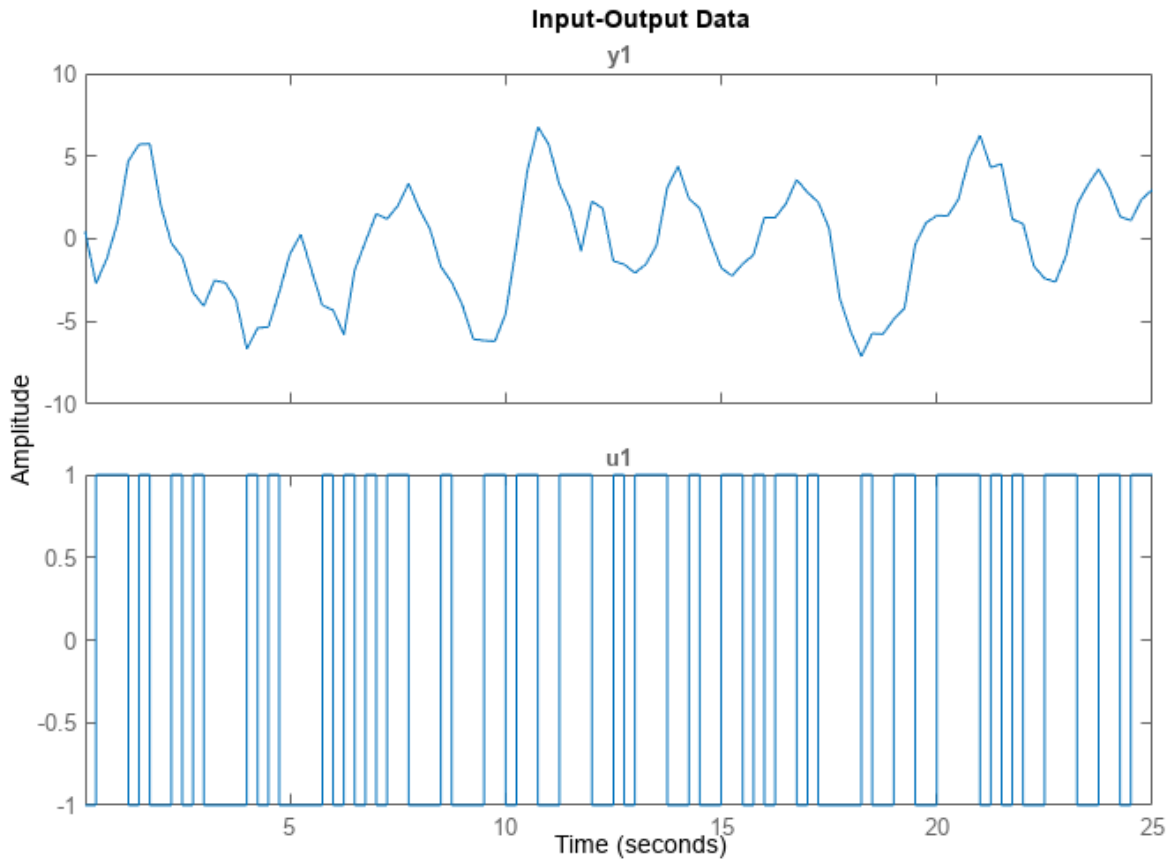
It is good practice to use only a portion of the data for estimation purposes, `ze` and save another part to validate the estimated models later on:

```
ze = z(1:200);
zv = z(201:350);
```

Performing Spectral Analysis

Now that we have obtained the simulated data, we can estimate models and make comparisons. Let us start with spectral analysis. We invoke the `spa` command which returns a spectral analysis estimate of the frequency function and the noise spectrum.

```
GS = spa(ze);
```

The result of spectral analysis is a complex-valued function of frequency: the frequency response. It is packaged into an object called `IDFRD` object (Identified Frequency Response Data):

```
GS
```

GS =

```
IDFRD model.
Contains Frequency Response Data for 1 output(s) and 1 input(s), and the spectra for disturbances
Response data and disturbance spectra are available at 128 frequency points, ranging from 0.0981

Sample time: 0.25 seconds
Output channels: 'y1'
Input channels: 'u1'
Status:
Estimated using SPA on time domain data "m0".
```

To plot the frequency response it is customary to use the Bode plot, with the `bode` or `bodeplot` command:

```
clf
h = bodeplot(GS); % bodeplot returns a plot handle, which bode does not
ax = axis; axis([0.1 10 ax(3:4)])
```
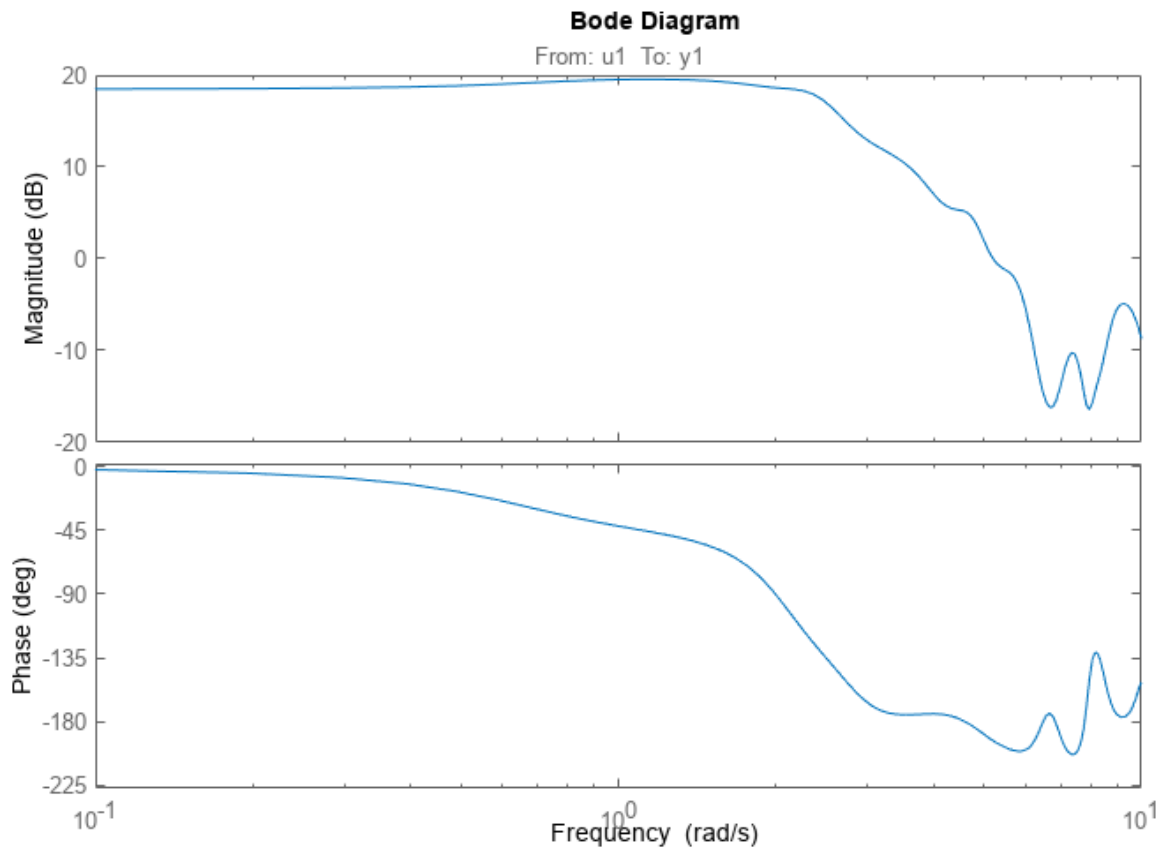

The estimate GS is uncertain, since it is formed from noise corrupted data. The spectral analysis method provides also its own assessment of this uncertainty. To display the uncertainty (say for example 3 standard deviations) we can use the `showConfidence` command on the plot handle h returned by the previous `bodeplot` command.

```
showConfidence(h,3)
```

4-129

Bode Diagram
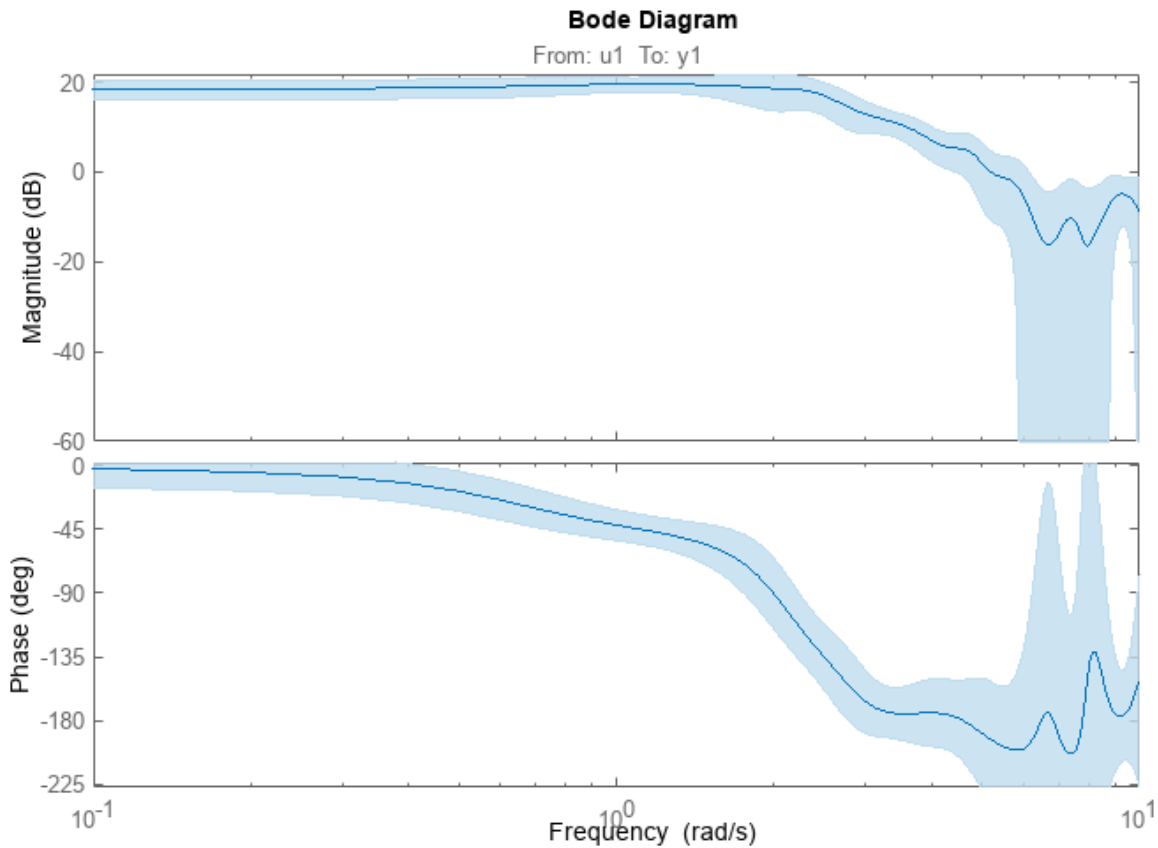
The plot says that although the nominal estimate of the frequency response (blue line) is not necessarily accurate, there is a 99.7% probability (3 standard deviations of the normal distribution) that the true response is within the shaded (light-blue) region.

Estimating Parametric State Space Models

Next let us estimate a default (without us specifying a particular model structure) linear model. It will be computed as a state-space model using a prediction error method. We use the `ssest` function in this case:

```
m = ssest(ze) % The order of the model will be chosen automatically

m =
  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   -0.007167      1.743
    x2      -2.181     -1.337

  B =
              u1
    x1   0.09388
```

```
    x2   0.2408

  C =
            x1      x2
    y1   47.34  -14.4

  D =
        u1
    y1   0

  K =
              y1
    x1   0.04108
    x2  -0.03751

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: estimate
   Number of free coefficients: 10
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on time domain data "m0".
Fit to estimation data: 75.08% (prediction focus)
FPE: 0.9835, MSE: 0.9262
```

At this point the state-space model matrices do not tell us very much. We could compare the frequency response of the model m with that of the spectral analysis estimate GS. Note that GS is a discrete-time model while m is a continuous-time model. It is possible to use ssest to compute discrete time models too.

```
bodeplot(m,GS)
ax = axis; axis([0.1 10 ax(3:4)])
```

Bode Diagram
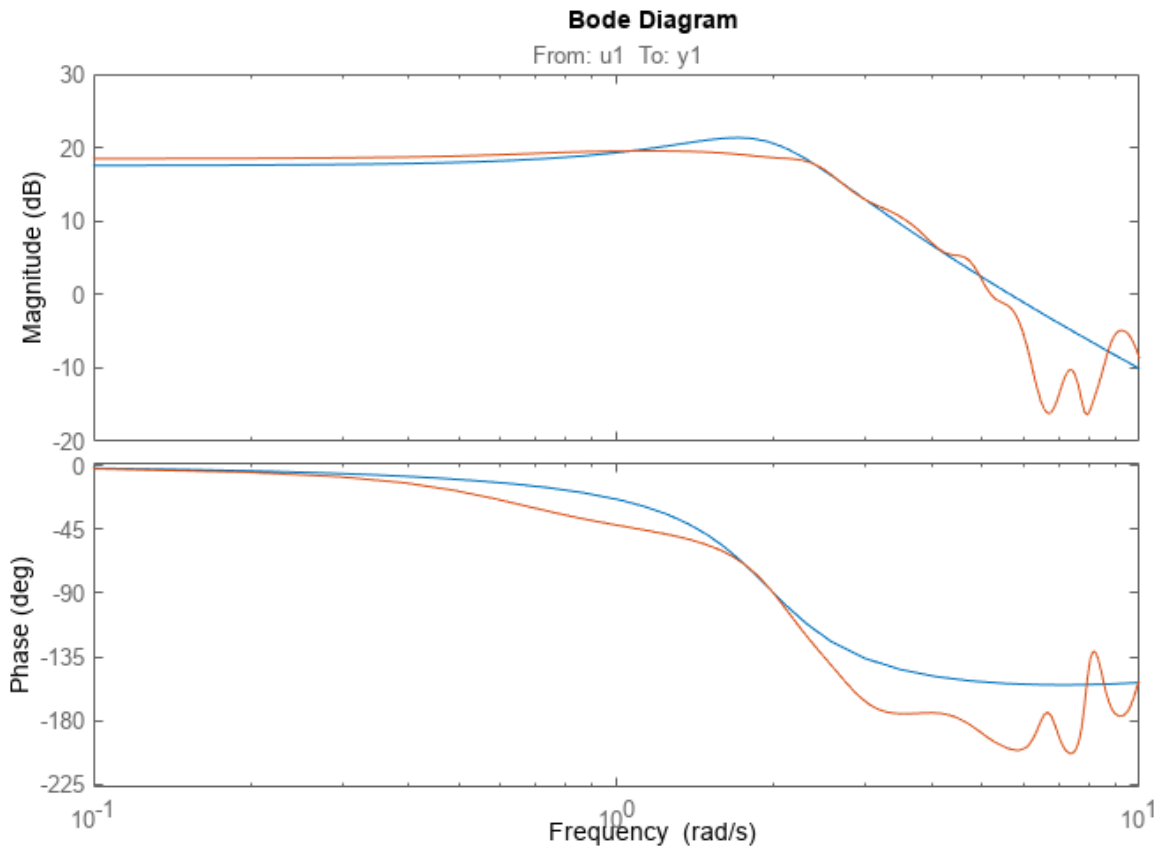
We note that the two frequency responses are close, despite the fact that they have been estimated in very different ways.

Estimating Simple Transfer Functions

There is a wide variety of linear model structures, all corresponding to linear difference or differential equations describing the relation between u and y. The different structures all correspond to various ways of modeling the noise influence. The simplest of these models are the transfer function and ARX models. A transfer function takes the form:

```
Y(s) = NUM(s)/DEN(s) U(s) + E(s)
```

where NUM(s) and DEN(s) are the numerator and denominator polynomials, and Y(s), U(s) and E(s) are the Laplace Transforms of the output, input and error signals (y(t), u(t) and e(t)) respectively. NUM and DEN can be parameterized by their orders which represent the number of zeros and poles. For a given number of poles and zeros, we can estimate a transfer function using the `tfest` command:

```
mtf = tfest(ze, 2, 2) % transfer function with 2 zeros and 2 poles

mtf =
  From input "u1" to output "y1":
  -0.05428 s^2 - 0.02386 s + 29.6
  -------------------------------
```

```
        s^2 + 1.361 s + 4.102

Continuous-time identified transfer function.

Parameterization:
   Number of poles: 2    Number of zeros: 2
   Number of free coefficients: 5
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using TFEST on time domain data "m0".
Fit to estimation data: 71.69%
FPE: 1.282, MSE: 1.195
```

AN ARX model is represented as: A(q)y(t) = B(q)u(t) + e(t),

or in long-hand,

$$y(t) + a_1 y(t-1) + \cdots + a_{na}y(t-na) \ = \ b_1 u(t-nk) + \ \cdots b_{nb}u(t-nb-nk+1) + e(t)$$

This model is linear in coefficients. The coefficients a_1, a_2, ..., b_1, b_2, .. can be computed efficiently using a least squares estimation technique. An estimate of an ARX model with a prescribed structure - two poles, one zero and a single lag on the input is obtained as ([na nb nk] = [2 2 1]):

```
mx = arx(ze,[2 2 1])
```

```
mx =
Discrete-time ARX model: A(z)y(t) = B(z)u(t) + e(t)
  A(z) = 1 - 1.32 z^-1 + 0.5393 z^-2

  B(z) = 0.9817 z^-1 + 0.4049 z^-2

Sample time: 0.25 seconds

Parameterization:
   Polynomial orders:   na=2   nb=2   nk=1
   Number of free coefficients: 4
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using ARX on time domain data "m0".
Fit to estimation data: 68.18% (prediction focus)
FPE: 1.603, MSE: 1.509
```

Comparing the Performance of Estimated Models

Are the models (mtf, mx) better or worse than the default state-space model m? One way to find out is to simulate each model (noise free) with the input from the validation data zv and compare the corresponding simulated outputs with the measured output in the set zv:

```
compare(zv,m,mtf,mx)
```

Simulated Response Comparison
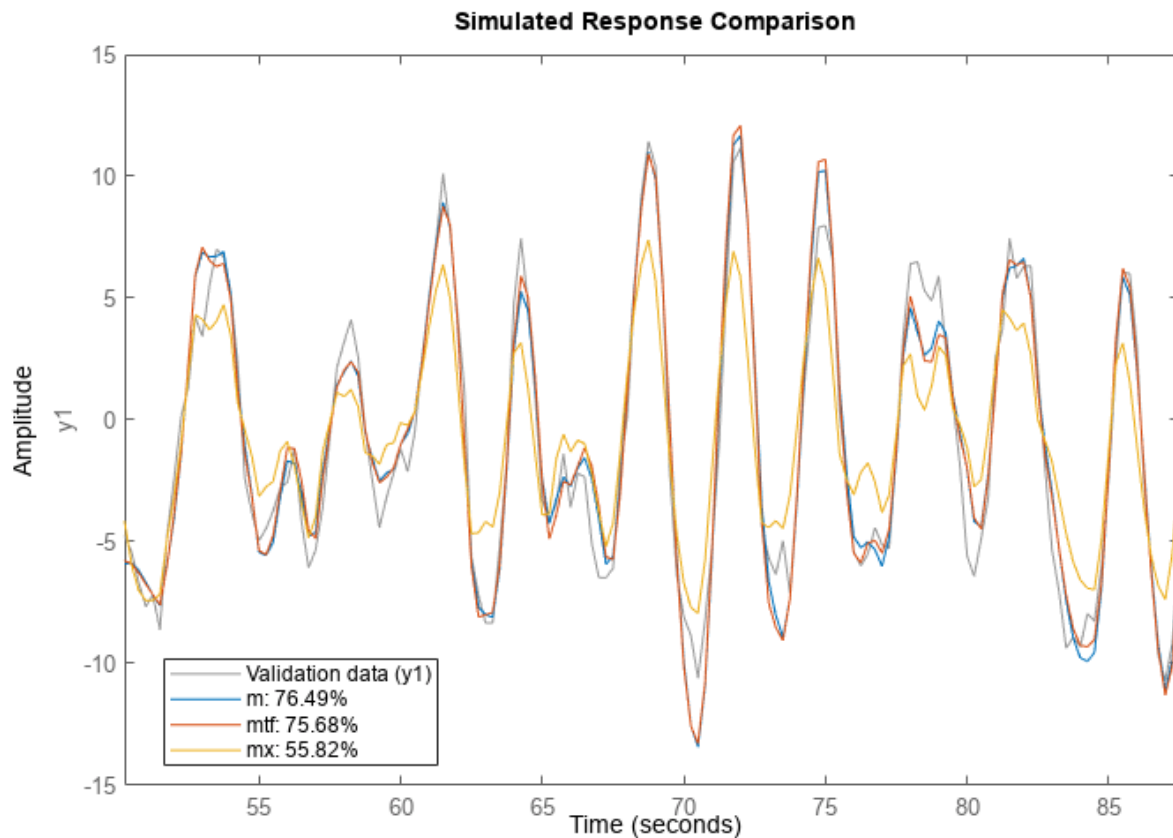
The fit is the percent of the output variation that is explained by the model. Clearly `m` is the better model, although `mtf` comes close. A discrete-time transfer function can be estimate using either the `tfest` or the `oe` command. The former creates an IDTF model while the latter creates an IDPOLY model of Output-Error structure (y(t) = B(q)/F(q)u(t) + e(t)), but they are mathematically equivalent.

```
md1 = tfest(ze,2,2,'Ts',0.25)   % two poles and 2 zeros (counted as roots of polynomials in z^-1)

md1 =
  From input "u1" to output "y1":
   0.8383 z^-1 + 0.7199 z^-2
  ---------------------------
  1 - 1.497 z^-1 + 0.7099 z^-2

Sample time: 0.25 seconds
Discrete-time identified transfer function.

Parameterization:
   Number of poles: 2    Number of zeros: 2
   Number of free coefficients: 4
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using TFEST on time domain data "m0".
Fit to estimation data: 71.46%
```

```
FPE: 1.264, MSE: 1.214
```

```
md2 = oe(ze,[2 2 1])
```

```
md2 =
Discrete-time OE model: y(t) = [B(z)/F(z)]u(t) + e(t)
  B(z) = 0.8383 z^-1 + 0.7199 z^-2

  F(z) = 1 - 1.497 z^-1 + 0.7099 z^-2

Sample time: 0.25 seconds

Parameterization:
   Polynomial orders:   nb=2   nf=2   nk=1
   Number of free coefficients: 4
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using OE on time domain data "m0".
Fit to estimation data: 71.46%
FPE: 1.264, MSE: 1.214
```

```
compare(zv, md1, md2)
```
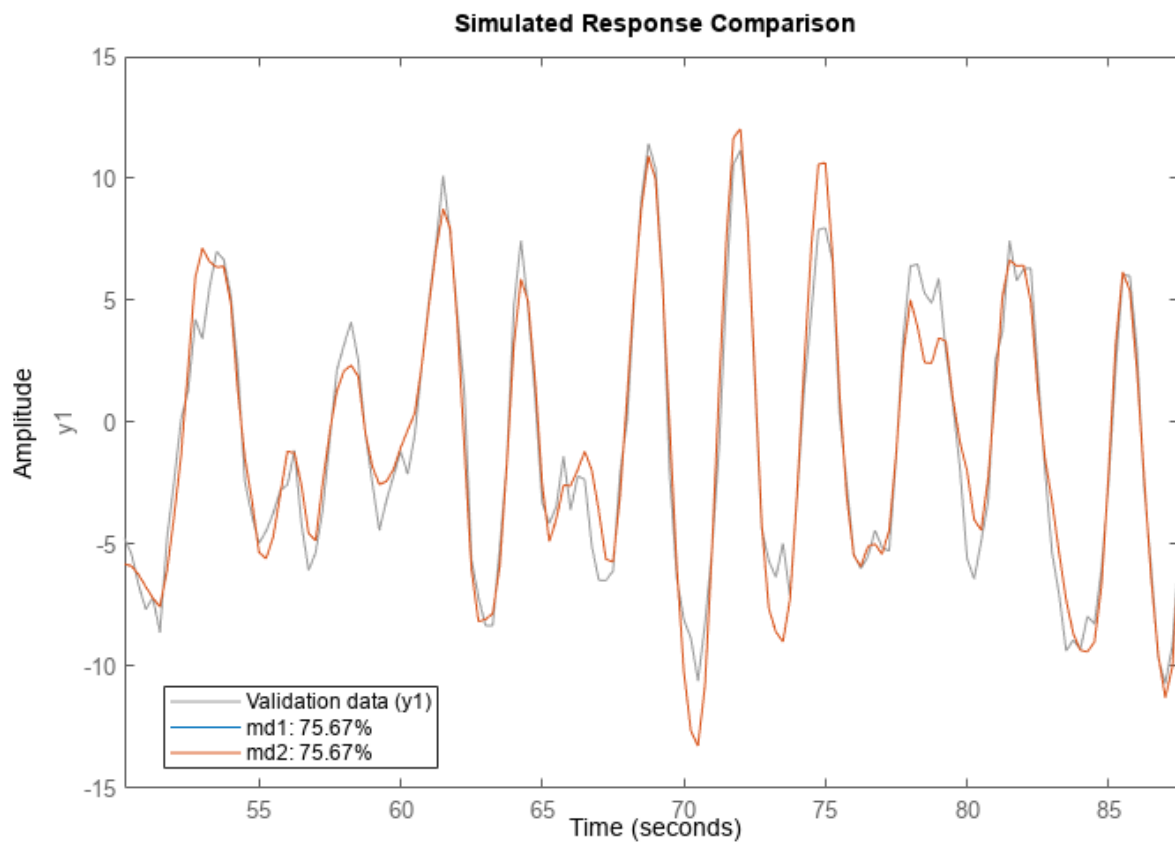
The models `md1` and `md2` deliver identical fits to the data.

Residual Analysis

A further way to gain insight into the quality of a model is to compute the "residuals": i.e. that part `e` in `y = Gu + He` that could not be explained by the model. Ideally, these should be uncorrelated with the input and also mutually uncorrelated. The residuals are computed and their correlation properties are displayed by the `resid` command. This function can be used to evaluate the residues both in the time and frequency domains. First let us obtain the residuals for the Output-Error model in the time domain:

```
resid(zv,md2) % plots the result of residual analysis
```
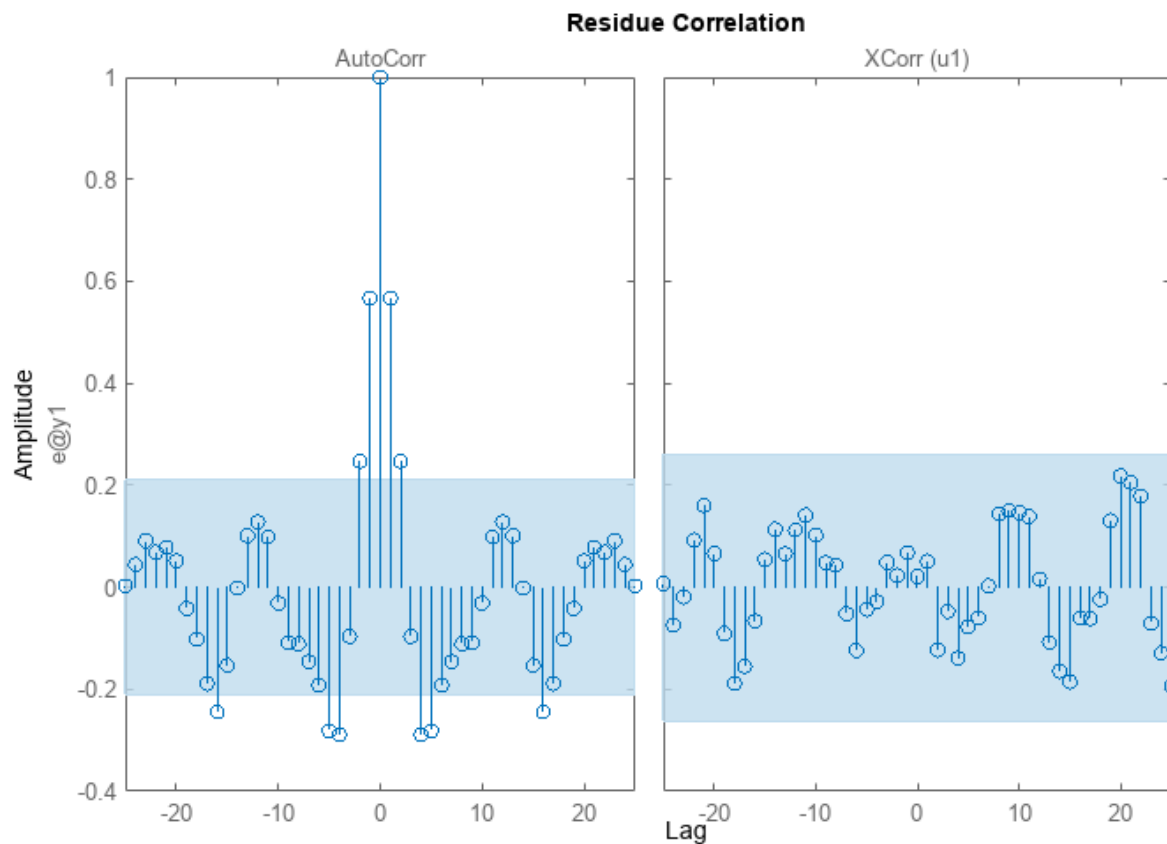

We see that the cross correlation between residuals and input lies in the confidence region, indicating that there is no significant correlation. The estimate of the dynamics `G` should then be considered as adequate. However, the (auto) correlation of `e` is significant, so `e` cannot be seen as white noise. This means that the noise model `H` is not adequate.

Estimating ARMAX and Box-Jenkins Models

Let us now compute a second order ARMAX model and a second order Box-Jenkins model. The ARMAX model has the same noise characteristics as the simulated model `m0` and the Box-Jenkins model allows a more general noise description.

```
am2 = armax(ze,[2 2 2 1])  % 2nd order ARMAX model
```

```
am2 =
Discrete-time ARMAX model: A(z)y(t) = B(z)u(t) + C(z)e(t)
  A(z) = 1 - 1.516 z^-1 + 0.7145 z^-2

  B(z) = 0.982 z^-1 + 0.5091 z^-2

  C(z) = 1 - 0.9762 z^-1 + 0.218 z^-2

Sample time: 0.25 seconds

Parameterization:
   Polynomial orders:   na=2   nb=2   nc=2   nk=1
   Number of free coefficients: 6
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using ARMAX on time domain data "m0".
Fit to estimation data: 75.08% (prediction focus)
FPE: 0.9837, MSE: 0.9264
```

```
bj2 = bj(ze,[2 2 2 2 1])    % 2nd order BOX-JENKINS model

bj2 =
Discrete-time BJ model: y(t) = [B(z)/F(z)]u(t) + [C(z)/D(z)]e(t)
  B(z) = 0.9922 z^-1 + 0.4701 z^-2

  C(z) = 1 - 0.6283 z^-1 - 0.1221 z^-2

  D(z) = 1 - 1.221 z^-1 + 0.3798 z^-2

  F(z) = 1 - 1.522 z^-1 + 0.7243 z^-2

Sample time: 0.25 seconds

Parameterization:
   Polynomial orders:   nb=2   nc=2   nd=2   nf=2   nk=1
   Number of free coefficients: 8
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using BJ on time domain data "m0".
Fit to estimation data: 75.47% (prediction focus)
FPE: 0.9722, MSE: 0.8974
```

Comparing Estimated Models - Simulation and Prediction Behavior

Now that we have estimated so many different models, let us make some model comparisons. This can be done by comparing the simulated outputs as before:

```
clf
compare(zv,am2,md2,bj2,mx,mtf,m)
```

4-137

Simulated Response Comparison
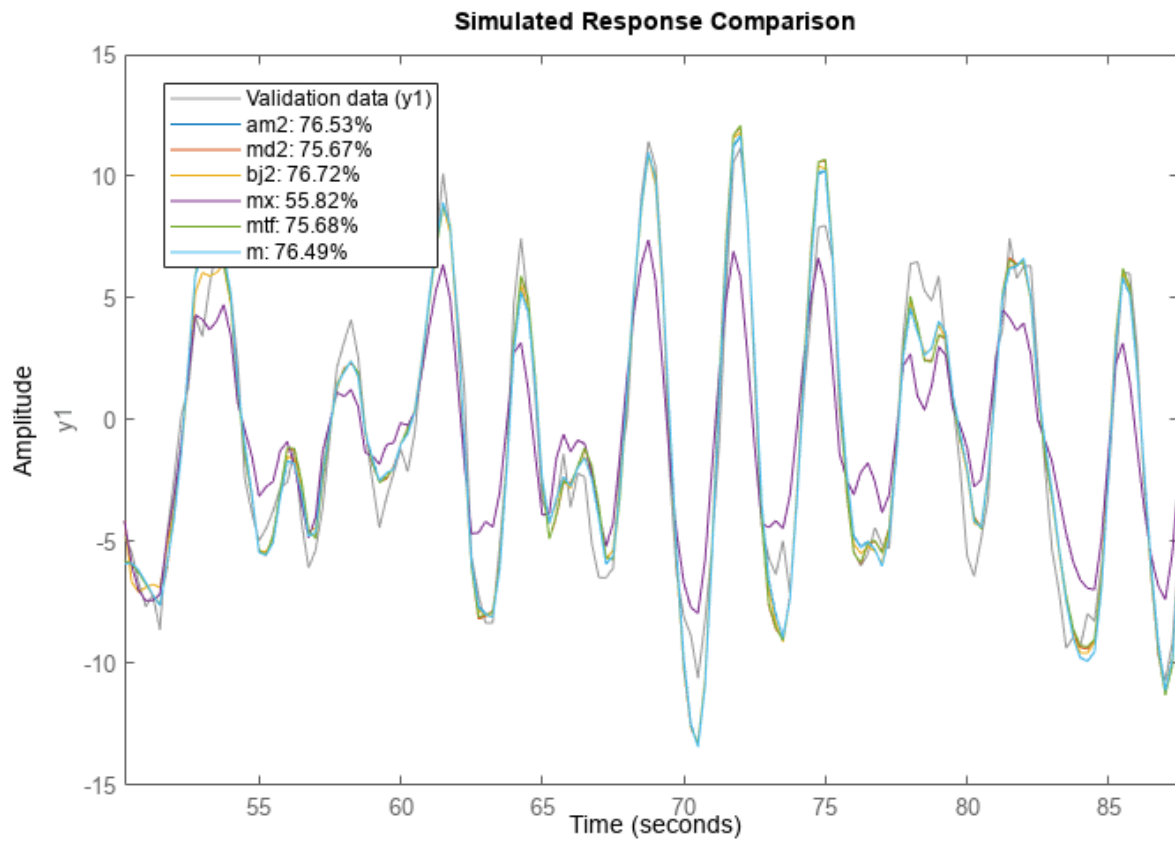
We can also compare how well the various estimated models are able to predict the output, say, 1 step ahead:

```
compare(zv,am2,md2,bj2,mx,mtf,m,1)
```
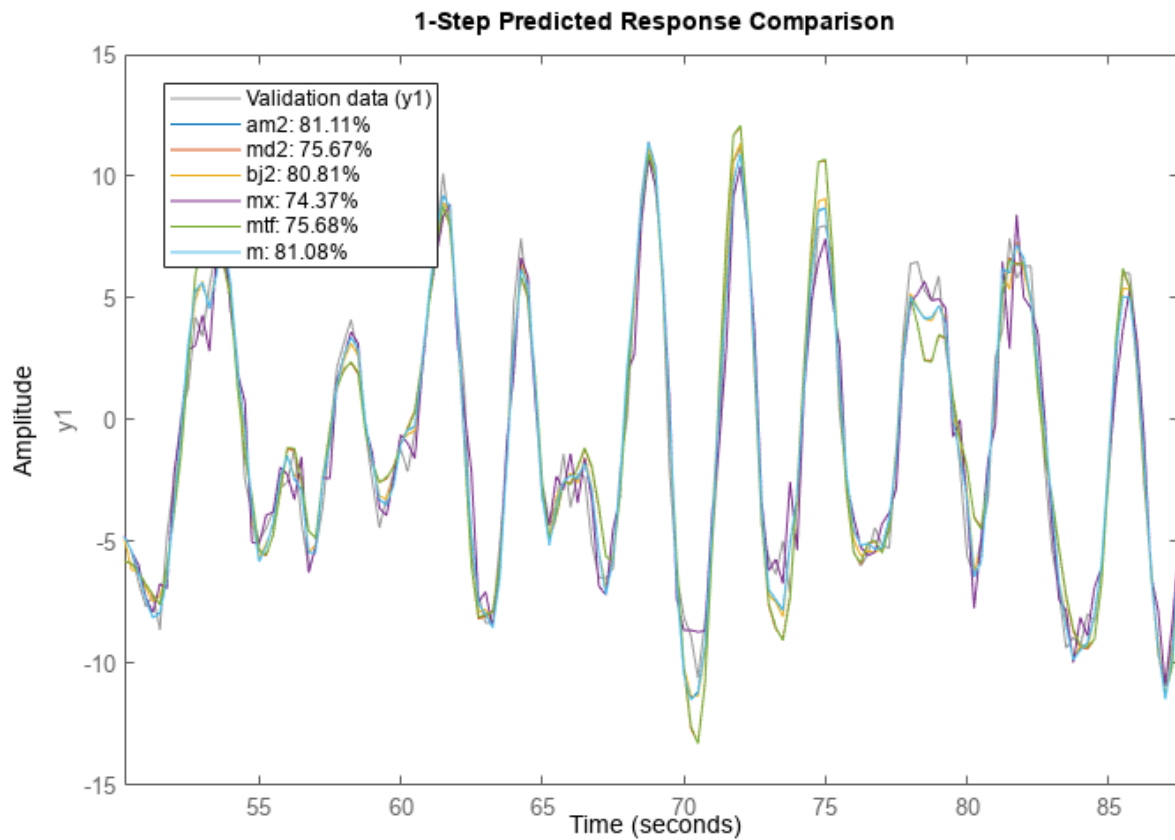
The residual analysis of model `bj2` shows that the prediction error is devoid of any information - it is not auto-correlated and also uncorrelated with the input. This shows that the extra dynamic elements of a Box-Jenkins model (the C and D polynomials) were able to capture the noise dynamics.

```
resid(zv,bj2)
```

Residue Correlation
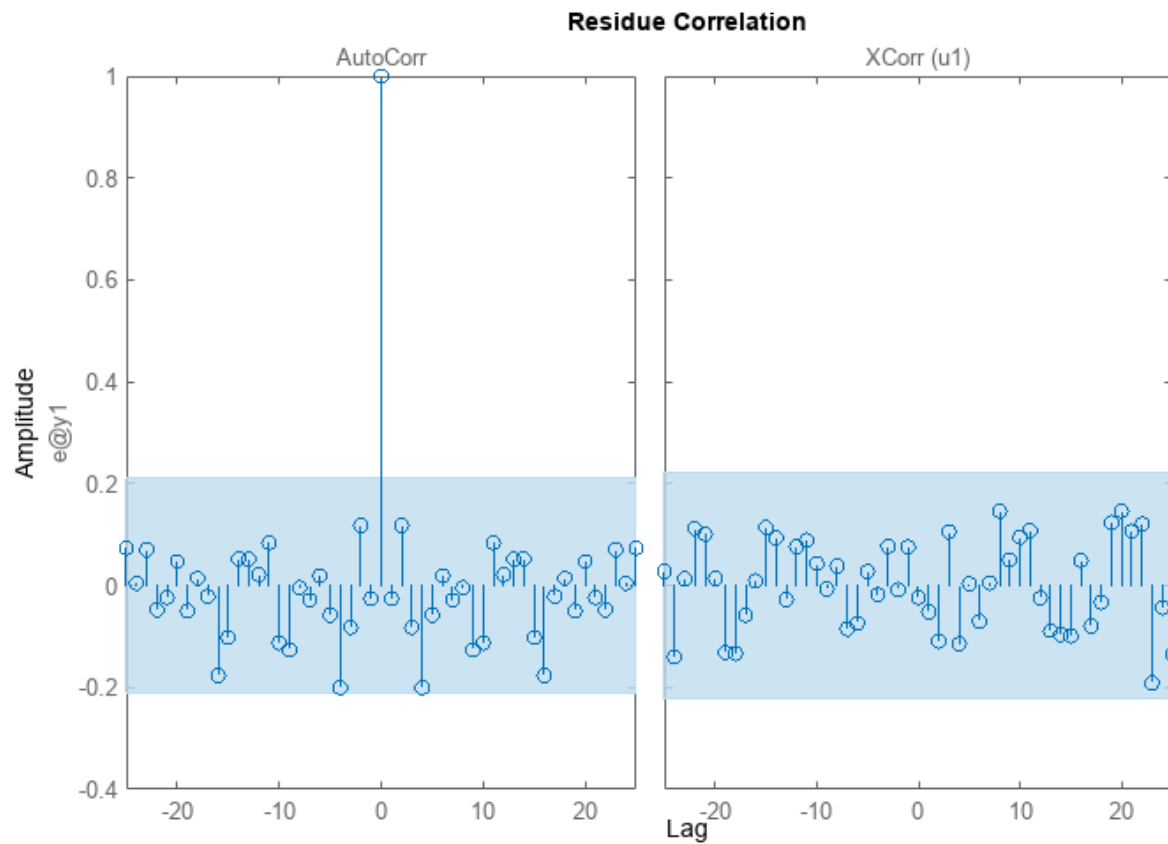
Comparing Frequency Functions

In order to compare the frequency functions for the generated models we use again the `bodeplot` command:

```
clf
opt = bodeoptions; opt.PhaseMatching = 'on';
w = linspace(0.1,4*pi,100);
bodeplot(GS,m,mx,mtf,md2,am2,bj2,w,opt);
legend({'SPA','SS','ARX','TF','OE','ARMAX','BJ'},'Location','West');
```
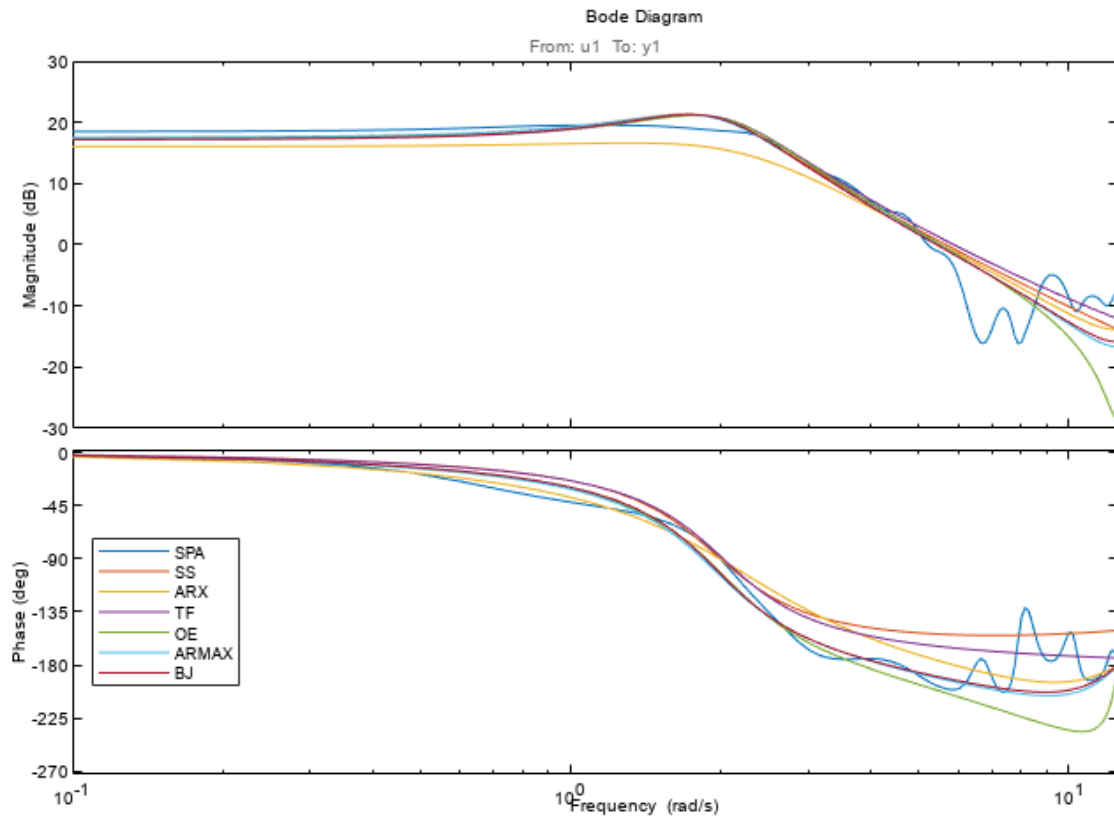
The noise spectra of the estimated models can also be analyzed. For example here we compare the noise spectra the ARMAX and the Box-Jenkins models with the state-space and the Spectral Analysis models. For this, we use the `spectrum` command:

```
spectrum(GS,m,mx,am2,bj2,w)
legend('SPA','SS','ARX','ARMAX','BJ');
```

Power Spectrum
From: e@y1 To: y1
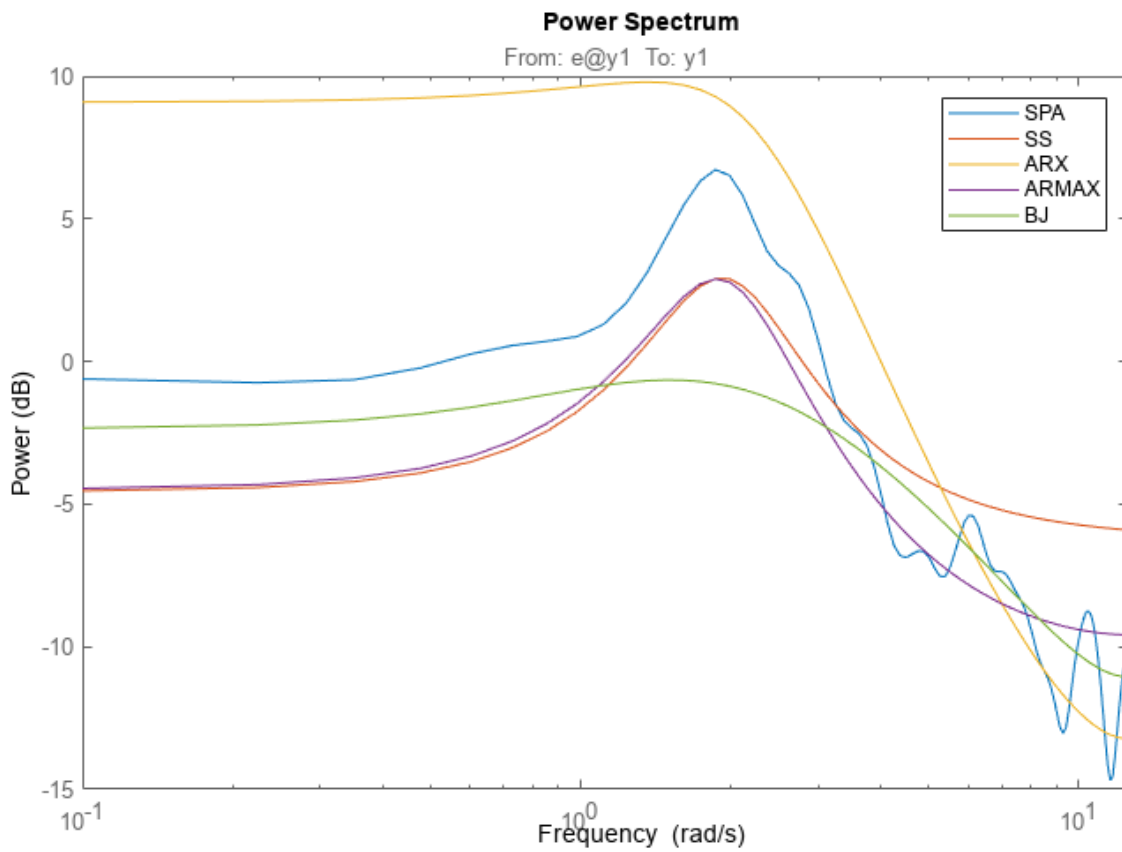
Comparing Estimated Models with the True System

Here we validate the estimated models against the true system `m0` that we used to simulate the input and output data. Let us compare the frequency functions of the ARMAX model with the true system.

```
bode(m0,am2,w)
```
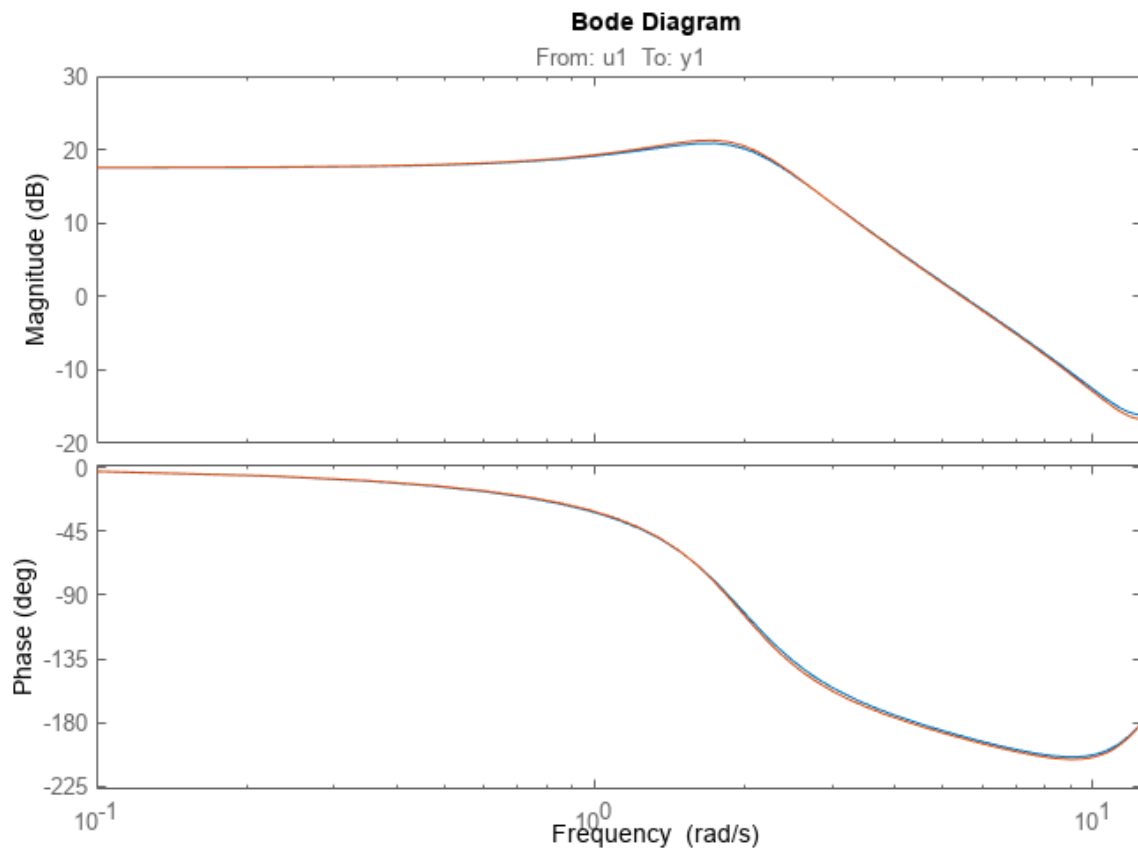
The responses practically coincide. The noise spectra can also be compared.

```
spectrum(m0,am2,w)
```

Power Spectrum
From: e@y1 To: y1
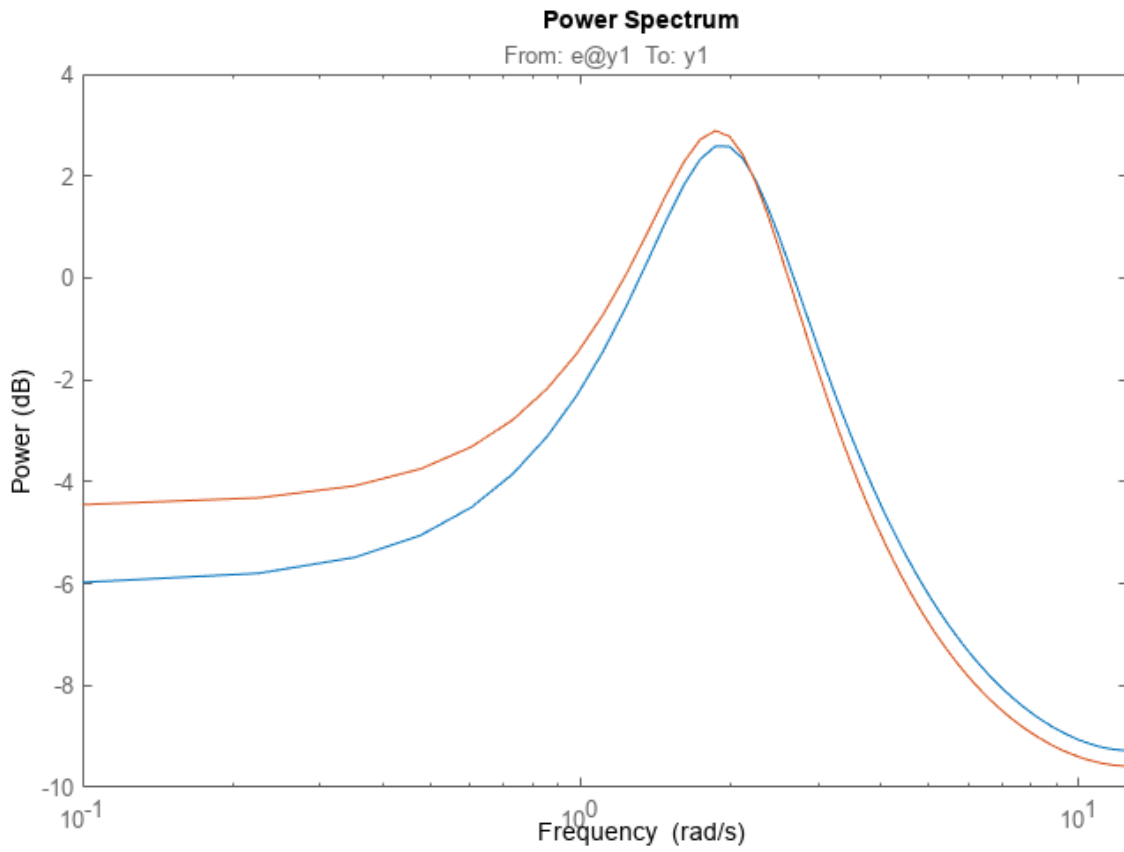
Let us also examine the pole-zero plot:

```
h = iopzplot(m0,am2);
```

Pole-Zero Map
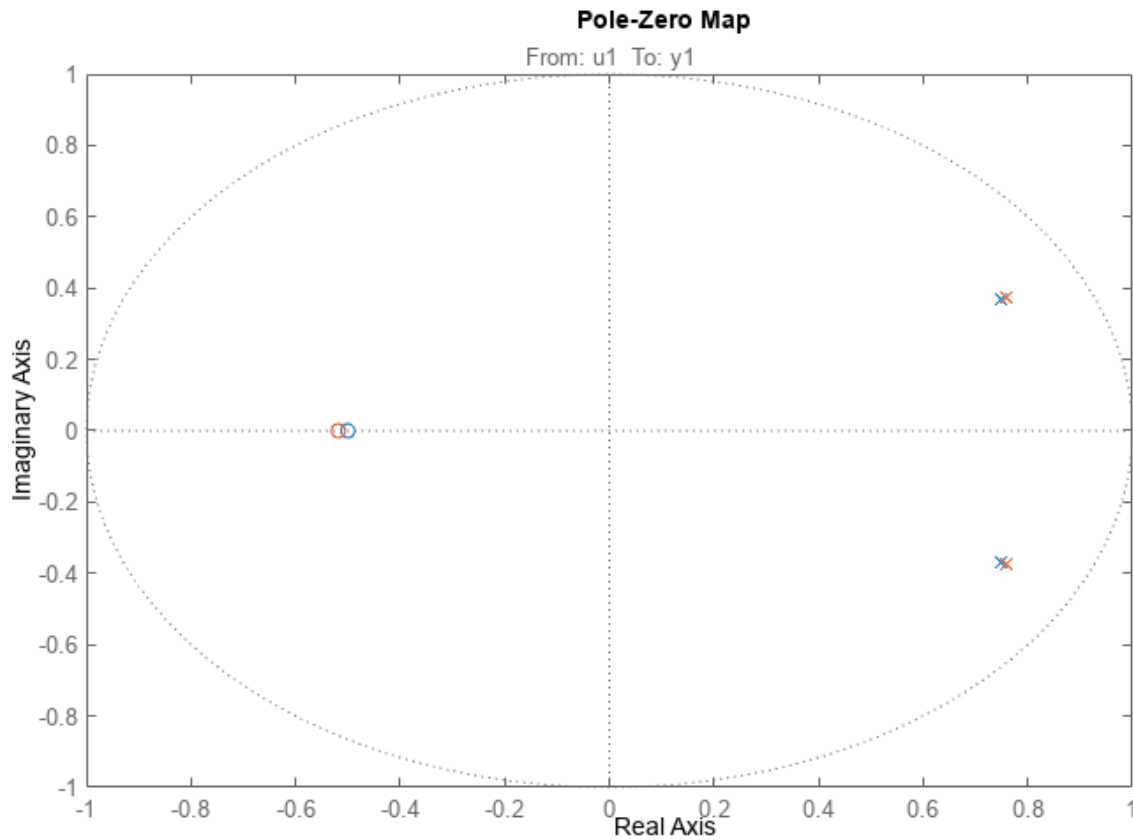
It can be seen that the poles and the zeros of the true system (blue) and the ARMAX model (green) are very close.

We can also evaluate the uncertainty of the zeros and poles. To plot confidence regions around the estimated poles and zeros corresponding to 3 standard deviations, use `showConfidence` or turn on the "Confidence Region" characteristic from the plot's context (right-click) menu.

```
showConfidence(h,3)
```

%
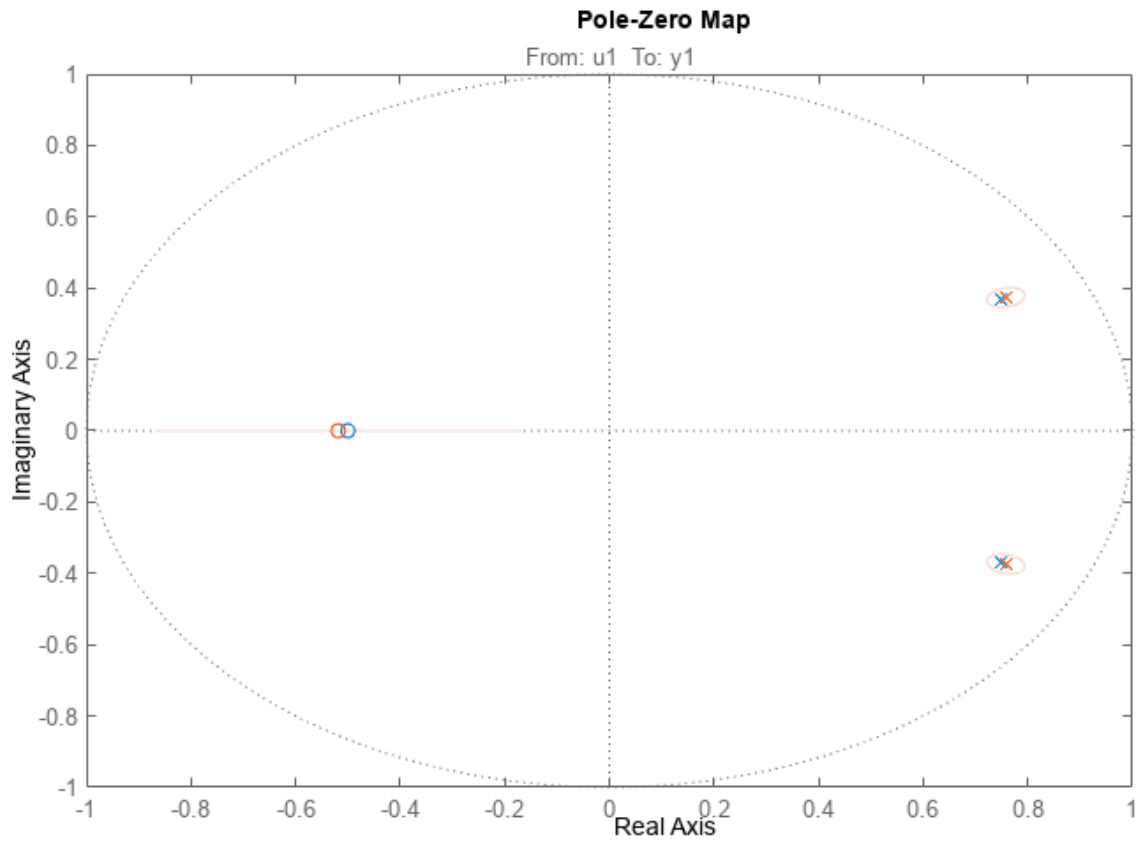
We see that the true, blue zeros and poles are well inside the green uncertainty regions.

Estimating Continuous-Time Models Using Simulink Data

This example illustrates how models simulated in Simulink® can be identified using System Identification Toolbox™. The example describes how to deal with continuous-time systems and delays, as well as the importance of the intersample behavior of the input.

Acquiring Simulation Data from a Simulink Model

Consider the system described by the following Simulink model:

```
open_system('iddemsl1')
set_param('iddemsl1/Random Number','seed','0')
```
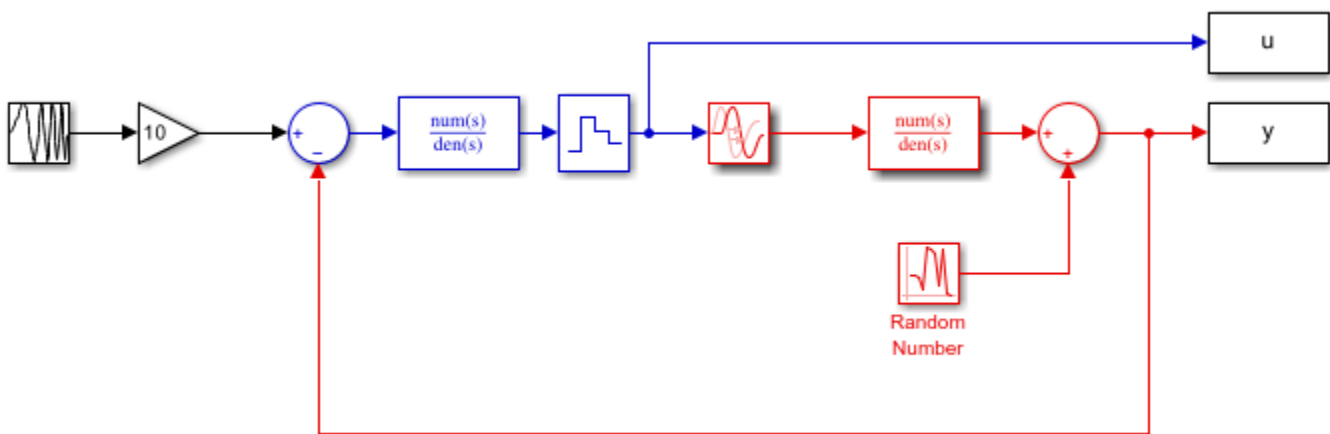

The red part is the system, the blue part is the controller and the reference signal is a swept sinusoid (a chirp signal). The data sample time is set to 0.5 seconds.

This system can be represented using an `idpoly` structure:

```
m0 = idpoly(1,0.1,1,1,[1 0.5],'Ts',0,'InputDelay',1,'NoiseVariance',0.01)
```

```
m0 =
Continuous-time OE model: y(t) = [B(s)/F(s)]u(t) + e(t)
  B(s) = 0.1

  F(s) = s + 0.5

Input delays (listed by channel): 1
Parameterization:
   Polynomial orders:   nb=1   nf=1   nk=0
   Number of free coefficients: 2
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

Let us simulate the model `iddemsl1` and save the data in an `iddata` object:

```
sim('iddemsl1')
dat1e = iddata(y,u,0.5); % The IDDATA object
```

Let us do a second simulation of the mode for validation purposes.

```
set_param('iddemsl1/Random Number','seed','13')
sim('iddemsl1')
dat1v = iddata(y,u,0.5);
```

Let us have a peek at the estimation data obtained during the first simulation:

```
plot(dat1e)
```
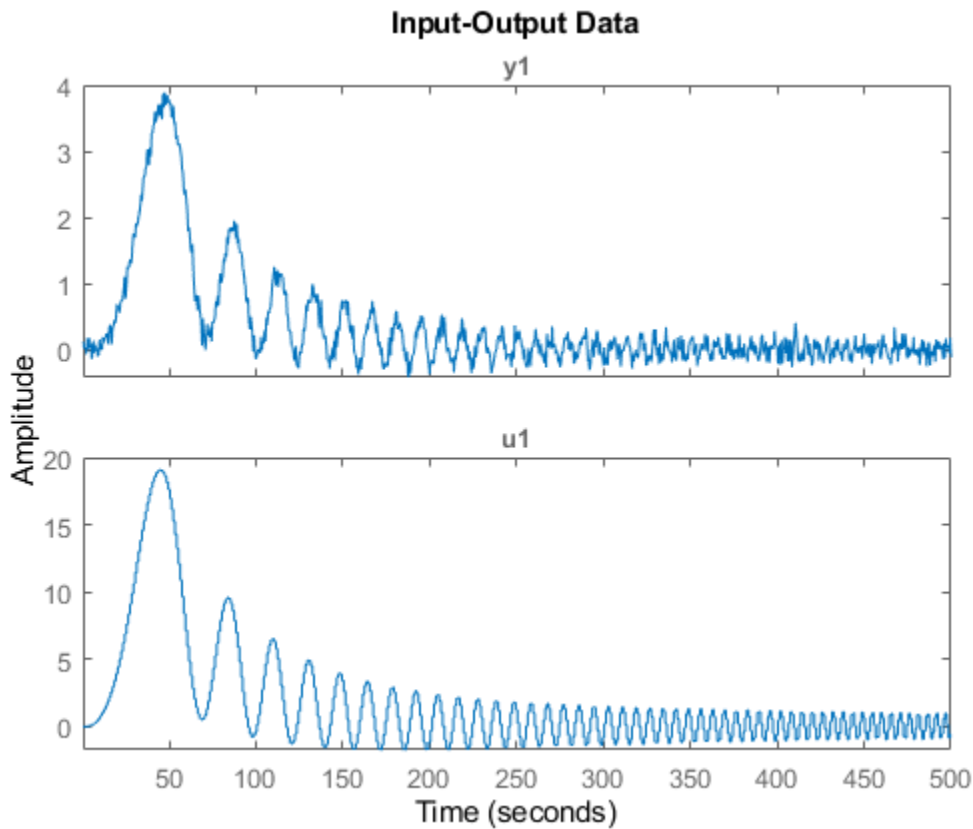

Estimating Discrete Models Using the Simulation Data

Let us begin by evaluating a default-order discrete model to gain some preliminary insight into the data characteristics:

```
m1 = n4sid(dat1e, 'best')      % A default order model
```

```
m1 =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + B u(t) + K e(t)
        y(t) = C x(t) + D u(t) + e(t)

  A =
            x1        x2        x3
```

```
     x1    0.7881    0.1643    0.1116
     x2   -0.1214    0.4223   -0.8489
     x3     0.155    0.7527    0.2119

   B =
                 u1
     x1  -0.0006427
     x2    -0.02218
     x3     0.07347

   C =
             x1       x2       x3
     y1  -5.591    0.871    1.189

   D =
         u1
     y1   0

   K =
                 y1
     x1  -0.001856
     x2   0.002363
     x3   -0.06805

Sample time: 0.5 seconds

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: estimate
   Number of free coefficients: 18
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using N4SID on time domain data "dat1e".
Fit to estimation data: 86.17% (prediction focus)
FPE: 0.01281, MSE: 0.01251
```

Check how well the model reproduces the validation data.

```
compare(dat1v,m1)
```

Simulated Response Comparison
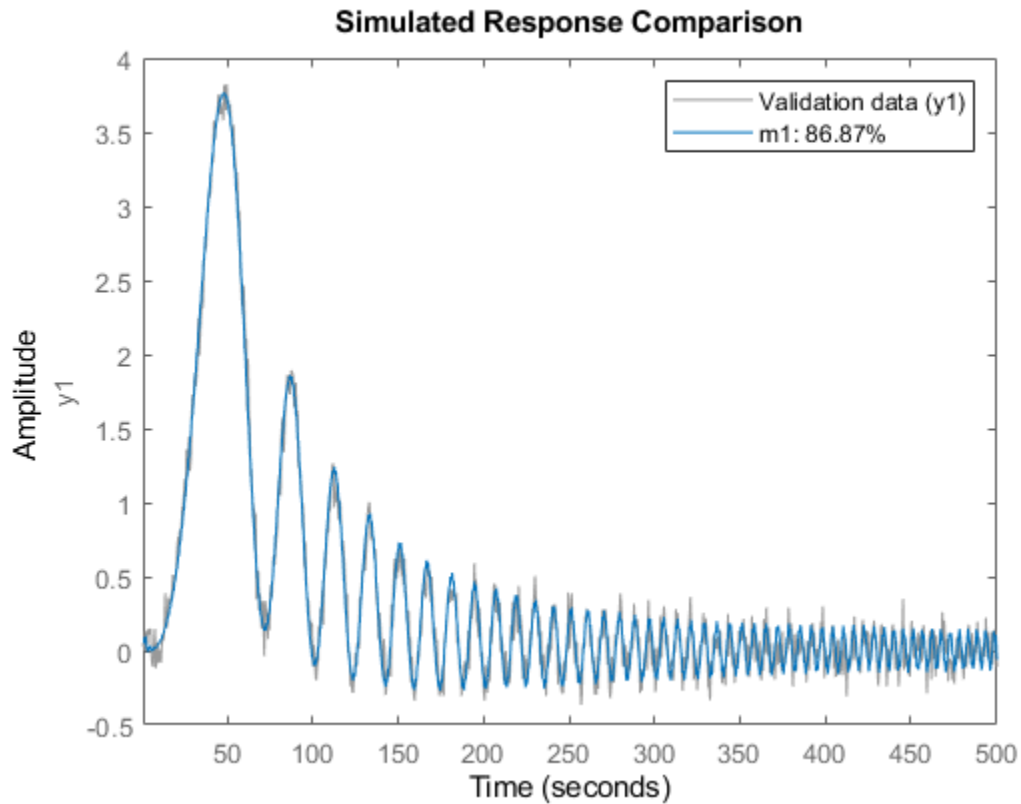
As observed, the validation data is predicted well by the model. To investigate more into the data characteristics, let us inspect the non-parametric impulse response computed using `dat1e` where the negative lags for analysis are automatically determined:

```
ImpModel = impulseest(dat1e,[],'negative');
clf
h = impulseplot(ImpModel);
showConfidence(h,3)
```
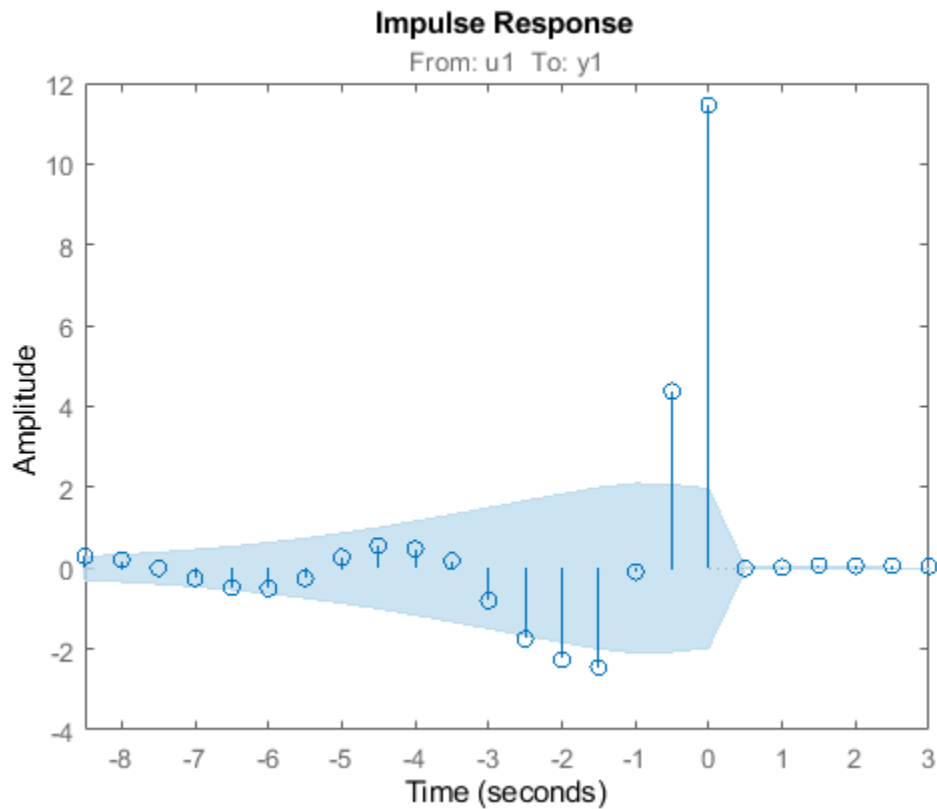
ImpModel is an FIR model whose order (no. of coefficients) are automatically determined. We also choose to analyze feedback effects by computing impulse response for negative delays. Influences from negative lags are not all insignificant. This is due to the regulator (output feedback). This means that the impulse response estimate cannot be used to determine the time delay. Instead, build several low order ARX-models with different delays and find out the best fit:

```
V = arxstruc(dat1e,dat1v,struc(1:2,1:2,1:10));
nn = selstruc(V,0) %delay is the third element of nn
```

```
nn =

    2    2    3
```

The delay is determined to 3 lags. (This is correct: the deadtime of 1 second gives two lag-delays, and the ZOH-block another one.) The corresponding ARX-model can also be computed, as follows:

```
m2 = arx(dat1e,nn)
compare(dat1v,m1,m2);
```

```
m2 =
Discrete-time ARX model: A(z)y(t) = B(z)u(t) + e(t)
  A(z) = 1 - 0.2568 z^-1 - 0.3372 z^-2

  B(z) = 0.04021 z^-3 + 0.04022 z^-4
```

```
Sample time: 0.5 seconds

Parameterization:
   Polynomial orders:    na=2    nb=2    nk=3
   Number of free coefficients: 4
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using ARX on time domain data "dat1e".
Fit to estimation data: 85.73% (prediction focus)
FPE: 0.01346, MSE: 0.0133
```
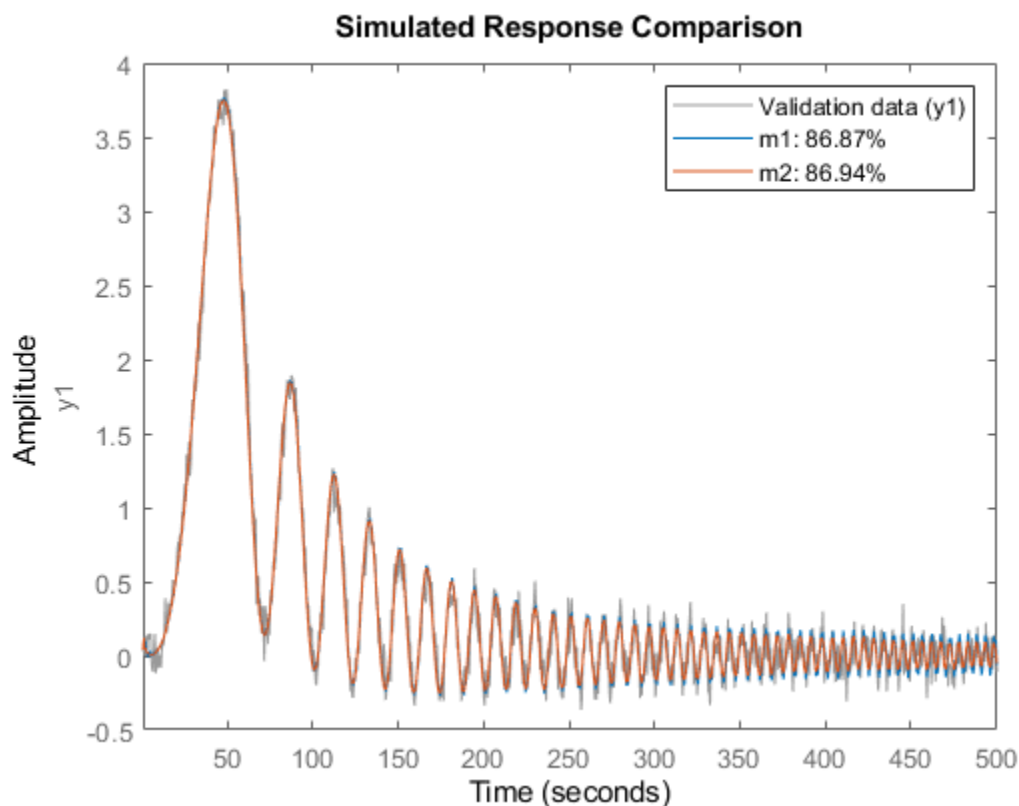

Refining the Estimation

The two models m1 and m2 behave similarly in simulation. Let us now try and fine-tune orders and delays. Fix the delay to 2 (which coupled with a lack of feedthrough gives a net delay of 3 samples) and find a default order state-space model with that delay:

```
m3 = n4sid(dat1e,'best','InputDelay',2,'Feedthrough',false);
% Refinement for prediction error minimization using pem (could also use
% |ssest|)
m3 = pem(dat1e, m3);
```

Let us look at the estimated system matrix.

```
m3.a  % the A-matrix of the resulting model
```

```
ans =

    0.7332   -0.3784    0.1735
    0.4705    0.3137   -0.6955
   -0.0267    0.7527    0.6343
```

A third order dynamics is automatically chosen, which together with the 2 "extra" delays gives a 5th order state space model.

It is always advisable not to blindly rely upon automatic order choices. They are influenced by random errors. A good way is to look at the model's zeros and poles, along with confidence regions:

```
clf
h = iopzplot(m3);
showConfidence(h,2) % Confidence region corresponding to 2 standard deviations
```
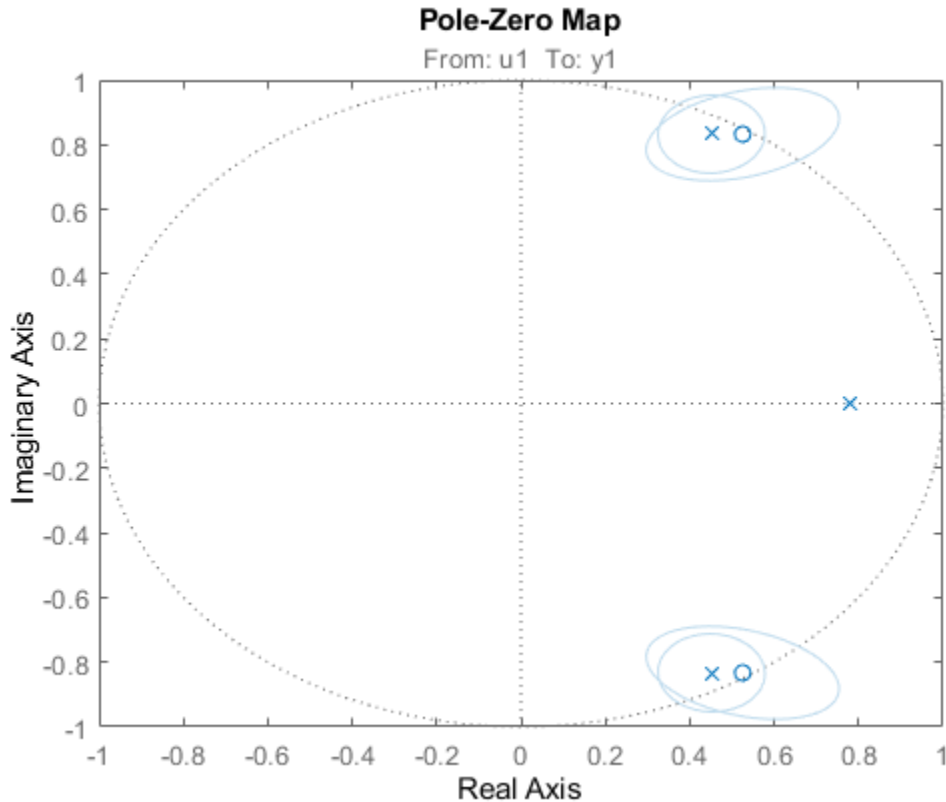

Clearly the two poles/zeros at the unit circle seem to cancel, indicating that a first order dynamics might be sufficient. Using this information, let us do a new first-order estimation:

```
m4 = ssest(dat1e,1,'Feedthrough',false,'InputDelay',2,'Ts',dat1e.Ts);
compare(dat1v,m4,m3,m1)
```

Simulated Response Comparison
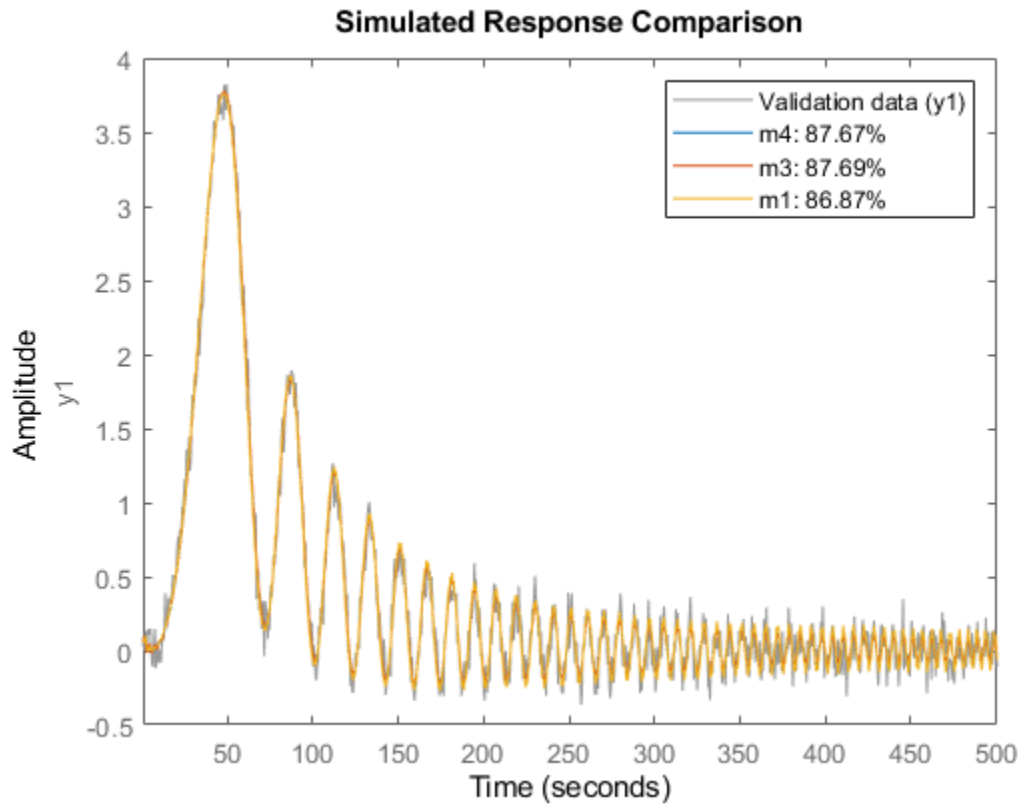
The `compare` plot shows that the simple first order model m4 gives a very good description of the data. Thus we shall select this model as our final result.

Converting Discrete Model to Continuous-Time (LTI)

Convert this model to continuous time, and represent it in transfer function form:

```
mc = d2c(m4);
idtf(mc)
```

```
ans =
  From input "u1" to output "y1":
                0.09828
  exp(-1*s) * ----------
              s + 0.4903

Continuous-time identified transfer function.

Parameterization:
   Number of poles: 1   Number of zeros: 0
   Number of free coefficients: 2
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

A good description of the system has been obtained, as displayed above.

Estimating Continuous-Time Model Directly

The continuous time model can also be estimated directly. The discrete model m4 has 2 sample input delay which represents a 1 second delay. We use the `ssest` command for this estimation:

```
m5 = ssest(dat1e,1,'Feedthrough',false,'InputDelay',1);
present(m5)
```

```
m5 =
  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
   x1  -0.4903 +/- 0.0079

  B =
                        u1
   x1  0.01345 +/- 7.803e+10

  C =
                     x1
   y1  7.307 +/- 4.239e+13

  D =
      u1
   y1   0

  K =
                       y1
   x1  -0.02227 +/- 1.292e+11

  Input delays (seconds): 1

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: estimate
   Number of free coefficients: 4
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Termination condition: No improvement along the search direction with line search..
Number of iterations: 9, Number of function evaluations: 129

Estimated using SSEST on time domain data "dat1e".
Fit to estimation data: 87.34% (prediction focus)
FPE: 0.01054, MSE: 0.01047
More information in model's "Report" property.
```

Uncertainty Analysis

The parameters of model m5 exhibit high levels of uncertainty even though the model fits the data 87%. This is because the model uses more parameters than absolutely required leading to a loss of uniqueness in parameter estimates. To view the true effect of uncertainty in the model, there are two possible approaches:

1 View the uncertainty as confidence bounds on model's response rather than on the parameters.

2 Estimate the model in canonical form.

Let use try both approaches. First we estimate the model in canonical form.

```
m5Canon = ssest(dat1e,1,'Feedthrough',false,'InputDelay',1,'Form','canonical');
present(m5Canon)
```

```
m5Canon =
  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
   x1   -0.4903 +/- 0.007881

  B =
                      u1
   x1   0.09828 +/- 0.001559

  C =
       x1
   y1    1

  D =
       u1
   y1    0

  K =
                      y1
   x1   -0.1628 +/- 0.03702

  Input delays (seconds): 1

Parameterization:
   CANONICAL form with indices: 1.
   Feedthrough: none
   Disturbance component: estimate
   Number of free coefficients: 3
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Termination condition: No improvement along the search direction with line search..
Number of iterations: 9, Number of function evaluations: 129

Estimated using SSEST on time domain data "dat1e".
Fit to estimation data: 87.34% (prediction focus)
FPE: 0.01054, MSE: 0.01047
```

More information in model's "Report" property.

m5Canon uses a canonical parameterization of the model. It fits the estimation data as good as the model m5. It shows small uncertainties in the values of its parameters giving an evidence of its reliability. However, as we saw for m5, a large uncertainty does not necessarily mean a "bad" model. To ascertain the quality of these models, let use view their responses in time and frequency domains with confidence regions corresponding to 3 standard deviations. We also plot the original system m0 for comparison.

The bode plot.

```
clf
opt = bodeoptions;
opt.FreqScale = 'linear';
h = bodeplot(m0,m5,m5Canon,opt);
showConfidence(h,3)
legend show
```
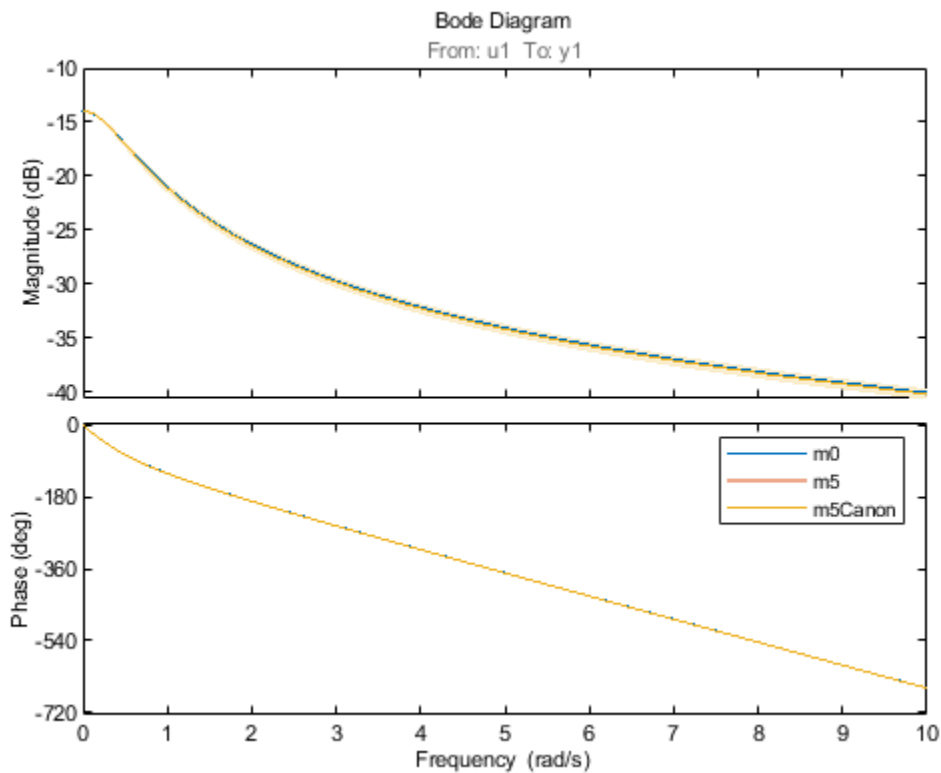

The step plot.

```
clf
showConfidence(stepplot(m0,m5,m5Canon),3)
legend show
```
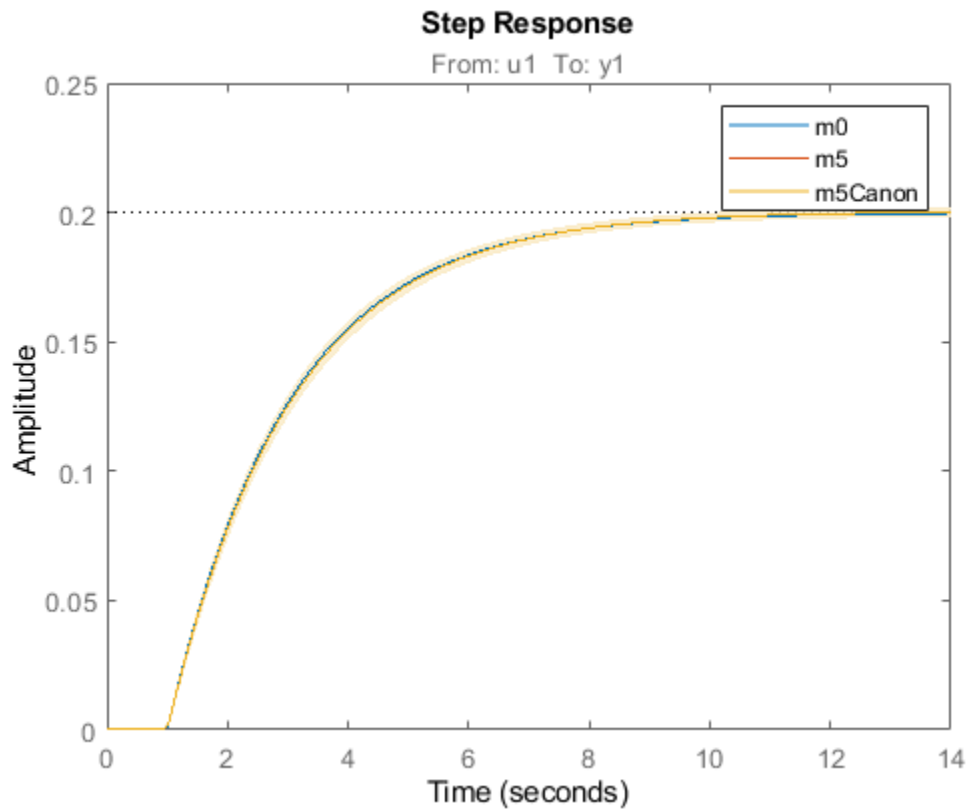
The uncertainty bounds for the two models are virtually identical. We can similarly generate pole-zero map (`iopzplot`) and Nyquist plot (`nyquistplot`) with confidence regions for these models.

`idtf(m5)`

```
ans =
  From input "u1" to output "y1":
                0.09828
  exp(-1*s) * ----------
                s + 0.4903

Continuous-time identified transfer function.

Parameterization:
   Number of poles: 1    Number of zeros: 0
   Number of free coefficients: 2
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by conversion from idss model.
```

Accounting for Intersample Behavior in Continuous-Time Estimation

When comparing continuous time models computed from sampled data, it is important to consider the intersample behavior of the input signal. In the example so far, the input to the system was piece-

wise constant, due to the Zero-order-Hold (zoh) circuit in the controller. Now remove this circuit, and consider a truly continuous system. The input and output signals are still sampled a 2 Hz, and everything else is the same:

```
open_system('iddemsl3')
sim('iddemsl3')
dat2e = iddata(y,u,0.5);
```
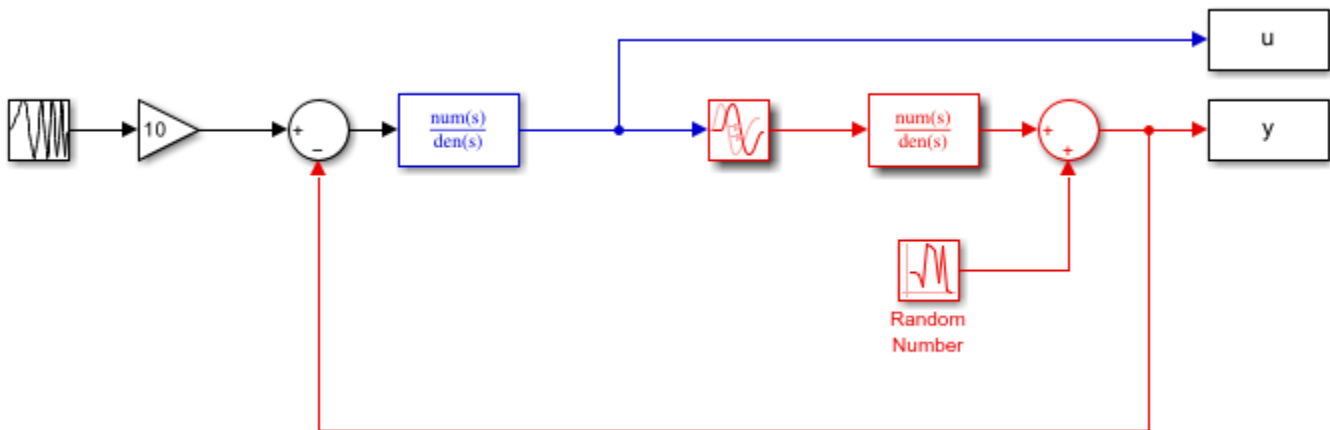

Discrete time models will still do well on these data, since when they are adjusted to the measurements, they will incorporate the sampling properties, and intersample input behavior (for the current input). However, when building continuous time models, knowing the intersample properties is essential. First build a model just as for the ZOH case:

```
m6 = ssest(dat2e,1,'Feedthrough',false,'InputDelay',1,'Form','canonical');
idtf(m6)
```

```
ans =
  From input "u1" to output "y1":
               0.1117
  exp(-1*s) * ----------
             s + 0.5596

Continuous-time identified transfer function.

Parameterization:
   Number of poles: 1   Number of zeros: 0
   Number of free coefficients: 2
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by conversion from idss model.
```

Let us compare the estimated model (m6) against the true model (m0):

```
step(m6,m0)   % Compare with true system
```

4-159
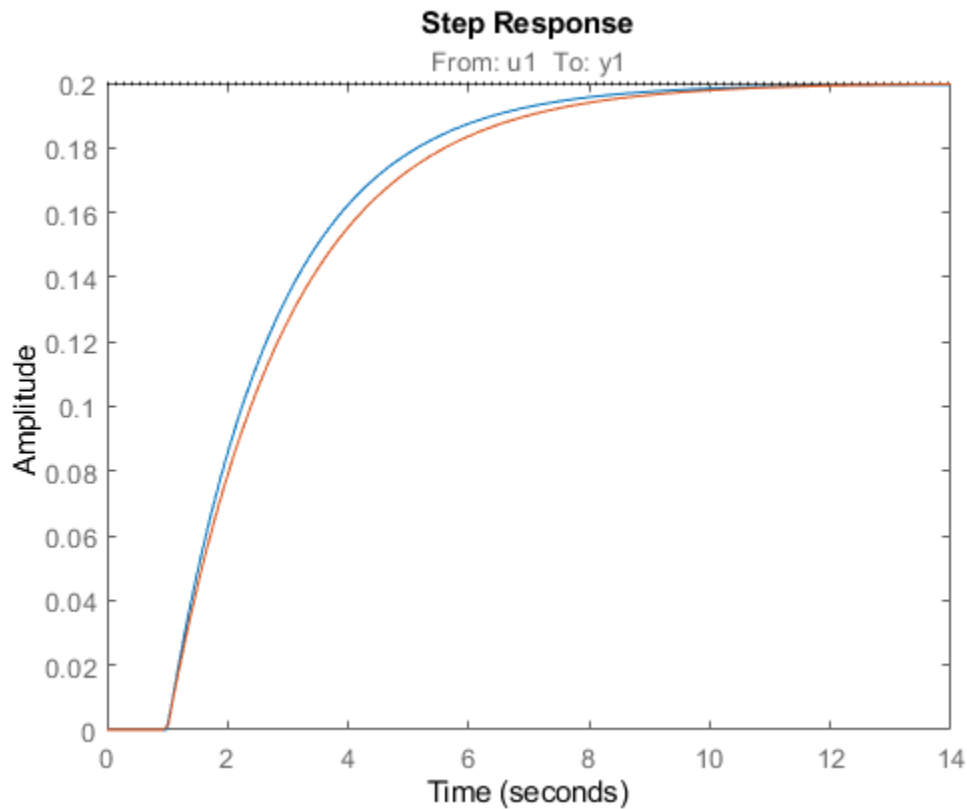
The agreement is now not so good. We may, however, include in the data object information about the input. As an approximation, let it be described as piecewise linear (First-order-hold, FOH) between the sampling instants. This information is then used by the estimator for proper sampling:

```
dat2e.Intersample = 'foh';
m7 = ssest(dat2e,1,'Feedthrough',false,'InputDelay',1,'Form','canonical');  % new estimation with
idtf(m7)
```

```
ans =
  From input "u1" to output "y1":
               0.09937
  exp(-1*s) * ----------
               s + 0.4957

Continuous-time identified transfer function.

Parameterization:
   Number of poles: 1    Number of zeros: 0
   Number of free coefficients: 2
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by conversion from idss model.
```

Let us look at the step response comparison again:

```
step(m7,m0)  % Compare with true system
```

Step Response

From: u1 To: y1
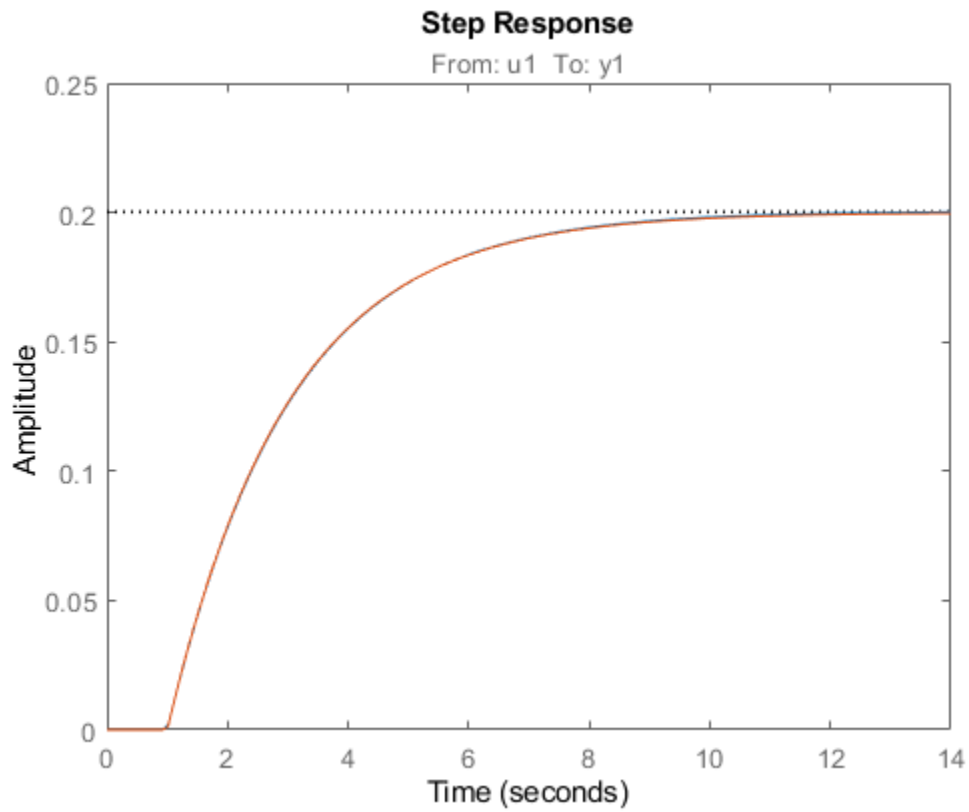
This model (m7) gives a much better result than m6. This concludes this example.

```
bdclose('iddemsl1');
bdclose('iddemsl3');
```

Linear Approximation of Complex Systems by Identification

This example shows how to obtain linear approximations of a complex, nonlinear system by means of linear model identification. The approach is based on selection of an input signal that excites the system. A linear approximation is obtained by fitting a linear model to the simulated response of the nonlinear model for the chosen input signal.

This example uses Simulink®, Control System Toolbox™ and Simulink Control Design™.

Introduction

In many situations, a linear model is obtained by simplification of a more complex nonlinear system under certain local conditions. For example, a high fidelity model of the aircraft dynamics may be described by a detailed Simulink model. A common approach taken to speed up the simulation of such systems, study their local behavior about an operating point or design compensators is to linearize them. If we perform the linearization of the original model analytically about an operating point, this, in general, would yield a model of an order as high as, or close to, the number of states in the original model. This order may be unnecessarily high for the class of inputs it is supposed to be used with for analysis or control system design. Hence we can consider an alternative approach centered around collecting input-output data from the simulation of the system and using it to derive a linear model of just the right order.

Analytical Linearization of F14 Model

Consider the F14 model. This is a already a linear model but contains derivative blocks and sources of disturbance that may affect the nature of its output. We can "linearize" it between its one input port and the two output ports as follows:

```
open_system('idF14Model')
IO = getlinio('idF14Model')
syslin = linearize('idF14Model',IO)
```

```
3x1 vector of Linearization IOs:
--------------------------
1. Linearization input perturbation located at the following signal:
- Block: idF14Model/Pilot
- Port: 1
- Signal Name: Stick Input
2. Linearization output measurement located at the following signal:
- Block: idF14Model/Gain5
- Port: 1
- Signal Name: Angle of Attack
3. Linearization output measurement located at the following signal:
- Block: idF14Model/Pilot G force
- Port: 1
- Signal Name: Pilot G force

syslin =

  A =
                 Transfer Fcn    Derivative   Transfer Fcn    Derivative1
     Transfer Fcn      -0.6385             0          689.4              0
     Derivative              1        -1e+05              0              0
     Transfer Fcn     -0.00592             0        -0.6571              0
```

```
Derivative1              0              0              1         -1e+05
Actuator Mod             0              0          1.424              0
Alpha-sensor      0.001451              0              0              0
Stick Prefil             0              0              0              0
Pitch Rate L             0              0              1              0
Proportional             0              0        -0.8156              0

                Actuator Mod  Alpha-sensor  Stick Prefil  Pitch Rate L
Transfer Fcn         -1280              0              0              0
Derivative               0              0              0              0
Transfer Fcn        -137.7              0              0              0
Derivative1              0              0              0              0
Actuator Mod           -20          2.986         -39.32          -1.67
Alpha-sensor             0         -2.526              0              0
Stick Prefil             0              0         -22.52              0
Pitch Rate L             0              0              0         -4.144
Proportional             0          -1.71          22.52         0.9567

                Proportional
Transfer Fcn             0
Derivative               0
Transfer Fcn             0
Derivative1              0
Actuator Mod        -3.864
Alpha-sensor             0
Stick Prefil             0
Pitch Rate L             0
Proportional             0

B =
                Stick Input
Transfer Fcn             0
Derivative               0
Transfer Fcn             0
Derivative1              0
Actuator Mod             0
Alpha-sensor             0
Stick Prefil             1
Pitch Rate L             0
Proportional             0

C =
                Transfer Fcn    Derivative  Transfer Fcn   Derivative1
Angle of Att        0.001451             0             0             0
Pilot G forc           -3106     3.106e+08     7.083e+04    -7.081e+09

                Actuator Mod  Alpha-sensor  Stick Prefil  Pitch Rate L
Angle of Att             0             0             0             0
Pilot G forc             0             0             0             0

                Proportional
Angle of Att             0
Pilot G forc             0

D =
                Stick Input
Angle of Att             0
Pilot G forc             0
```

Continuous-time state-space model.
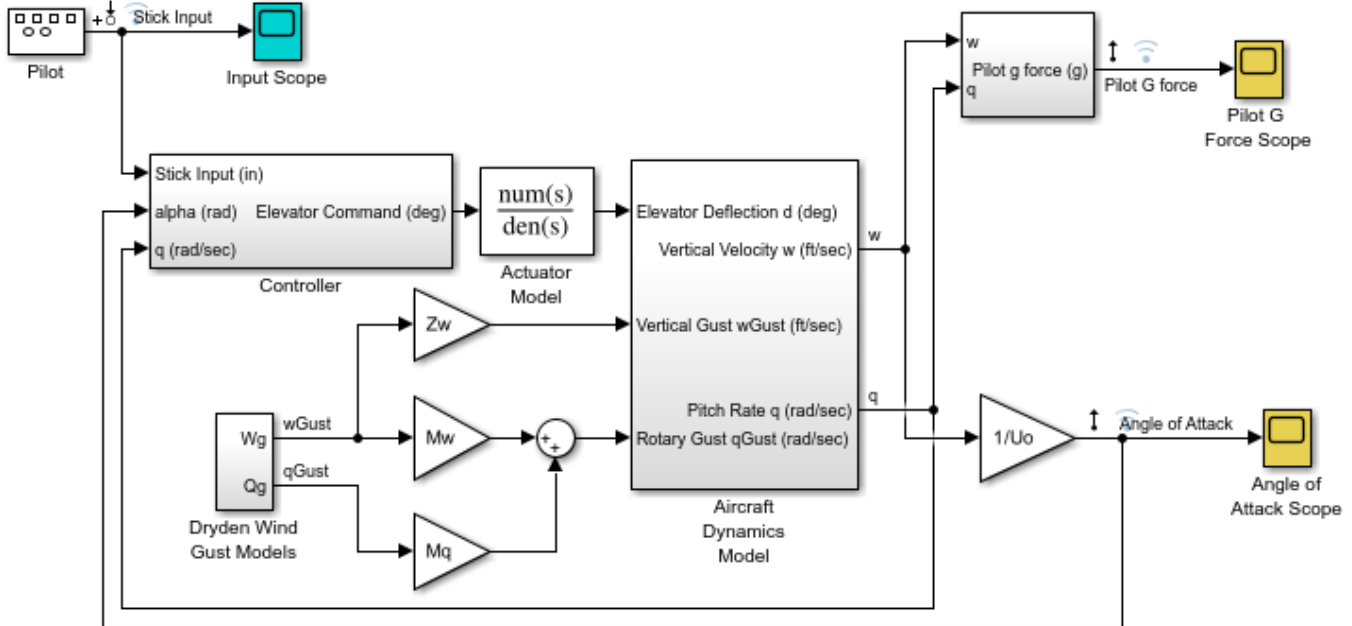
Copyright 1990-2011 The MathWorks, Inc.

syslin is a model with 2 outputs, one input and 9 states. This is because the linearization path from the "Stick Input" input to the two outputs has 9 states in the original system. We can verify this by using `operpoint`:

```
operpoint('idF14Model')
```

```
ans =


 Operating point for the Model idF14Model.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
x
_

(1.) idF14Model/Actuator Model
0
(2.) idF14Model/Aircraft Dynamics Model/Transfer Fcn.1
0
(3.) idF14Model/Aircraft Dynamics Model/Transfer Fcn.2
0
(4.) idF14Model/Controller/Alpha-sensor Low-pass Filter
0
```

```
(5.) idF14Model/Controller/Pitch Rate Lead Filter
0
(6.) idF14Model/Controller/Proportional plus integral compensator
0
(7.) idF14Model/Controller/Stick Prefilter
0
(8.) idF14Model/Dryden Wind Gust Models/Q-gust model
0
(9.) idF14Model/Dryden Wind Gust Models/W-gust model
0
0

Inputs: None
----------
```

Could this order be reduced while still maintaining fidelity to the responses for the chosen square-wave ("Stick input") input?

Preparing Identification Data

Simulate the model and log the input square wave (u) and the outputs "Angle of attack" (y1) and "Pilot G force" (y2) for 0:30 second time span. This data, after interpolation to a uniformly spaced time vector (sample time of 0.0444 seconds), is stored in the "idF14SimData.mat" file.

```
load idF14SimData
Z = iddata([y1, y2],u,Ts,'Tstart',0);
Z.InputName = 'Stick input'; Z.InputUnit = 'rad/s';
Z.OutputName = {'Angle of attack', 'Pilot G force'};
Z.OutputUnit = {'rad', 'g'};
t = Z.SamplingInstants;

subplot(311)
plot(t,Z.y(:,1)), ylabel('Angle of attack (rad)')
title('Logged Input-Output Data')

subplot(312)
plot(t,Z.y(:,2)), ylabel('Pilot G force (g)')

subplot(313)
plot(t,Z.u), ylabel('Stick input (rad/s)')
axis([0 30 -1.2 1.2])
xlabel('Time (seconds)')
```
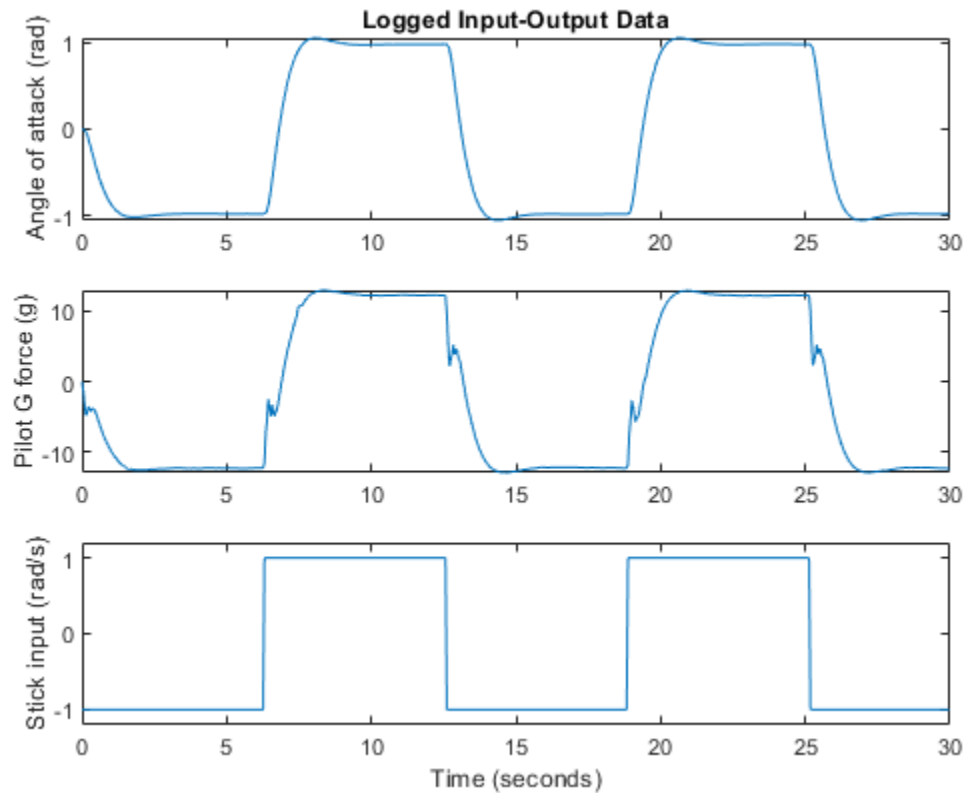
Estimation of State Space Models

Estimate state-space models of orders 2 to 4 using the `ssest` command. We configure the estimation to use "simulation" focus and choose to not estimate the disturbance component of the model.

```
opt = ssestOptions('Focus','simulation');
syslin2 = ssest(Z, 2, 'DisturbanceModel', 'none', opt);
syslin3 = ssest(Z, 3, 'DisturbanceModel', 'none', opt);
syslin4 = ssest(Z, 4, 'DisturbanceModel', 'none', opt);
```

Compare the performance of the linearized model `syslin` and the three identified models to the data. Note that syslin is an SS model while `syslin2`, `syslin3` and `syslin4` are IDSS models.

```
syslin.InputName = Z.InputName;
syslin.OutputName = Z.OutputName; % reconcile names to facilitate comparison
clf
compare(Z, syslin, syslin2, syslin3, syslin4)
```
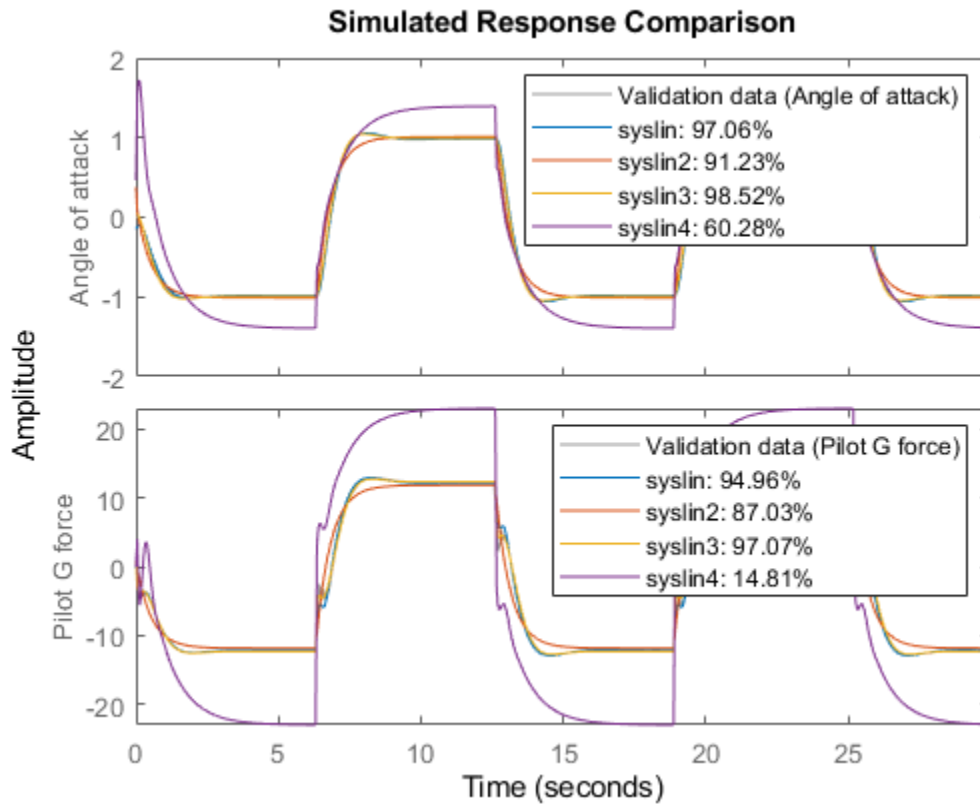
The plot shows that the 3rd order model (`syslin3`) works pretty well as a linear approximation of aircraft dynamic about its default (t=0) operating conditions. It fits the data slightly better than the one obtained by analytical linearization (`syslin`). If the original idF14Model is linear, why doesn't its linearization result, `syslin`, produce a 100% fit to the data? There are two reasons for this:

1 The measured outputs are affected by the wind gusts which means that the logged outputs are not simply a function of the Stick input. There are disturbances affecting it.

2 The "Pilot G force" block uses derivative blocks whose linearization depends upon the value of the time constant "c". "c" should be small (we use 1e-5) but not zero. A non-zero value for c introduces an approximation error in the linearization.

Let us view the parameters of the model `syslin3` which seems to capture the responses pretty well:

`syslin3`

```
syslin3 =
  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       x3
    x1   -1.006   -2.029   0.5842
    x2    8.284   -19.39   -5.611
    x3    2.784   -12.63   -6.956
```

```
  B =
      Stick input
   x1       0.2614
   x2        5.512
   x3        3.606

  C =
                   x1        x2        x3
   Angle of att   -8.841   -0.5347    1.402
   Pilot G forc   -86.42   -15.85     66.12

  D =
              Stick input
   Angle of att          0
   Pilot G forc          0

  K =
      Angle of att  Pilot G forc
   x1            0             0
   x2            0             0
   x3            0             0

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 18
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on time domain data "Z".
Fit to estimation data: [98.4;97.02]%
FPE: 2.367e-05, MSE: 0.1103
```

Model Simplification by Reduction and Estimation

We can also take the approach of reducing the order of the linearized model `syslin` and refining the parameters of the reduced model to best fit the data Z. To figure out a good value for the reduced order, we use `hsvd`:

```
[S, BalData] = hsvd(syslin);
clf; bar(S)
```
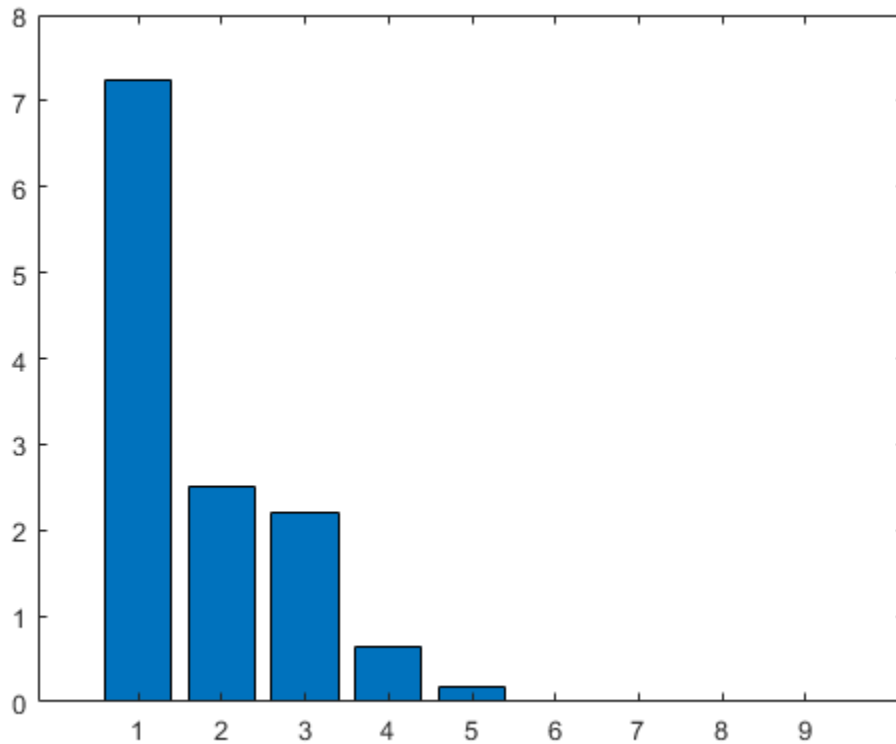
The bar chart shows that the singular values are quite small for states 4 and above. Hence 3rd order might be optimal for reduction.

```
sysr = balred(syslin,3,BalData)
opt2 = bodeoptions; opt2.PhaseMatching = 'on';
clf; bodeplot(sysr,syslin,opt2)
```

```
sysr =

  A =
            x1        x2        x3
   x1    -2.854    -7.61    -54.04
   x2   -0.9714    2.341     9.123
   x3    0.6979   -7.203    -24.08

  B =
      Stick input
   x1       -137.7
   x2         -869
   x3       -506.7

  C =
                        x1          x2          x3
   Angle of att  -0.0005063  -0.0008826   -0.001016
   Pilot G forc   -0.005926    -0.04692     -0.1646

  D =
```

```
                 Stick input
   Angle of att      -0.03784
   Pilot G forc        -1.617

Continuous-time state-space model.
```


Bode Diagram
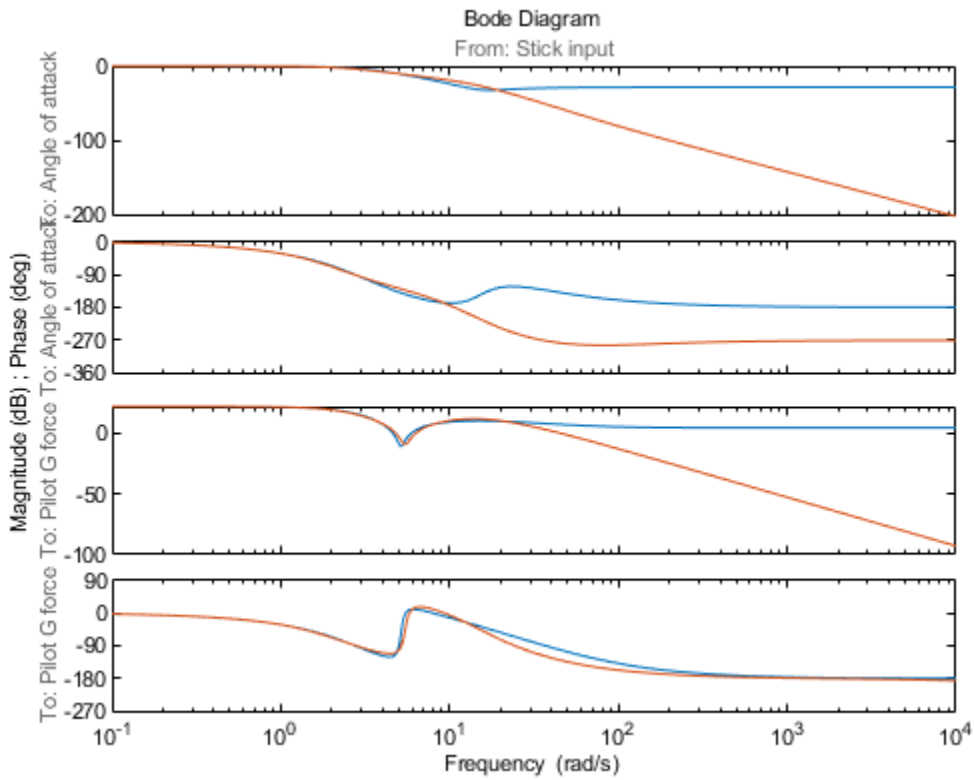
The bode plot shows good fidelity up to 10 rad/s. `sysr` is able to emulate the responses as good as the original 9-state model as shown by the `compare` plot:

```
compare(Z, sysr, syslin)
```
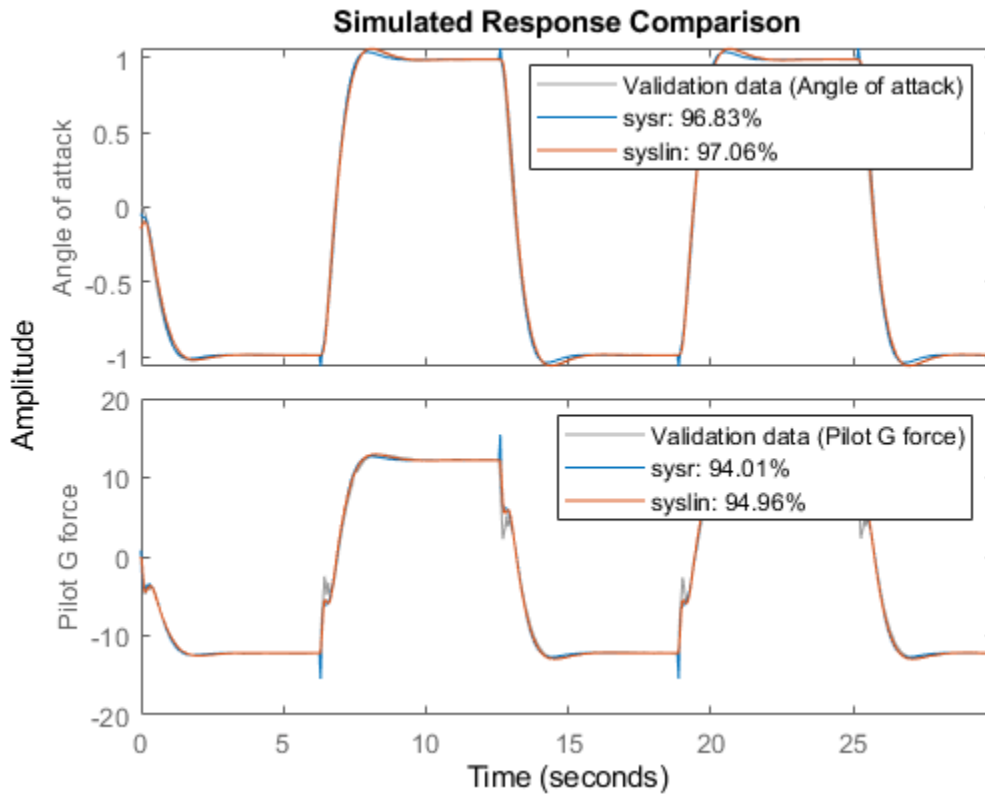
Let us refine the parameters of `sysr` to improve its fit to data. For this estimation, we choose the "Levenberg-Marquardt" search method and change the maximum number of allowable iterations to 10. These choices were made based on some trial and error. We also turn on the display of estimation progress.

```
opt.Display = 'on';
opt.SearchMethod = 'lm';
opt.SearchOptions.MaxIterations = 10;
sysr2 = ssest(Z, sysr, opt)
compare(Z, sysr2)
```

```
sysr2 =
  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        x3
   x1     -4.048    -7.681    -54.01
   x2    -0.4844     1.549     8.895
   x3    -0.2398    -6.777    -25.78

  B =
        Stick input
   x1        -137.7
   x2          -869
```

```
   x3         -506.7

 C =
                        x1            x2            x3
   Angle of att  -0.0003361   -0.0004964      0.00215
   Pilot G forc    -0.01191     -0.03599      -0.1434

 D =
               Stick input
   Angle of att    0.003022
   Pilot G forc      0.6438

 K =
       Angle of att  Pilot G forc
   x1             0             0
   x2             0             0
   x3             0             0

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: yes
   Disturbance component: none
   Number of free coefficients: 20
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on time domain data "Z".
Fit to estimation data: [98.78;97.03]%
FPE: 1.434e-05, MSE: 0.1097
```

```
Estimation data: Time domain data Z
Data has 2 outputs, 1 inputs and 676 samples.
Number of states: 3
```

ESTIMATION PROGRESS

```
Algorithm: Levenberg-Marquardt search
--------------------------------------------------------------------------------
-------
                        Norm of      First-order   Improvement (%)
Iteration      Cost     step          optimality   Expected  Achieved   Bisections
--------------------------------------------------------------------------------
-------
     0    0.00032574        -        3.66e+06      161           -          -
     1    0.000272415     2.84       1.35e+07      161         16.4        0
     2    1.52025e-05     0.0307     4.12e+06      127         94.4        1
     3    1.42291e-05     0.321      4.91e+06      23.8         6.4        0
     4    1.39223e-05     0.0249     4.72e+05      12.7         2.16       1
     5    1.38968e-05     0.022      8.37e+03       8          0.183       1
     6    1.38904e-05     0.0588     1.55e+04      4.94        0.0462      1
     7    1.38873e-05     0.0527     5.02e+04      12.2        0.0218      2
```

RESULT

```
Termination condition: Maximum number of iterations reached..
Number of iterations: 10, Number of function evaluations: 32

Status: Estimated using SSEST
Fit to estimation data: [98.78;97.03]%, FPE: 1.43383e-05
```

■ Stop   Close
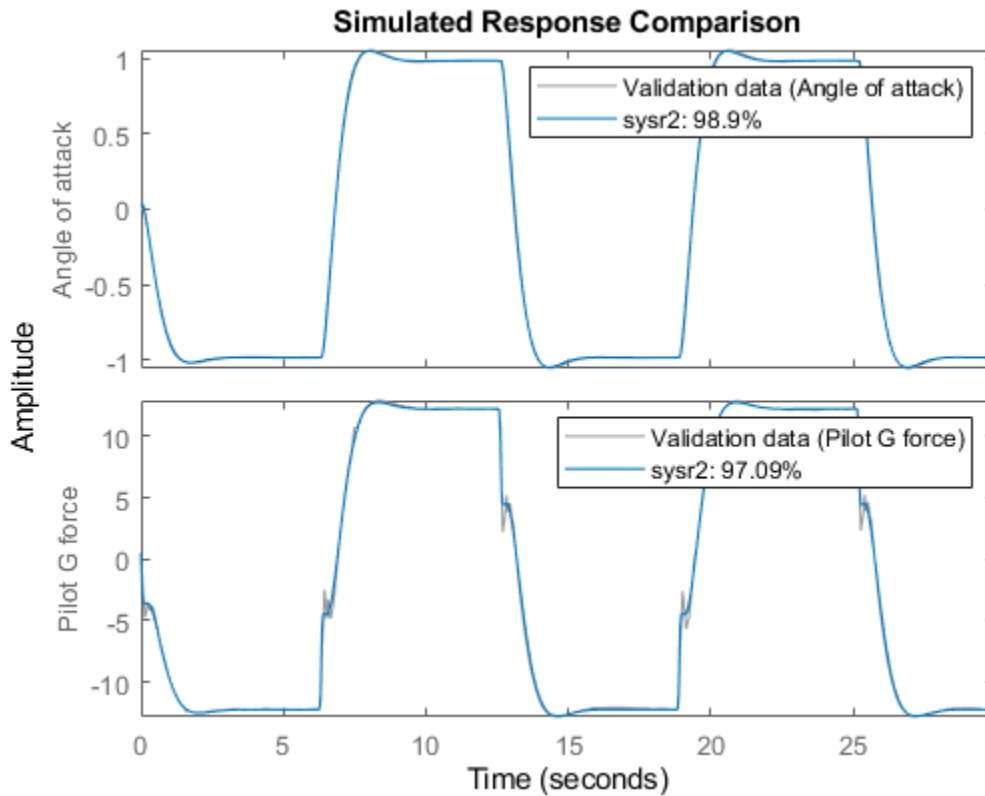
The refined model `sysr2` fits the response of the F14 model quite well (about 99% for the first output, and about 97% for the second).

Conclusions

We showed an alternative approach to analytical linearization for obtaining linear approximations of complex systems. The results are derived for a particular input signal and, strictly speaking, are applicable to that input only. To improve the applicability of results to various input profiles, we could perform several simulations using the various types of inputs. We could then merge the resulting datasets into one multi-experiment dataset (see `iddata/merge`) and use it for estimations. In this example, we used a complex, but linear system for convenience. The real benefit of this approach would be seen for nonlinear systems.

We also showed an approach for reducing the order of a linear system while keeping the reduced model faithful to the simulated response of the original Simulink model. The role of the reduced model `sysr` was to provide an initial guess for the estimated model `sysr2`. The approach also highlights the fact that any linear system, including those of a different class, can be used as the initial model for estimations.

```
bdclose('idF14Model')
```

Analyze Data in Multivariable Systems and Identify Models

This example shows how to deal with data with several input and output channels (MIMO data). Common operations, such as viewing the MIMO data, estimating and comparing models, and viewing the corresponding model responses are highlighted.

The Data Set

We start by looking at the data set SteamEng.

```
load SteamEng
```

This data set is collected from a laboratory scale steam engine. It has the inputs **Pressure** of the steam (actually compressed air) after the control valve, and **Magnetization voltage** over the generator connected to the output axis.

The outputs are **Generated voltage** in the generator and **Rotational speed** of the generator (Frequency of the generated AC voltage).The sample time was 50 ms.

First collect the measured channels into an `iddata` object:

```
steam = iddata([GenVolt,Speed],[Pressure,MagVolt],0.05);
steam.InputName  = {'Pressure';'MagVolt'};
steam.OutputName = {'GenVolt';'Speed'};
```

Let us have a look at the data

```
plot(steam(:,1,1))
```

Input-Output Data

GenVolt


```
plot(steam(:,1,2))
```

Input-Output Data


```
plot(steam(:,2,1))
```

Input-Output Data

Speed

Pressure

Time (seconds)

Amplitude
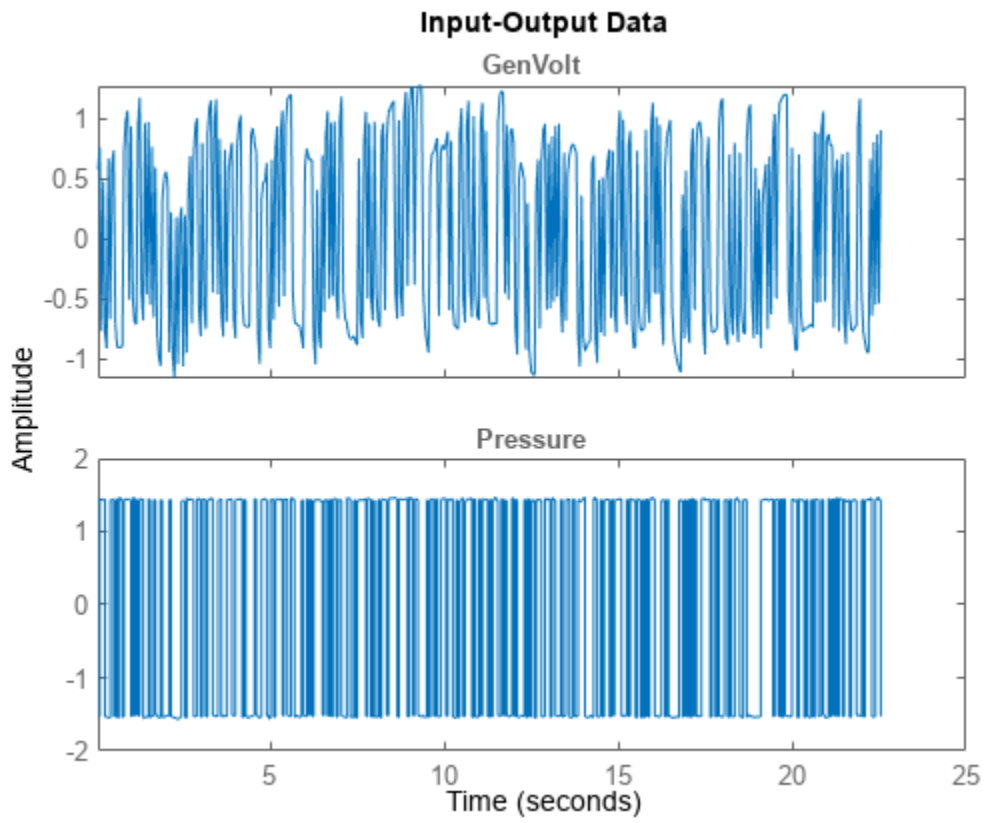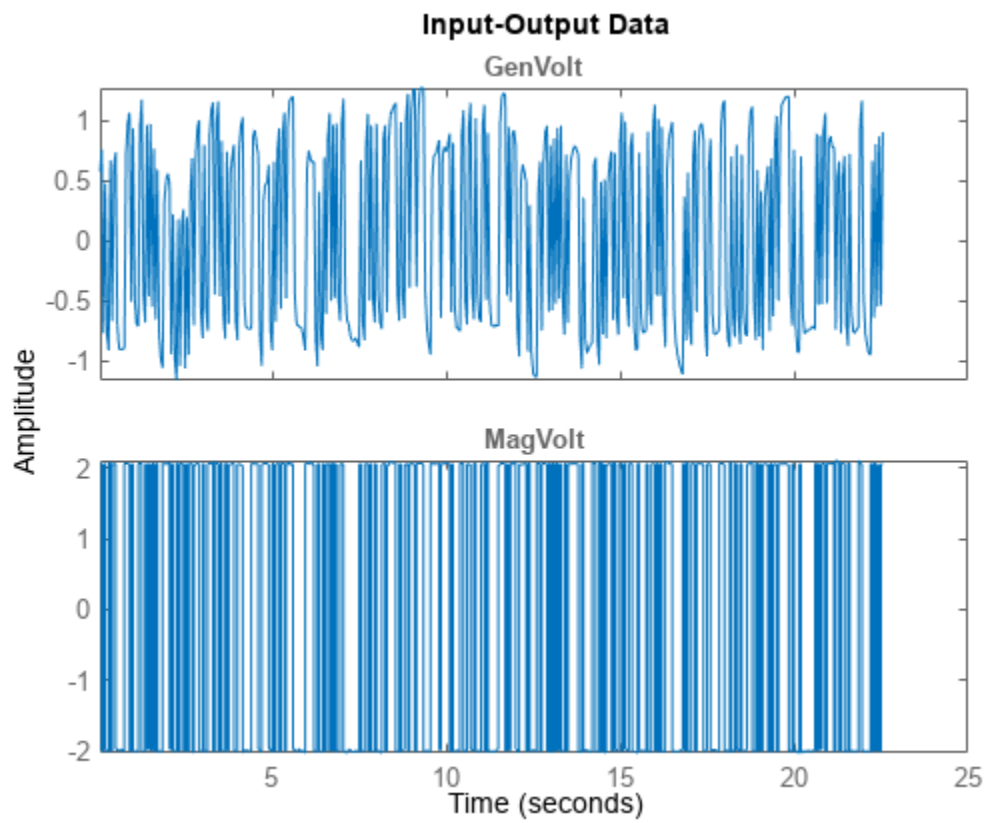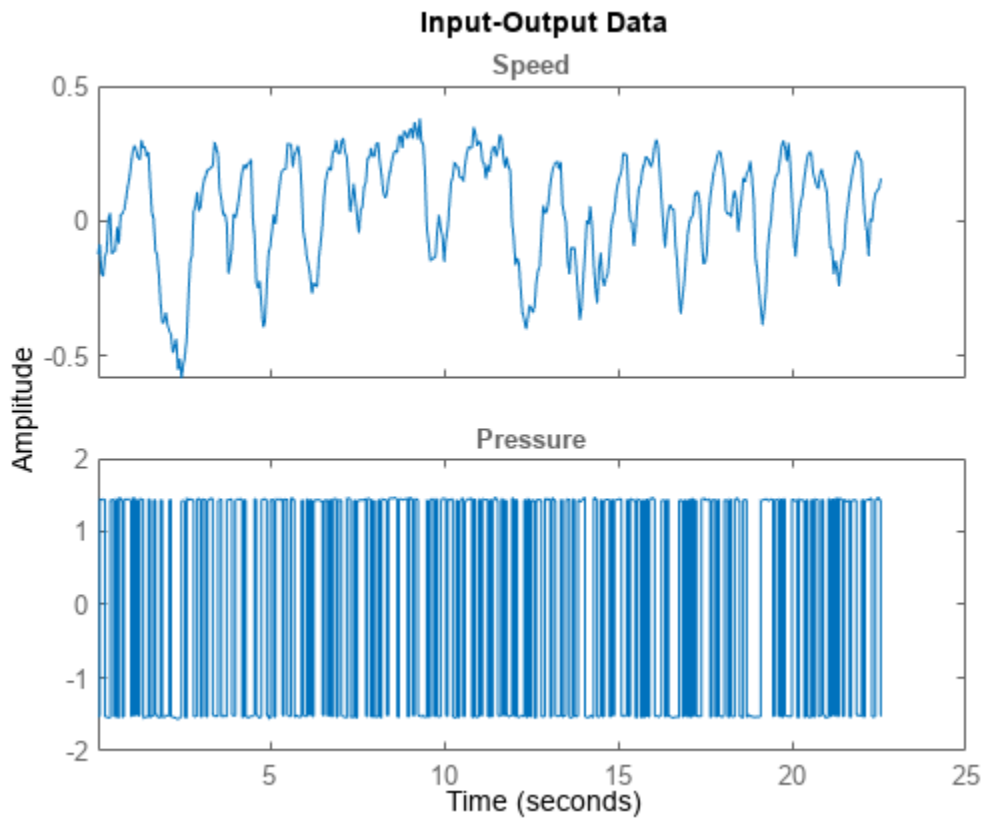
```
plot(steam(:,2,2))
```
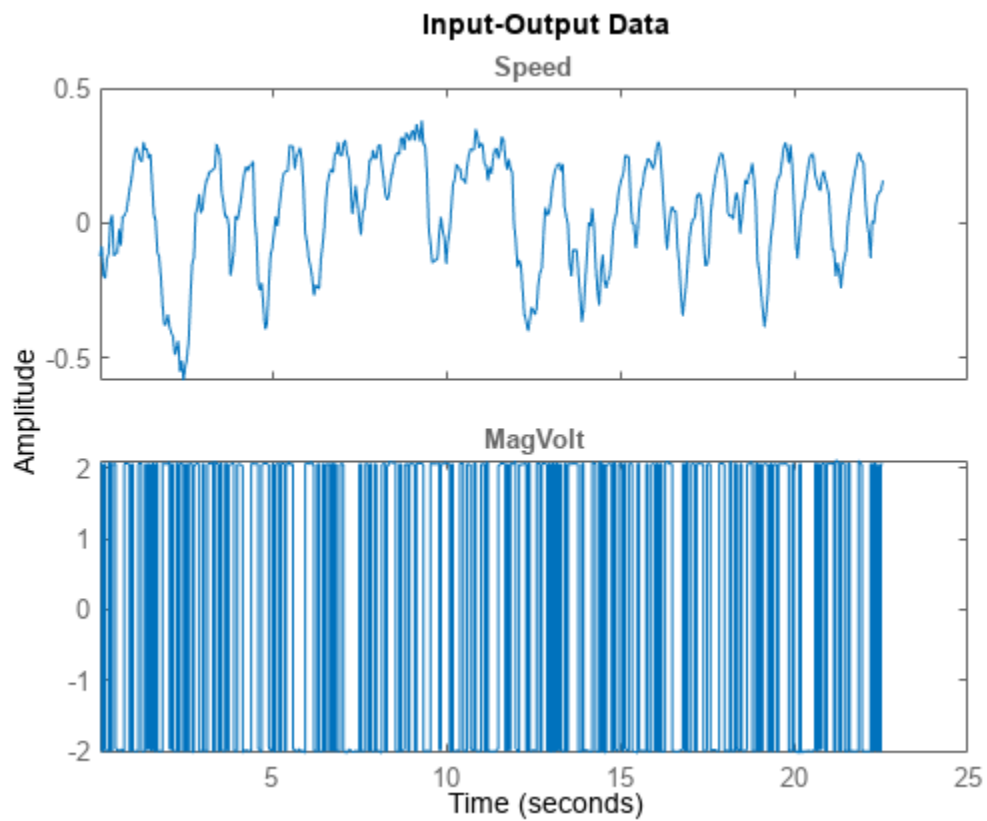
Step and Impulse Responses

A first step to get a feel for the dynamics is to look at the step responses between the different channels estimated directly from data:

```
mi = impulseest(steam,50);
clf, step(mi)
```
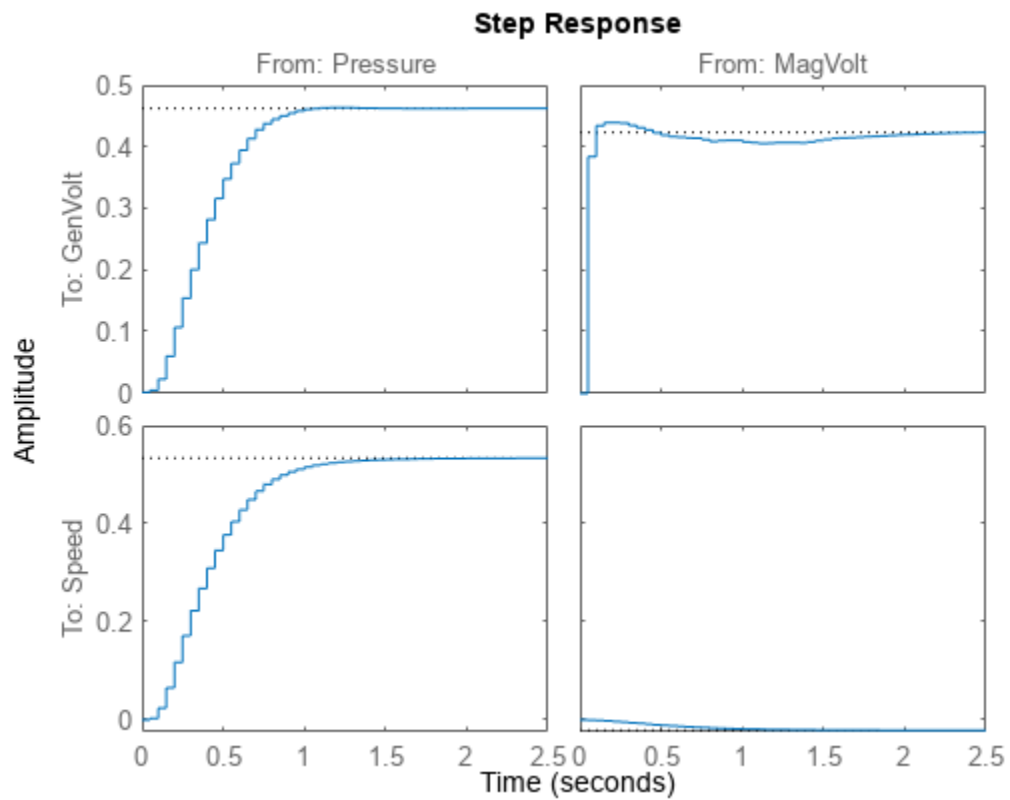
Responses with Confidence Regions

To look at the significance of the responses, the impulse plot can be used instead, with confidence regions corresponding to 3 standard deviations:

```
showConfidence(impulseplot(mi),3)
```
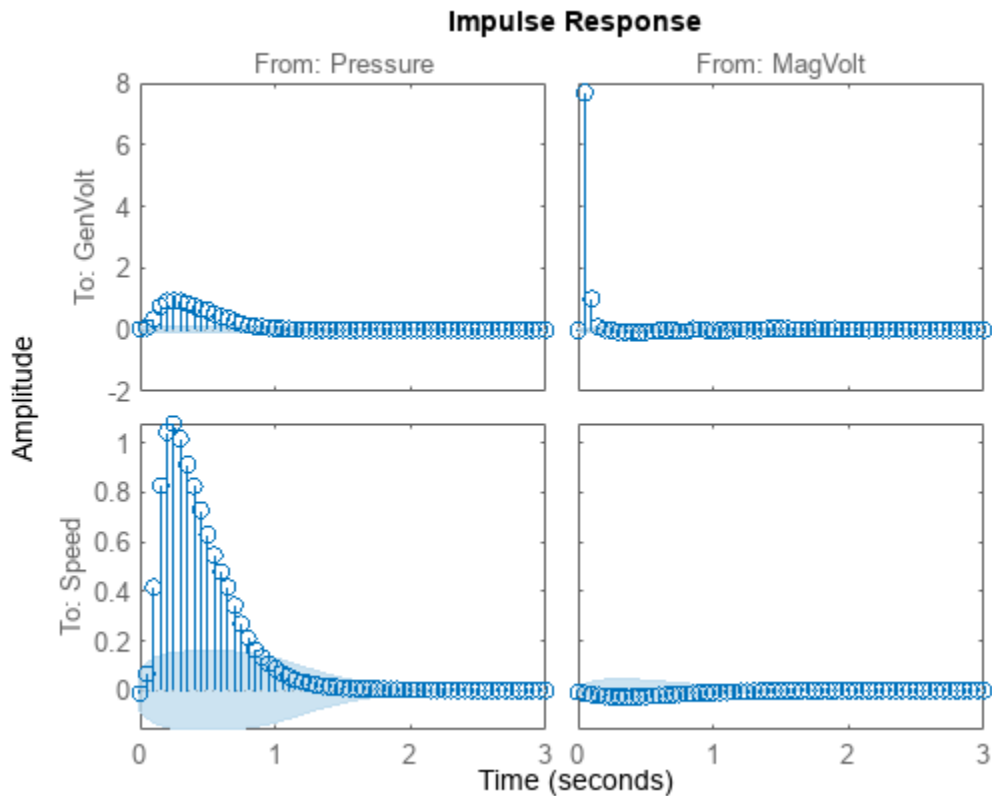
Clearly the off-diagonal influences dominate (Compare the y-scales!) That is, `GenVolt` is primarily affected by `MagVolt` (not much dynamics) and `Speed` primarily depends on `Pressure`. Apparently the response from `MagVolt` to `Speed` is not very significant.

A Two-Input-Two-Output Model

A quick first test is also to look a a default continuous time state-space prediction error model. Use only the first half of the data for estimation:

```
mp = ssest(steam(1:250))

mp =
  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        x3        x4
   x1    -29.43    -4.561    0.5994      -5.2
   x2    0.4849   -0.8662    -4.101    -2.336
   x3     2.839     5.084    -8.566    -3.855
   x4    -12.13    0.9224     1.818    -34.29

  B =
        Pressure    MagVolt
   x1     0.1033     -1.617
   x2    -0.3028   -0.09415
```

```
   x3    -1.566     0.2953
   x4  -0.04477    -2.681

  C =
                x1       x2       x3       x4
   GenVolt    -16.39   0.3767  -0.7566    2.808
   Speed      -5.623    2.246  -0.5356    3.423

  D =
            Pressure   MagVolt
   GenVolt         0         0
   Speed           0         0

  K =
       GenVolt    Speed
   x1  -0.3555   0.0853
   x2  -0.0231    5.195
   x3    1.526    2.132
   x4    1.787  0.03215

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: estimate
   Number of free coefficients: 40
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on time domain data.
Fit to estimation data: [86.9;74.84]% (prediction focus)
FPE: 3.897e-05, MSE: 0.01414
```

Compare with the step responses estimated directly from data:

```
h = stepplot(mi,'b',mp,'r',2); % Blue for direct estimate, red for mp
showConfidence(h)
```
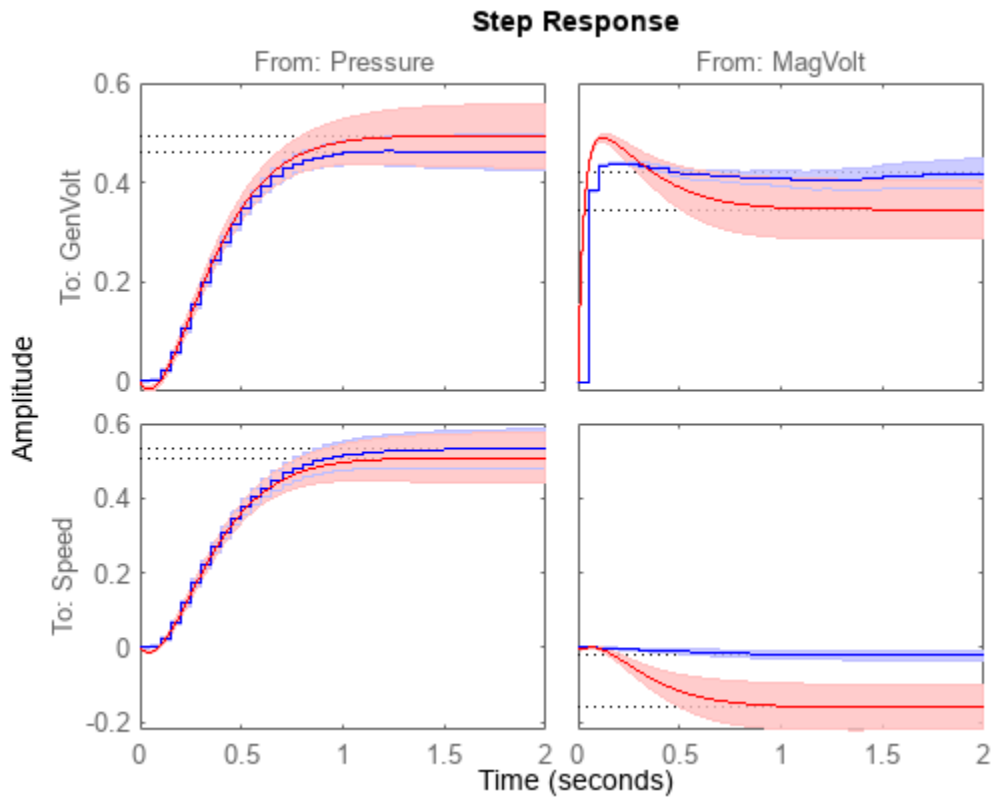
The agreement is good with the variation permissible within the shown confidence bounds.

To test the quality of the state-space model, simulate it on the part of data that was not used for estimation and compare the outputs:

```
compare(steam(251:450),mp)
```

Simulated Response Comparison
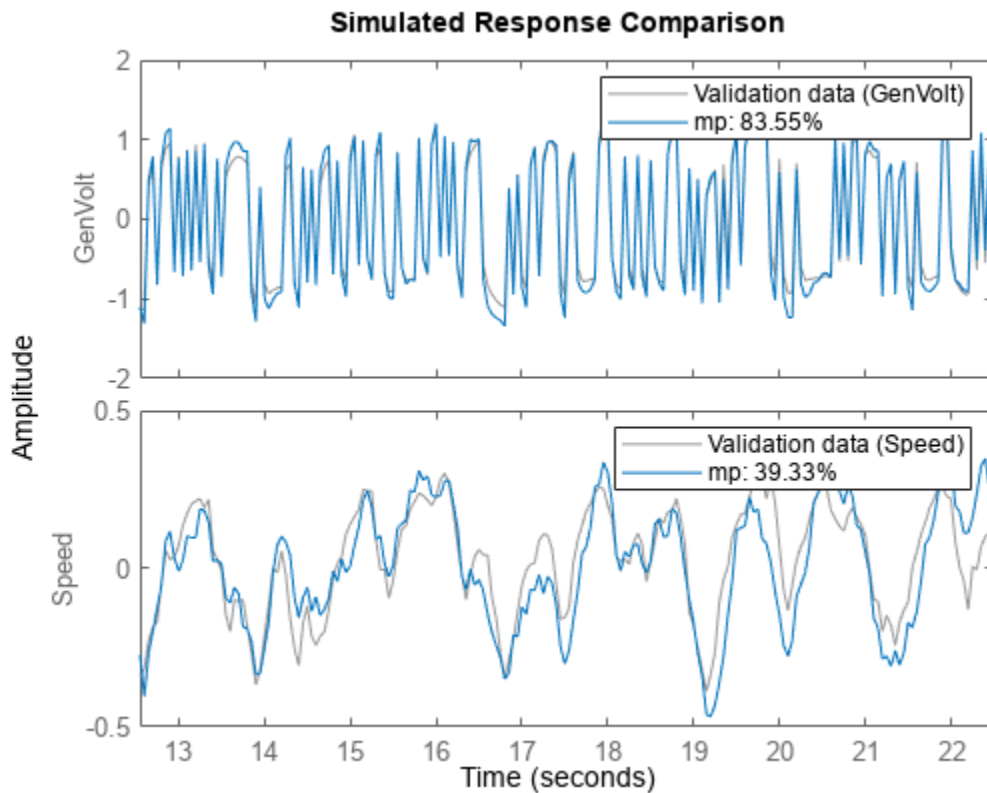
The model is very good at reproducing the Generated Voltage for the validation data, and does a reasonable job also for the speed. (Use the pull-down menu to see the fits for the different outputs.)

Spectral Analysis

Similarly, comparisons of the frequency response of `mp` with a spectral analysis estimate gives:

```
msp = spa(steam);
```

```
bode(msp,mp)
```

```
clf, bode(msp,'b',mp,'r')
```
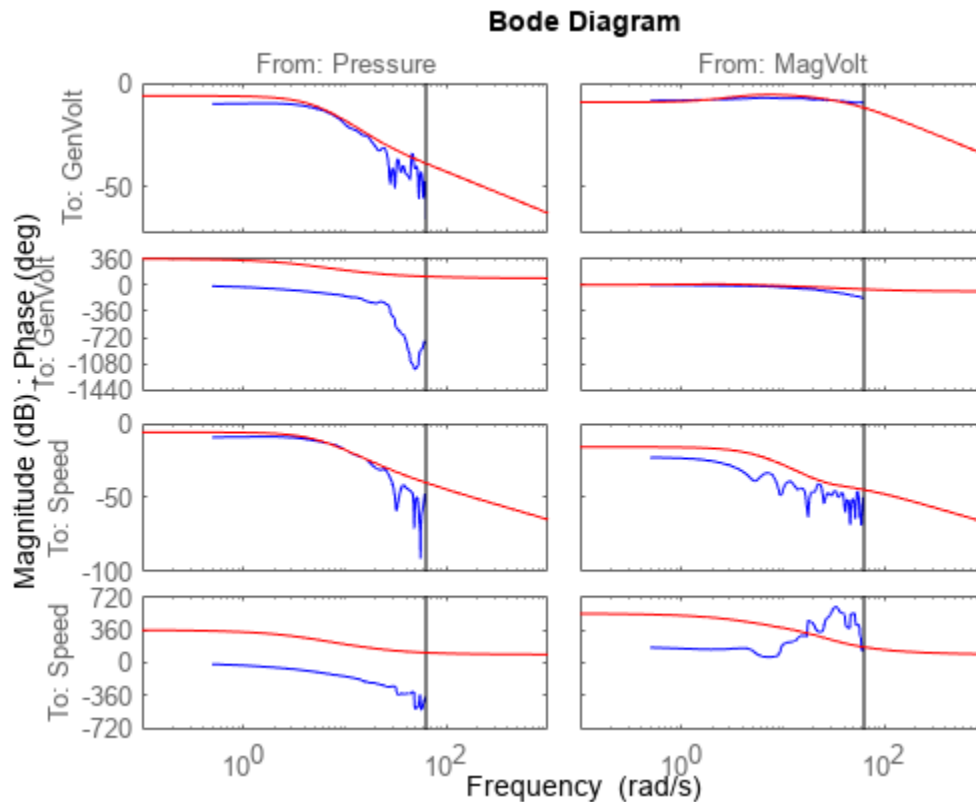
You can right-click on the plot and select the different I/O pairs for close looks. You can also choose 'Characteristics: Confidence Region' for a picture of the reliability of the bode plot.

As before the response from `MagVolt` to `Speed` is insignificant and difficult to estimate.

Single-Input-Single-Output (SISO) Models

This data set quickly gave good models. Otherwise you often have to try out sub-models for certain channels, to see significant influences The toolbox objects give full support to the necessary bookkeeping in such work. The input and output names are central for this.

The step responses indicate that `MagVolt` primarily influences `GenVolt` while `Pressure` primarily affects `Speed`. Build two simple SISO model for this: Both names and numbers can be used when selecting channels.

```
m1 = tfest(steam(1:250,'Speed','Pressure'),2,1); % TF model with 2 poles 1 zero
m2 = tfest(steam(1:250,1,2),1,0) % Simple TF model with 1 pole.

m2 =
  From input "MagVolt" to output "GenVolt":
    18.57
  ---------
  s + 43.53

Continuous-time identified transfer function.

Parameterization:
```

```
    Number of poles: 1   Number of zeros: 0
    Number of free coefficients: 2
    Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using TFEST on time domain data.
Fit to estimation data: 73.34%
FPE: 0.04645, MSE: 0.04535
```

Compare these models with the MIMO model mp:

```
compare(steam(251:450),m1,m2,mp)
```


Simulated Response Comparison
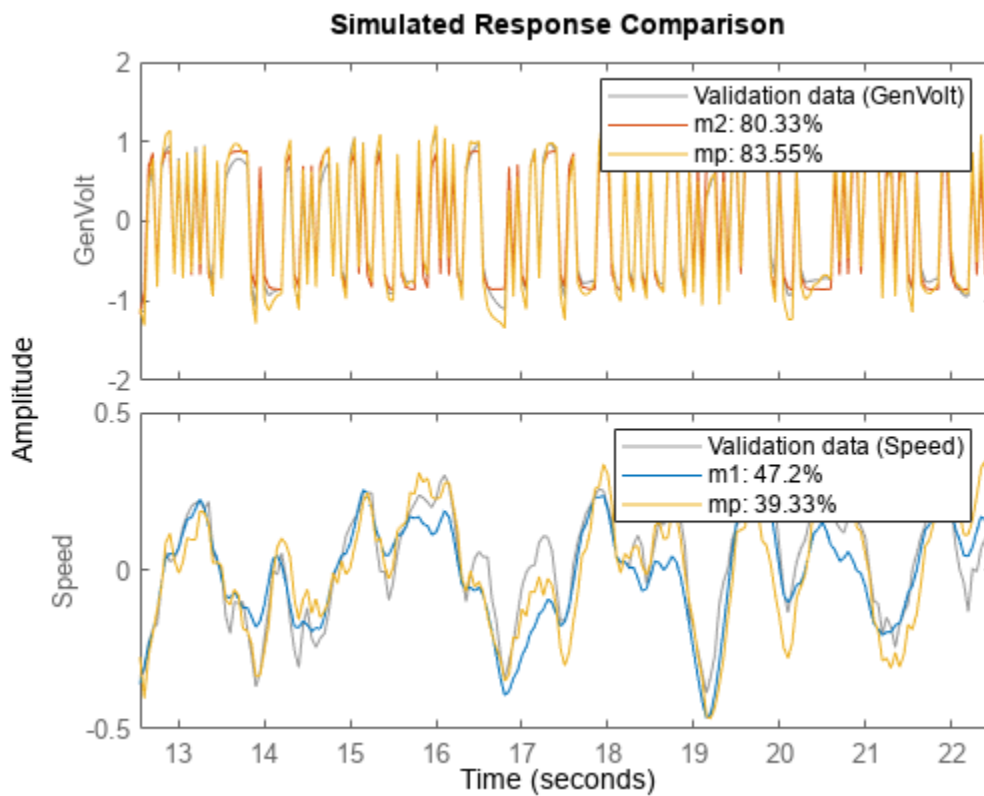
The SISO models compare well with the full model. Let us now compare the Nyquist plots. m1 is blue, m2 is green and mp is red. Note that the sorting is automatic. mp describes all input output pairs, while m1 only contains Pressure to Speed and m2 only contains MagVolt to GenVolt.

```
clf
showConfidence(nyquistplot(m1,'b',m2,'g',mp,'r'),3)
```
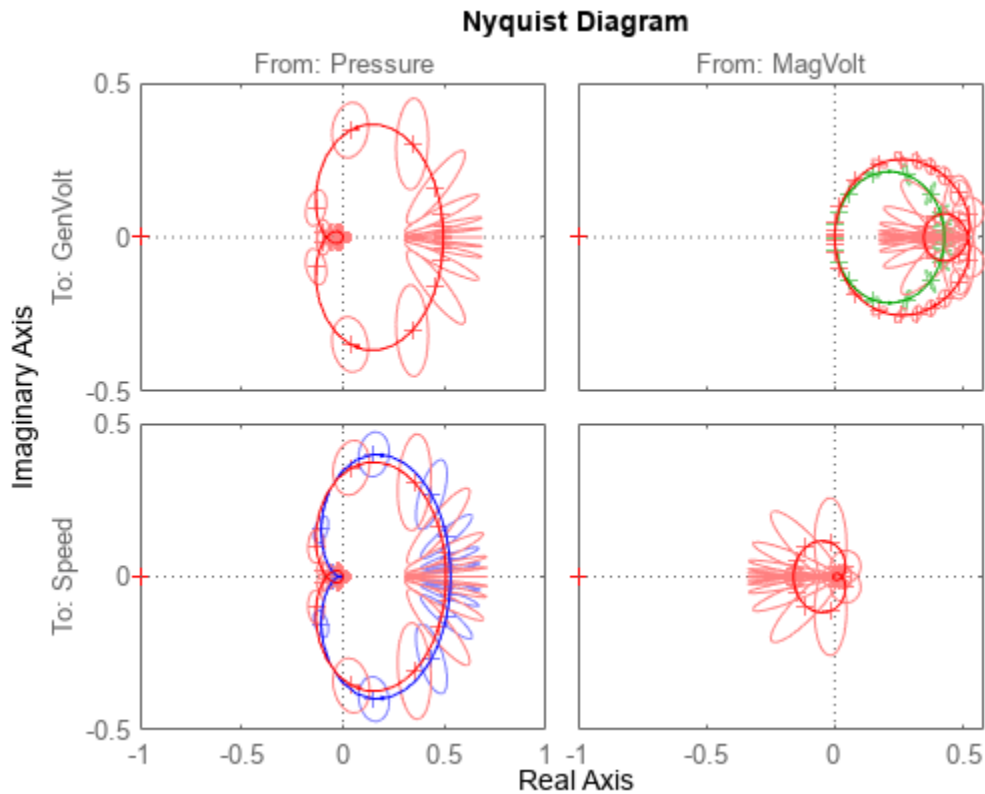
The SISO models do a good job to reproduce their respective outputs.

The rule-of-thumb is that the model fitting becomes harder when you add more outputs (more to explain!) and simpler when you add more inputs.

Two-Input-Single-Output Model

To do a good job on the output GenVolt, both inputs could be used.

```
m3 = armax(steam(1:250,'GenVolt',:),'na',4,'nb',[4 4],'nc',2,'nk',[1 1]);
m4 = tfest(steam(1:250,'GenVolt',:),2,1);
compare(steam(251:450),mp,m3,m4,m2)
```

Simulated Response Comparison
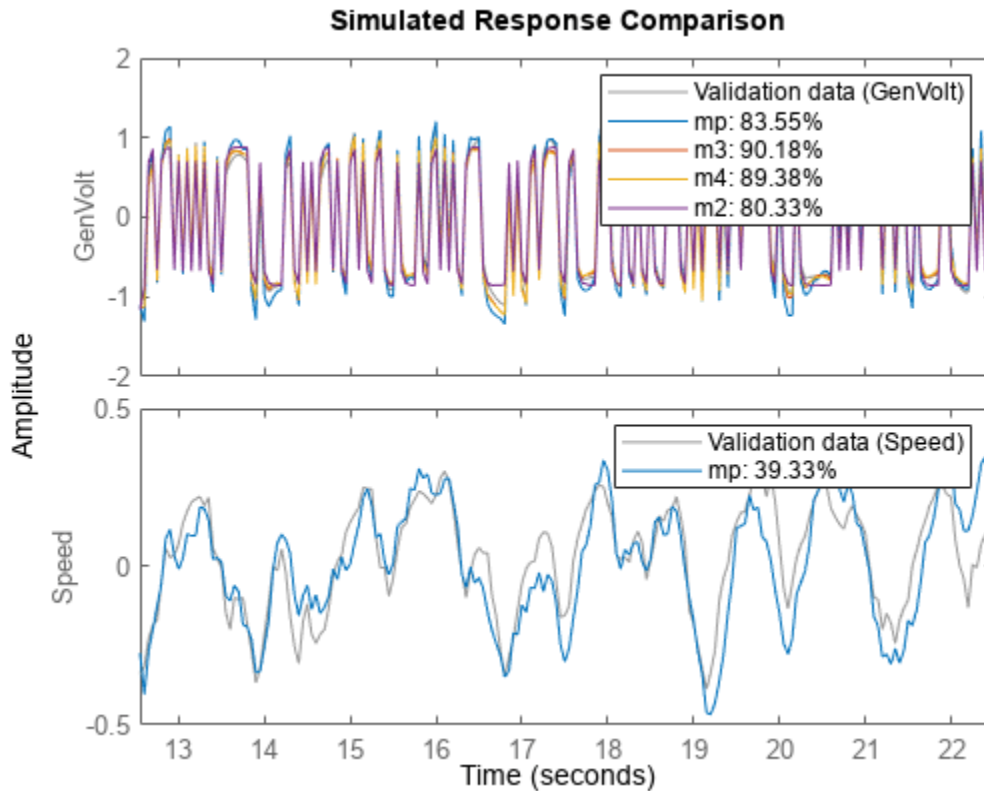
About 10% improvement was possible by including the input `Pressure` in the models `m3` (discrete time) and `m4` (continuous time), compared to `m2` that uses just `MagVolt` as input.

Merging SISO Models

If desired, the two SISO models `m1` and `m2` can be put together as one "Off-Diagonal" model by first creating a zero dummy model:

```
mdum = idss(zeros(2,2),zeros(2,2),zeros(2,2),zeros(2,2));
mdum.InputName = steam.InputName;
mdum.OutputName = steam.OutputName;
mdum.ts = 0; % Continuous time model
m12 = [idss(m1),mdum('Speed','MagVolt')];    % Adding Inputs.
                                             % From both inputs to Speed
m22 = [mdum('GenVolt','Pressure'),idss(m2)]; % Adding Inputs.
                                             % From both inputs to GenVolt

mm = [m12;m22]; % Adding the outputs to a 2-by-2 model.

compare(steam(251:450),mp,mm)
```
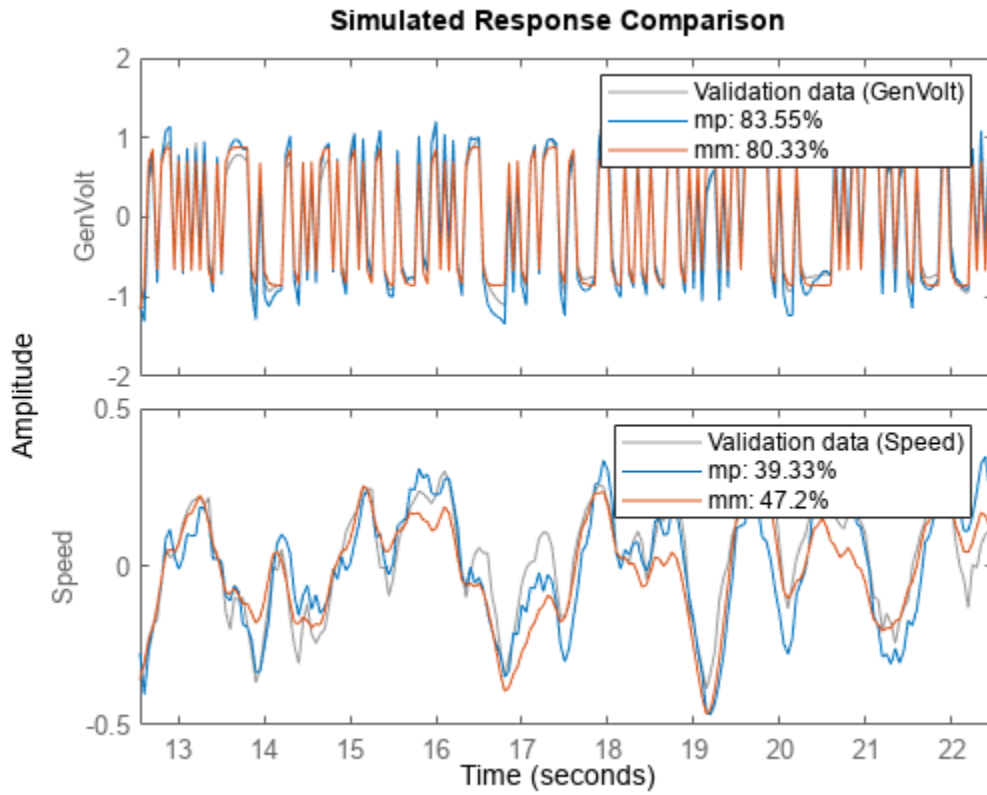
Clearly the "Off-Diagonal" model `mm` performs like `m1` and `m2` in explaining the outputs.

Glass Tube Manufacturing Process

This example shows linear model identification of a glass tube manufacturing process. The experiments and the data are discussed in:

V. Wertz, G. Bastin and M. Heet: Identification of a glass tube drawing bench. Proc. of the 10th IFAC Congress, Vol 10, pp 334-339 Paper number 14.5-5-2. Munich August 1987.

The output of the process is the thickness and the diameter of the manufactured tube. The inputs are the air-pressure inside the tube and the drawing speed.

The problem of modeling the process from the input speed to the output thickness is described below. Various options for analyzing data and determining model order are discussed.

Experimental Data

We begin by loading the input and output data, saved as an iddata object:

```
load thispe25.mat
```

The data are contained in the variable `glass`:

```
glass

glass =

Time domain data set with 2700 samples.
Sample time: 1 seconds

Outputs       Unit (if specified)
   Thickn

Inputs        Unit (if specified)
   Speed
```

Data has 2700 samples of one input (Speed) and one output (Thickn). The sample time is 1 sec.

For estimation and cross-validation purpose, split it into two halves:

```
ze = glass(1001:1500); %Estimation data
zv = glass(1501:2000,:); %Validation data
```

A close-up view of the estimation data:

```
plot(ze(101:200)) %Plot the estimation data range from samples 101 to 200.
```

Input-Output Data
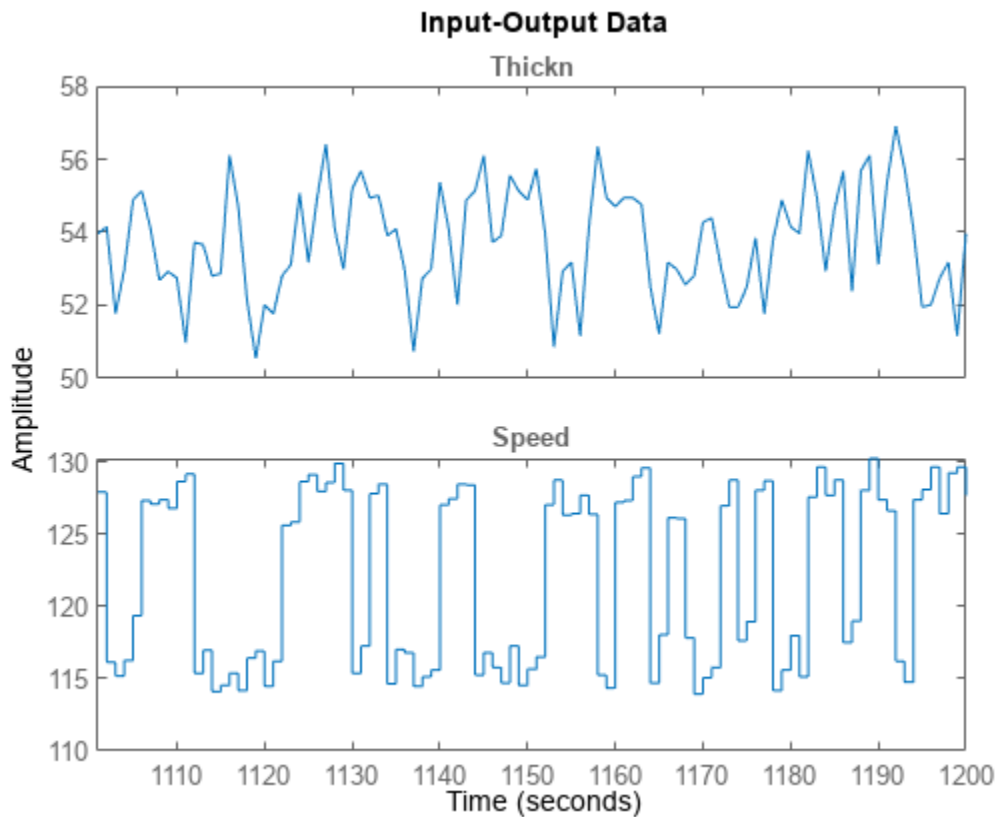
Preliminary Analysis of Data

Let us remove the mean values as a first preprocessing step:

```
ze = detrend(ze);
zv = detrend(zv);
```

The sample time of the data is 1 second, while the process time constants might be much slower. We may detect some rather high frequencies in the output. In order to affirm this, let us first compute the input and output spectra:

```
sy = spa(ze(:,1,[]));
su = spa(ze(:,[],1));
clf
spectrum(sy,su)
axis([0.024 10 -5 20])
legend({'Output','Input'})
grid on
```

4-191

Power Spectrum
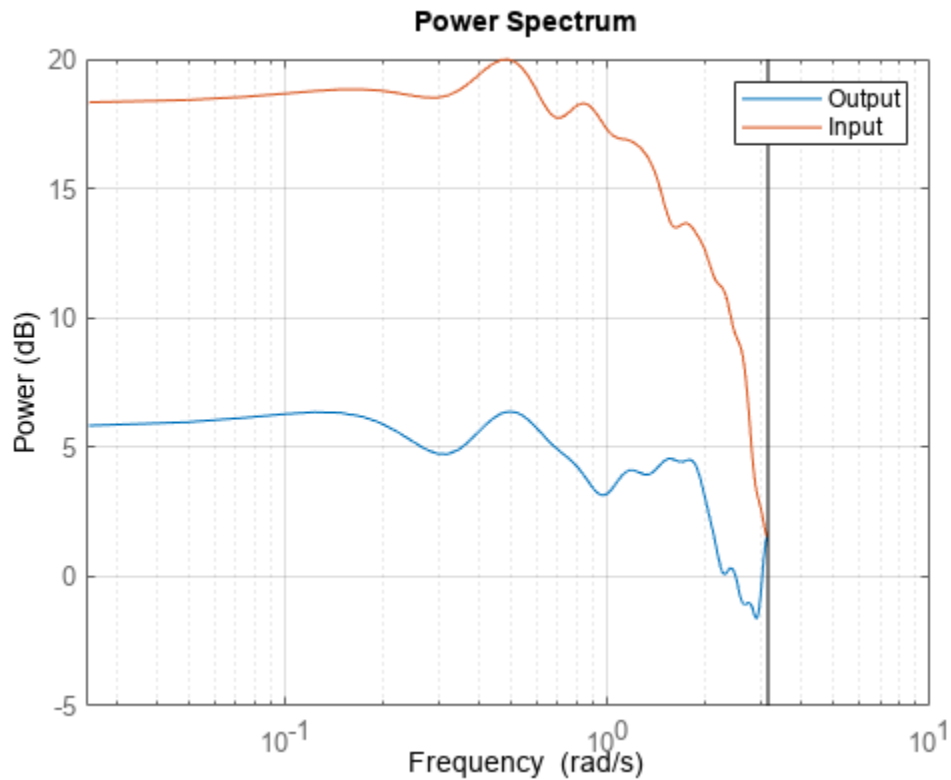
Note that the input has very little relative energy above 1 rad/sec while the output contains relatively larger values above that frequency. There are thus some high frequency disturbances that may cause some problem for the model building.

We compute the impulse response, using part of the data to gain some insight into potential feedback and delay from input to output:

```
Imp = impulseest(ze,[],'negative',impulseestOptions('RegularizationKernel','SE'));
showConfidence(impulseplot(Imp,-10:30),3)
grid on
```
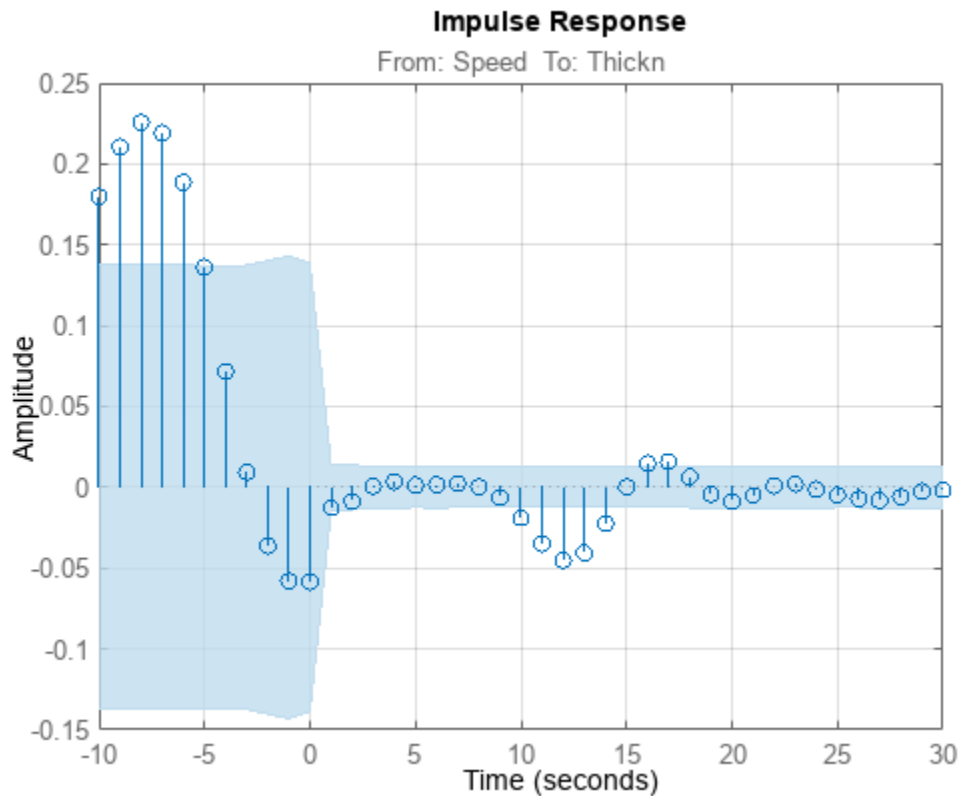
We see a delay of about 12 samples in the impulse response (first significant response value outside the confidence interval), which is quite substantial. Also, the impulse response is not insignificant for negative time lags. This indicates that there is a good probability of feedback in the data, so that future values of output influence (get added to) the current inputs. The input delay may be calculated explicitly using `delayest`:

```
delayest(ze)
```

```
ans = 12
```

The probability of feedback may be obtained using `feedback`:

```
feedback(ze) %compute probability of feedback in data
```

```
ans = 100
```

Thus, it is almost certain that there is feedback present in the data.

We also, as a preliminary test, compute the spectral analysis estimate:

```
g = spa(ze);
showConfidence(bodeplot(g))
grid on
```

4-193

Bode Diagram
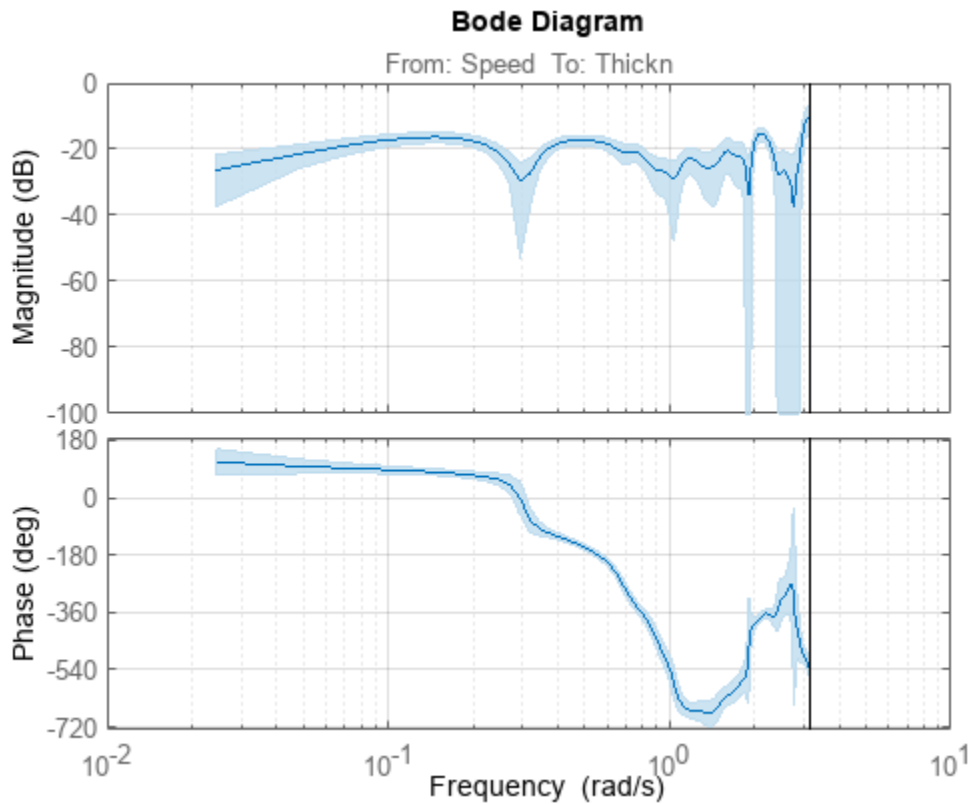
We note, among other things, that the high frequency behavior is quite uncertain. It may be advisable to limit the model range to frequencies lower than 1 rad/s.

Parametric Models of the Process Behavior

Let us do a quick check if we can pick up good dynamics by just computing a fourth order ARX model using the estimation data and simulate that model using the validation data. We know that the delay is about 12 seconds.

```
m1 = arx(ze,[4 4 12]);
compare(zv,m1);
```
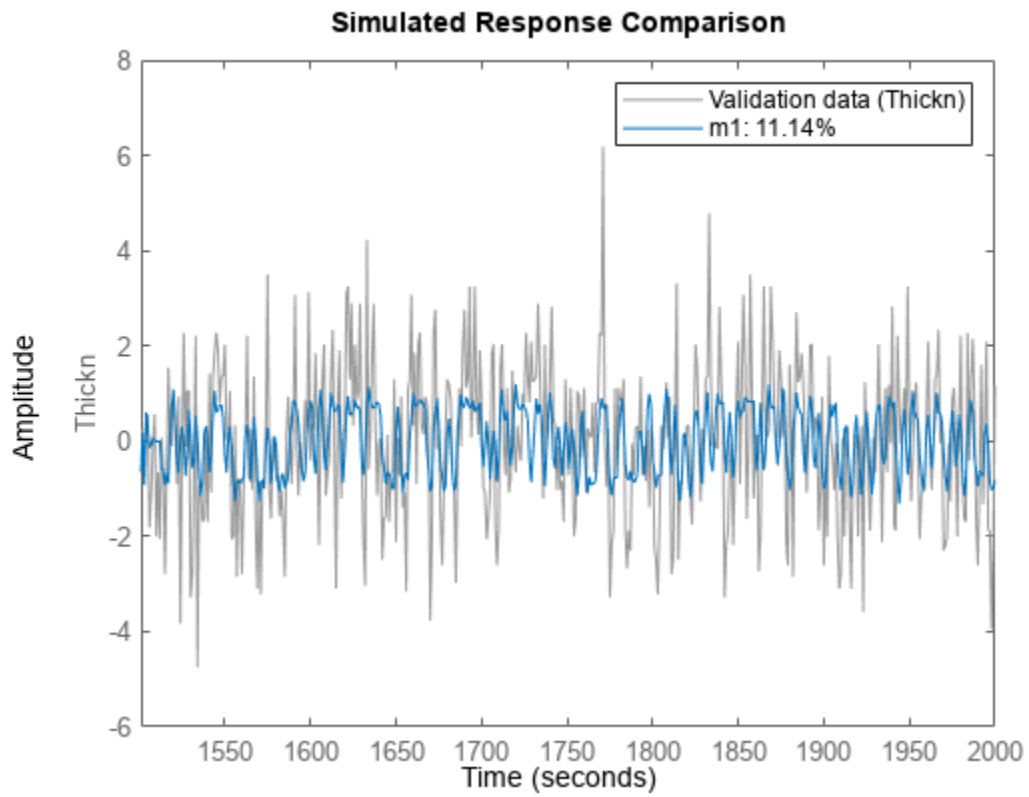
A close view of simulation results:

```
compare(zv,m1,inf,'Samples',101:200)
```
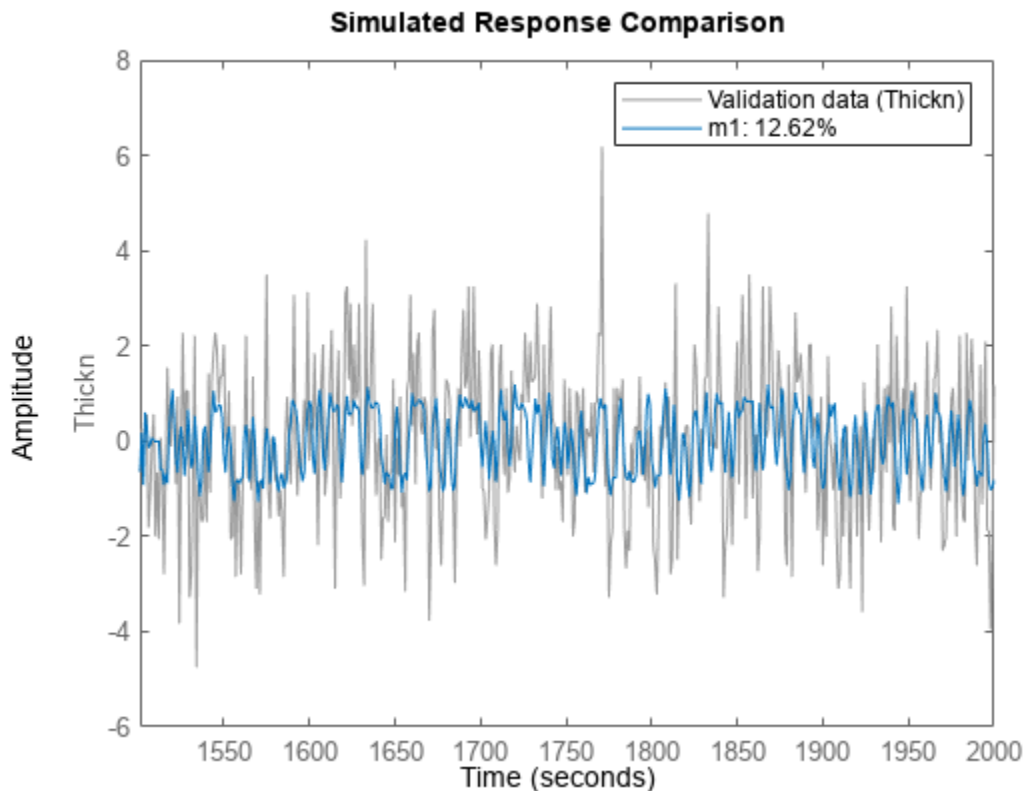
There are clear difficulties to deal with the high frequency components of the output. That, in conjunction with the long delay, suggests that we decimate the data by four (i.e. low-pass filter it and pick every fourth value):

```
if exist('resample','file')==2
    % Use "resample" command for decimation if Signal Processing Toolbox(TM)
    % is available.
    zd = resample(detrend(glass),1,4,20);
else
    % Otherwise, use the slower alternative - "idresamp"
    zd  = idresamp(detrend(glass),4);
end

zde = zd(1:500);
zdv = zd(501:size(zd,'N'));
```

Let us find a good structure for the decimated data. First compute the impulse response:

```
Imp = impulseest(zde);
showConfidence(impulseplot(Imp,200),3)
axis([0  100  -0.05  0.01])
grid on
```
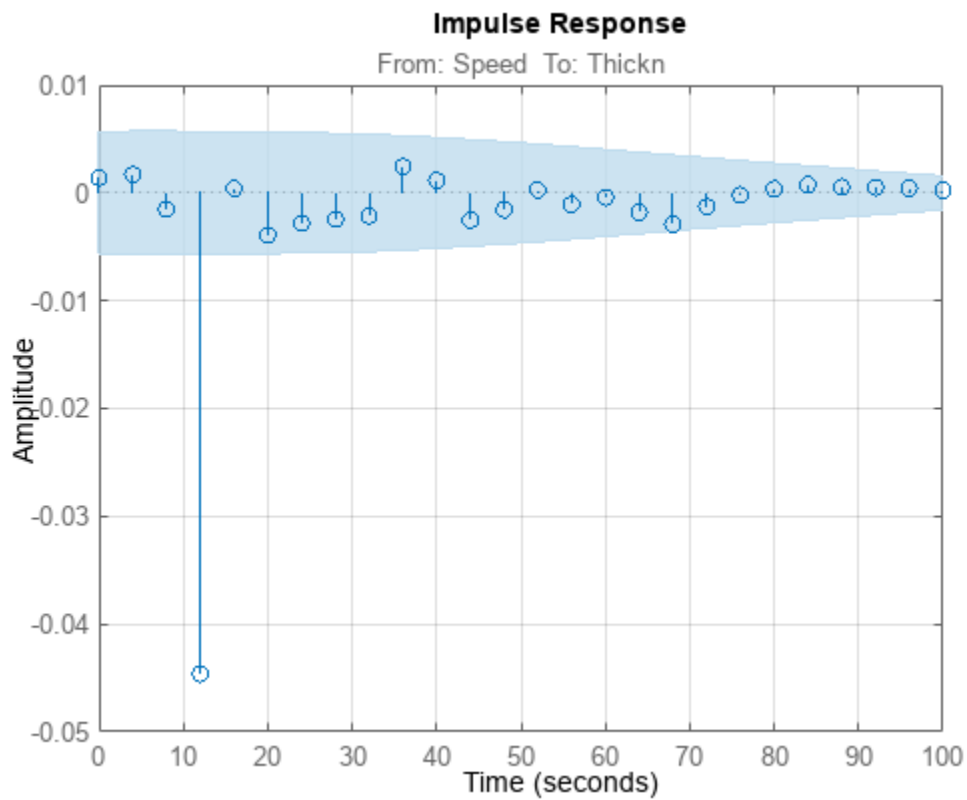
We again see that the delay is about 3 samples (which is consistent with what we saw above; 12 second delay with sample time of 4 seconds in zde). Let us now try estimating a default model, where the order is automatically picked by the estimator.

```
Mdefault = n4sid(zde);
compare(zdv,Mdefault)
```
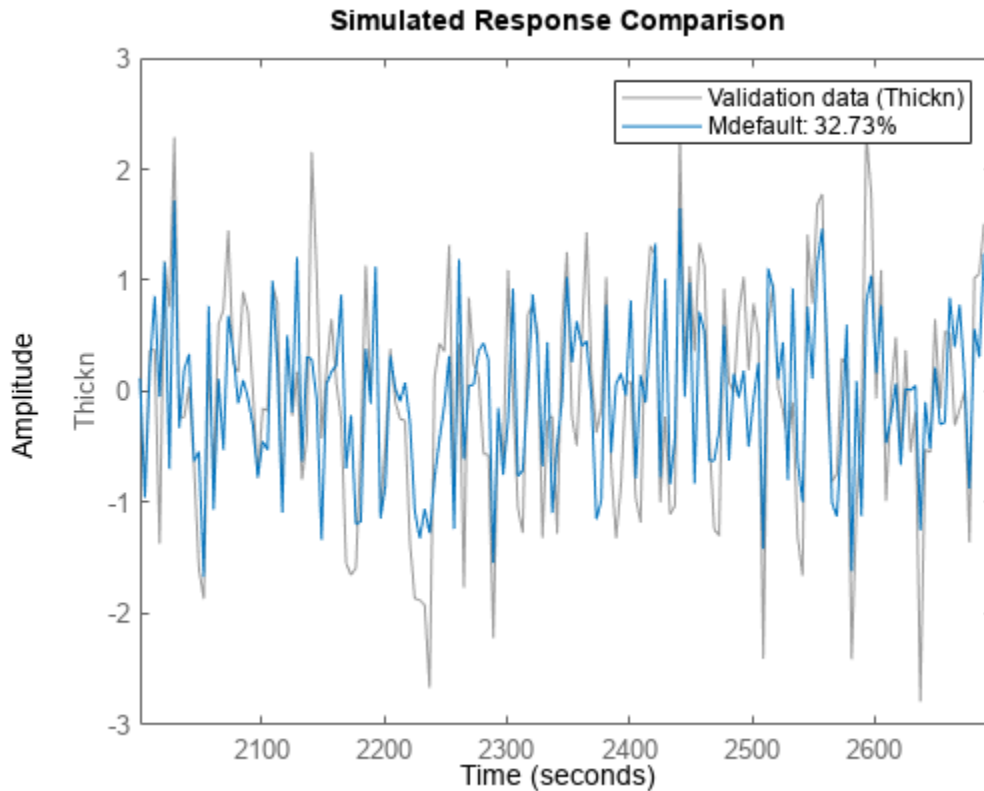
The estimator picked a 4th order model. It seems to provide a better fit than that for the undecimated data. Let us now systematically evaluate what model structure and orders we can use. First we look for the delay:

```
V = arxstruc(zde,zdv,struc(2,2,1:30));
nn = selstruc(V,0)
```

nn = *1×3*

        2   2   3

ARXSTRUC also suggests a delay of 3 samples which is consistent with the observations from the impulse response. Therefore, we fix the delay to the vicinity of 3 and test several different orders with and around this delay:

```
V = arxstruc(zde,zdv,struc(1:5,1:5,nn(3)-1:nn(3)+1));
```

Now we call `selstruc` on the returned matrix in order to pick the most preferred model order (minimum loss function, which is shown in the first row of V).

```
nn = selstruc(V,0); %choose the "best" model order
```

SELSTRUC could be called with just one input to invoke an interactive mode of order selection (`nn = selstruc(V)`).

Let us compute and check the model for the "best" order returned in variable `nn`:

```
m2 = arx(zde,nn);
compare(zdv,m2,inf,compareOptions('Samples',21:150));
```

Simulated Response Comparison
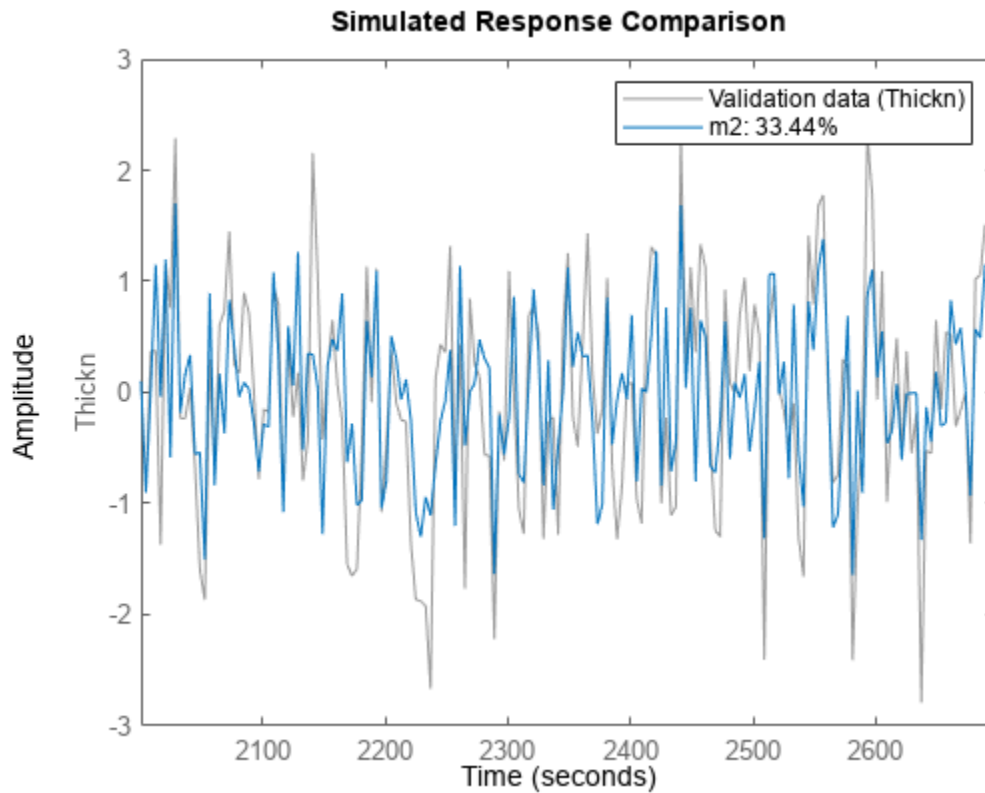
The model `m2` is about same as `Mdefault` is fitting the data but uses lower orders.

Let us test the residuals:

```
resid(zdv,m2);
```
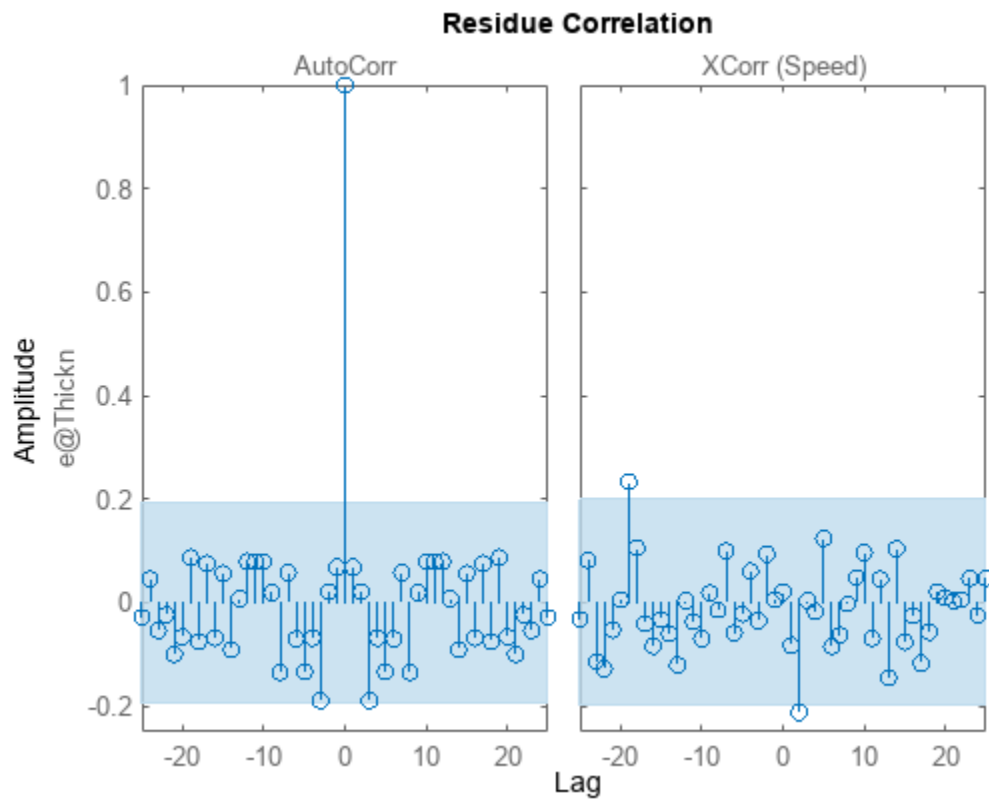
The residuals are inside the confidence interval region, indicating that the essential dynamics have been captured by the model. What does the pole-zero diagram tell us?

```
clf
showConfidence(iopzplot(m2),3)
axis([ -1.1898,1.3778,-1.5112,1.5688])
```
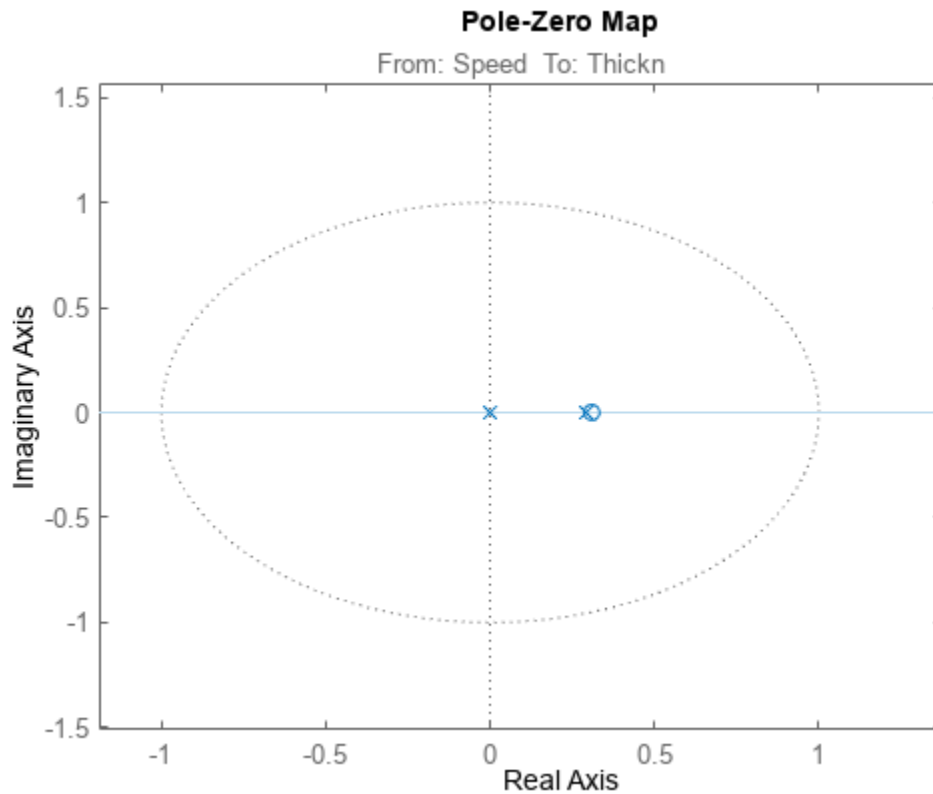
From the pole-zero plot, there is am indication of pole-zero cancellations for several pairs. This is because their locations overlap, within the confidence regions. This shows that we should be able to do well with lower order models. Try a [1 1 3] ARX model:

```
m3 = arx(zde,[1 1 3])

m3 =
Discrete-time ARX model: A(z)y(t) = B(z)u(t) + e(t)
  A(z) = 1 - 0.115 z^-1

  B(z) = -0.1788 z^-3

Sample time: 4 seconds

Parameterization:
   Polynomial orders:    na=1    nb=1    nk=3
   Number of free coefficients: 2
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using ARX on time domain data "zde".
Fit to estimation data: 35.07% (prediction focus)
FPE: 0.4437, MSE: 0.4384
```

Simulation of model m3 compared against the validation data shows:

```
compare(zdv,Mdefault,m2,m3)
```

4-201
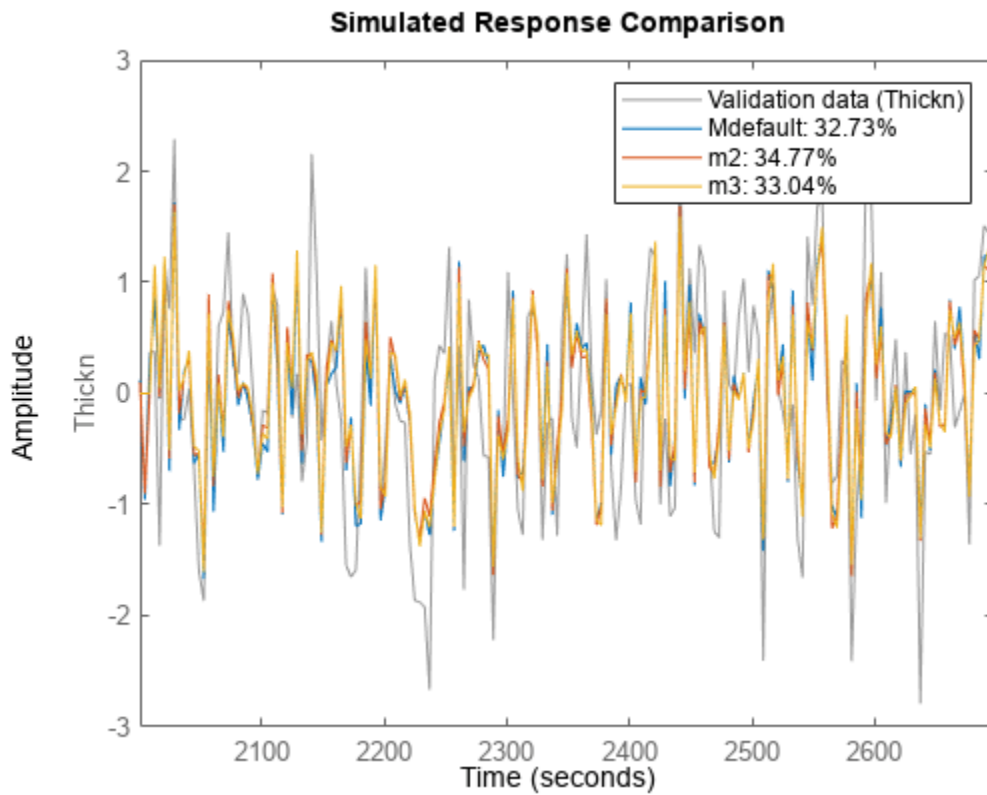
The three models deliver comparable results. Similarly, we can compare the 5-step ahead prediction capability of the models:

```
compare(zdv,Mdefault,m2,m3,5)
```

5-Step Predicted Response Comparison
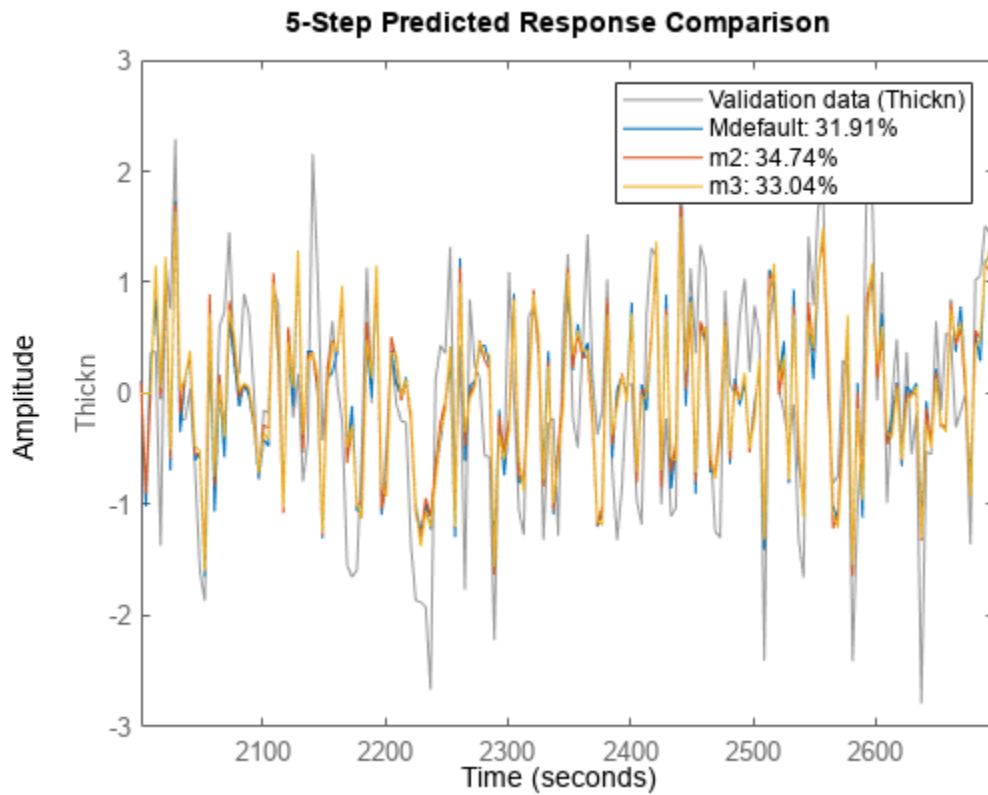
As these plots indicate, a reduction in model order does not significantly reduce its effectiveness is predicting future values.

Modal Analysis of a Flexible Flying Wing Aircraft

This example shows computation of bending modes of a flexible wing aircraft. The vibration response of the wing is collected at multiple points along its span. The data is used to identify a dynamic model of the system. The modal parameters are extracted from the identified model. The modal parameter data is combined with the sensor position information to visualize the various bending modes of the wing. This example requires Signal Processing Toolbox™.

The Flexible Wing Aircraft

In this example, we study the data gathered from a small flexible flying wing aircraft built at the Uninhabited Aerial Vehicle Laboratories, University of Minnesota [1]. The geometry of the aircraft is shown below.
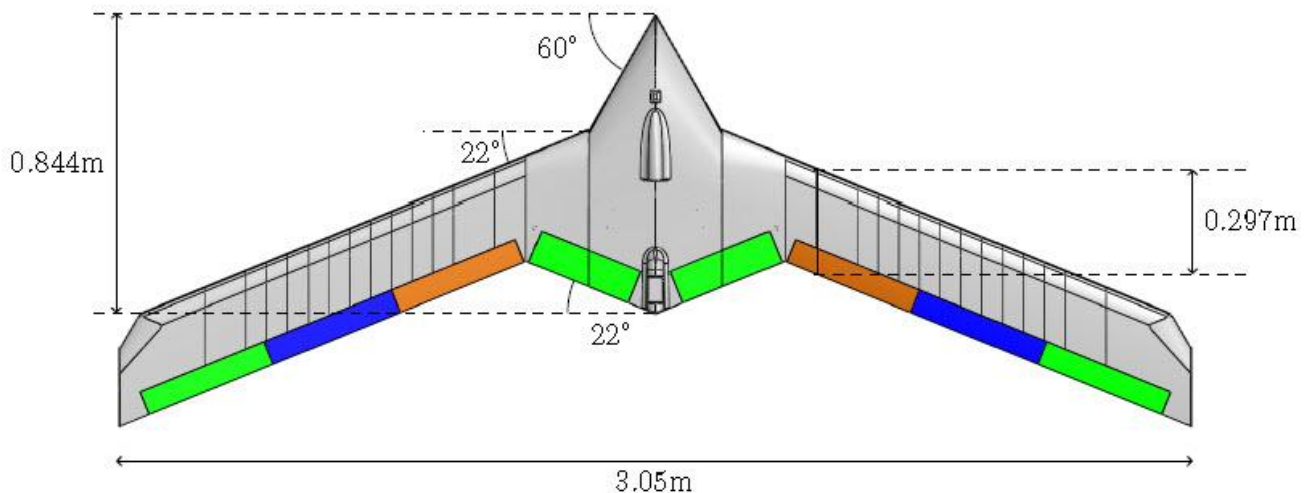
The aircraft wing can undergo large deformations under loading. The flexible mode frequencies are lower than those in common aircraft with more rigid wings. This flexible design reduces material costs, increases agility and the flight range of the aircraft. However, if not controlled, the flexible modes can lead to catastrophic aeroelastic instabilities (flutter). Designing effective control laws for suppressing these instabilities requires accurate determination of the wing's various bending modes.

Experimental Setup

The objective of the experiment is to gather vibration response of the aircraft at various locations in response to an external excitation. The aircraft is suspended from a wooden frame using a single spring at its center of gravity. The spring is sufficiently flexible so that the natural frequency of the spring-mass oscillation does not interfere with the fundamental frequencies of the aircraft. An input force is applied via an Unholtz-Dickie Model 20 electrodynamic shaker near the center of the aircraft.
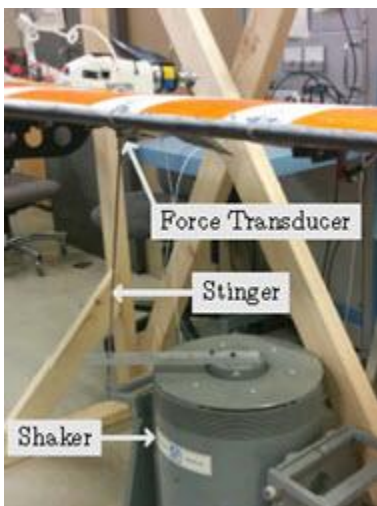
Twenty PCB-353B16 accelerometers are placed along the wing span to collect the vibration response as shown in the next figure.
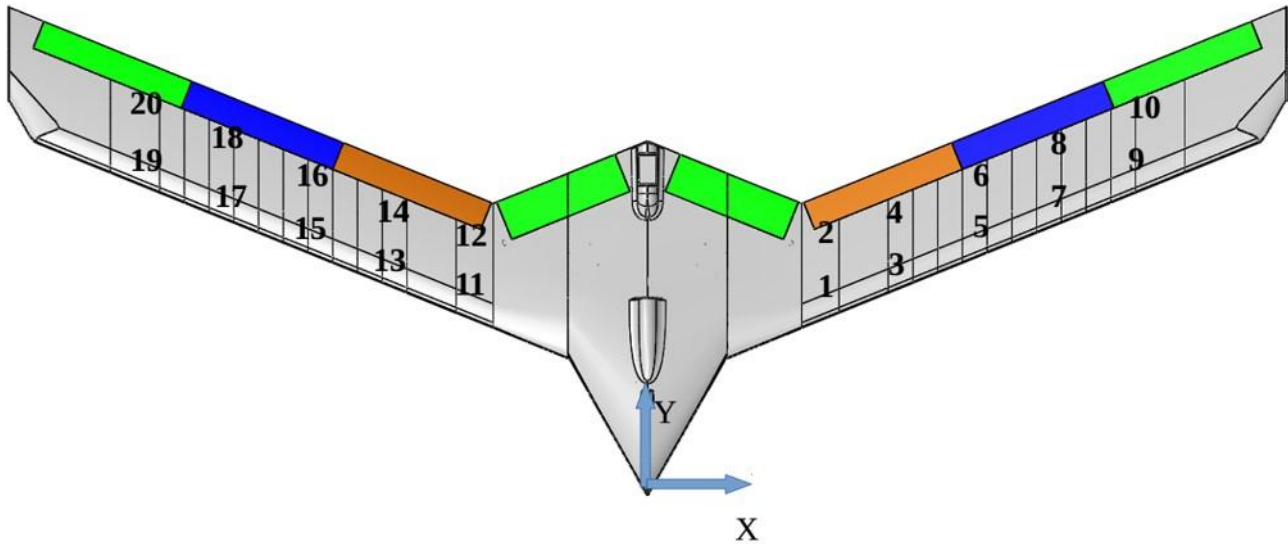
The shaker input command is specified as a constant amplitude chirp input of the form $A\sin(\omega(t)t)$. The chirp frequency varies linearly with time, that is, $\omega(t) = c_0 + c_1 t$. The frequency range covered by the input signal is 3–35 Hz. The data is collected by two accelerometers (leading and trailing edge accelerometers at one x-location) at a time. Hence 10 experiments are conducted to collect all the 20 accelerometer responses. The accelerometer and force transducer measurements are all sampled at 2000 Hz.

Data Preparation

The data is represented by 10 sets of two-output/one-input signals, each containing 600K samples. The data is available at the MathWorks support files site. See the disclaimer. Download the data file and load the data into the MATLAB® workspace.

```
url = 'https://www.mathworks.com/supportfiles/ident/flexible-wing-GVT-data/FlexibleWingData.mat'
websave('FlexibleWingData.mat',url);
load FlexibleWingData.mat MeasuredData
```

The variable `MeasuredData` is a structure with fields `Exp1`, `Exp2`, …, `Exp10`. Each field is a structure with fields `y` and `u` containing the two responses and the corresponding input force data. Plot the data for the first experiment.

```
fs = 2000;                 % data sampling frequency
Ts = 1/fs;                 % sample time
y = MeasuredData.Exp1.y;   % output data (2 columns, one for each accelerometer)
u = MeasuredData.Exp1.u;   % input force data
t = (0:length(u)-1)' * Ts;
figure
subplot(211)
plot(t,y)
ylabel('Outputs (m/s^2)')
legend('Leading edge','Trailing edge')
subplot(212)
plot(t,u)
ylabel('Input')
xlabel('Time (seconds)')
```
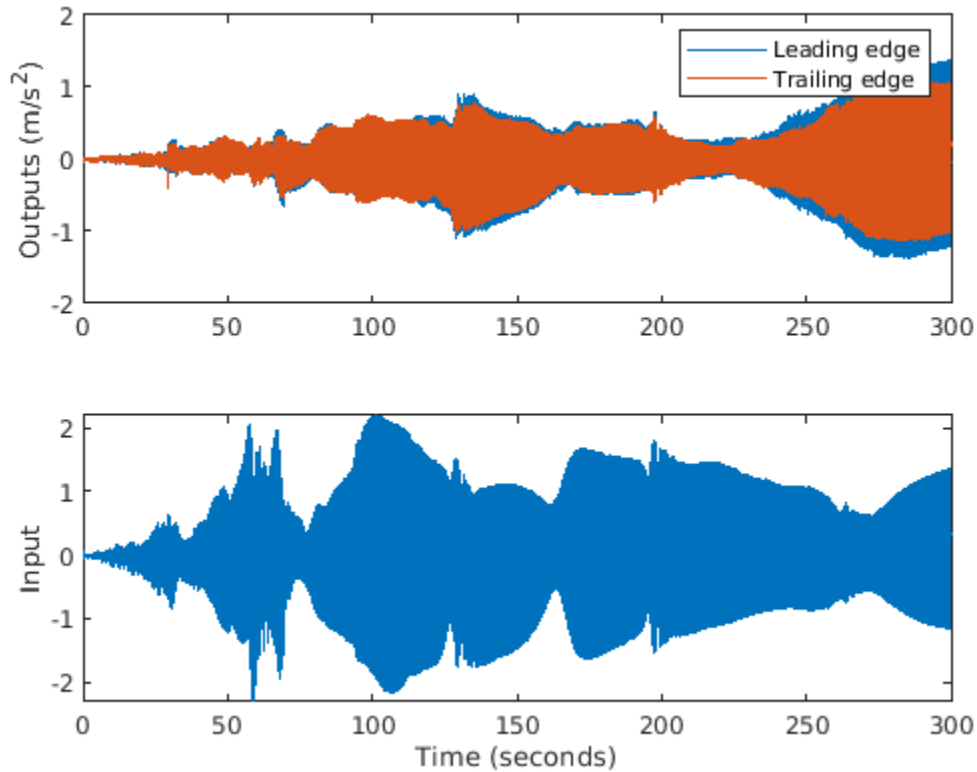
In order to prepare data for model identification, the data is packaged into `iddata` objects. The `iddata` objects are standard way of packaging time-domain data in System Identification Toolbox™. The input signal is treated as bandlimited.

```
ExpNames = fieldnames(MeasuredData);
Data = cell(1, 10);
for k = 1:10
    y =  MeasuredData.(ExpNames{k}).y;
    u =  MeasuredData.(ExpNames{k}).u;
    Data{k} = iddata(y, u, Ts, 'InterSample', 'bl');
end
```

Merge the dataset objects into one multi-experiment data object.

```
Data = merge(Data{:})
```

```
Data =
Time domain data set containing 10 experiments.

Experiment    Samples         Sample Time
   Exp1       600001               0.0005
   Exp2       600001               0.0005
   Exp3       600001               0.0005
   Exp4       600001               0.0005
   Exp5       600001               0.0005
   Exp6       600001               0.0005
   Exp7       600001               0.0005
   Exp8       600001               0.0005
```

4-207

```
    Exp9       600001              0.0005
    Exp10      600001              0.0005

Outputs       Unit (if specified)
   y1
   y2

Inputs        Unit (if specified)
   u1
```

Model Identification

We want to identify a dynamic model whose frequency response matches that of the actual aircraft as closely as possible. A dynamic model encapsulates a mathematical relationship between the inputs and outputs of the system as a differential or difference equation. Examples of dynamic models are transfer functions and state-space models. In System Identification Toolbox, dynamic models are encapsulated by objects such as `idtf` (for transfer functions), `idpoly` (for AR, ARMA models) and `idss` (for state-space models). Dynamic models can be created by running estimation commands such as `tfest` and `ssest` commands on measured data in either time-domain or frequency-domain.

For this example, we first convert the measured time-domain data into frequency response data by empirical transfer function estimation using the `etfe` command. The estimated FRF is then used to identify a state-space model of the aircraft's vibration response. It is possible to directly use time-domain data for dynamic model identification. However, FRF form of data allows compression of large datasets into fewer samples as well as more easily adjust estimation behavior to relevant frequency ranges. FRFs are encapsulated by `idfrd` objects.

Estimate a two-output/one-input frequency response function (FRF) for each data experiment. Use no windowing. Use 24,000 frequency points for response computation.

```
G = cell(1, 10);
N = 24000;
for k = 1:10
    % Convert time-domain data into a FRF using ETFE command
    Data_k = getexp(Data, k);
    G{k} = etfe(Data_k, [], N); % G{k} is an @idfrd object
end
```

Concatenate all FRFs into a single 20-output/one-input FRF.

```
G = cat(1, G{:});        % concatenate outputs of all estimated FRFs
G.OutputName = 'y';      % name outputs 'y1', 'y2', ..., 'y20'
G.InterSample = 'bl';
```

To get a feel for the estimated frequency response, plot the amplitude for outputs 1 and 15 (picked arbitrarily). Zoom into the frequency range of interest (4–35 Hz).

```
opt = bodeoptions;              % plot options
opt.FreqUnits = 'Hz';           % show frequency in Hz
opt.PhaseVisible = 'off';       % do not show phase
OutputNum = [1 15];             % pick outputs 1 and 15 for plotting
clf
bodeplot(G(OutputNum, :), opt)   % plot frequency response
xlim([4 35])
grid on
```

Bode Diagram
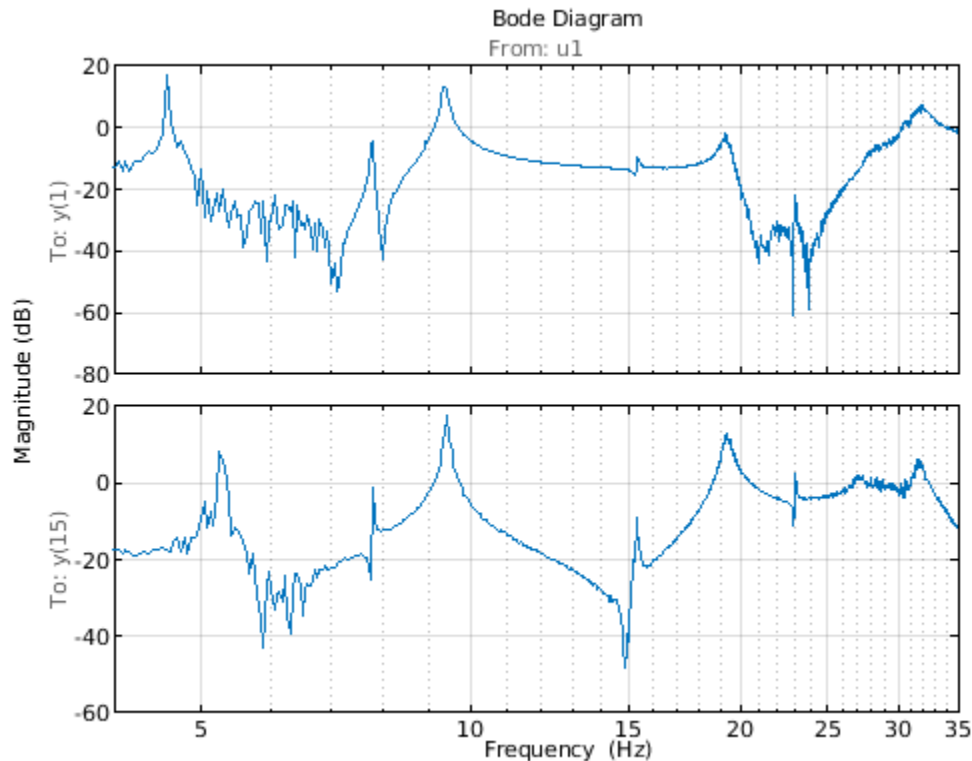
The FRF shows at least 9 resonant frequencies. For analysis we want to focus on 6-35 Hz frequency span where the most critical flexible bending modes of the aircraft lie. Hence reduce the FRF to this frequency region.

```
f = G.Frequency/2/pi;              % extract frequency vector in Hz (G stores frequency in rad/s)
Gs = fselect(G, f>6 & f<=32)       % "fselect" selects the FRF in the requested range (6.5 - 35 Hz)

Gs =
IDFRD model.
Contains Frequency Response Data for 20 output(s) and 1 input(s).
Response data is available at 624 frequency points, ranging from 37.96 rad/s to 201.1 rad/s.

Sample time: 0.0005 seconds
Output channels: 'y(1)', 'y(2)', 'y(3)', 'y(4)', 'y(5)', 'y(6)', 'y(7)', 'y(8)', 'y(9)', 'y(10)'
Input channels: 'u1'
Status:
Estimated model
```

Gs thus contains the frequency response measurements at the 20 measurement locations. Next, identify a state-space model to fit Gs. The subspace estimator n4sid provides a quick noniterative estimate. The state-space model structure is configured as follows:

1   We estimate an 24th-order continuous-time model. The order was found after some trials with various orders and checking the resulting fit of the model to the FRF.

2   The model contains a feedthrough term (*D* matrix is non-zero). This is because we are limiting our analysis to ≤ 35 Hz while the wing's bandwidth is significantly larger than that (response is significant at 35 Hz).

3   To speed up computation, we suppress computation of parameter covariance.

4   The FRF magnitude varies significantly across the frequency range. In order to ensure that the low amplitudes receive equal emphasis as the higher amplitudes, we apply a custom weighting that varies inversely as the square root of the response. Since there are 20 channels, the weighting vector is taken as the mean of the weights obtained from each (this works because the FRFs of the 20 channels are similar in shape).

We set up the estimation options for `n4sid` using `n4sidOptions`.

```
FRF = squeeze(Gs.ResponseData);
Weighting = mean(1./sqrt(abs(FRF))).';
n4Opt = n4sidOptions('EstimateCovariance',false,...
    'WeightingFilter',Weighting,...
    'OutputWeight',eye(20));
sys1 = n4sid(Gs,24,'Ts',0,'Feedthrough',true,n4Opt);
Fit = sys1.Report.Fit.FitPercent'
```

```
Fit = 1×20

   54.2689   55.3738   84.0188   85.7017   81.9259   80.2293   55.7448   40.6773   58.6429   76.2
```

The "Fit" numbers shows the percentage fit between the data (`Gs`) and model's (`sys1`) frequency response using a normalized root-mean-square-error (NRMSE) goodness-of-fit measure. The poorest and best fits are plotted next.

```
[~,Imin] = min(Fit);
[~,Imax] = max(Fit);
clf
bodeplot(Gs([Imin, Imax],:), sys1([Imin, Imax],:), opt);
xlim([6 32])
title('Worst (top) and best (bottom) fits between data and model')
grid on
legend('Data', 'Model')
```
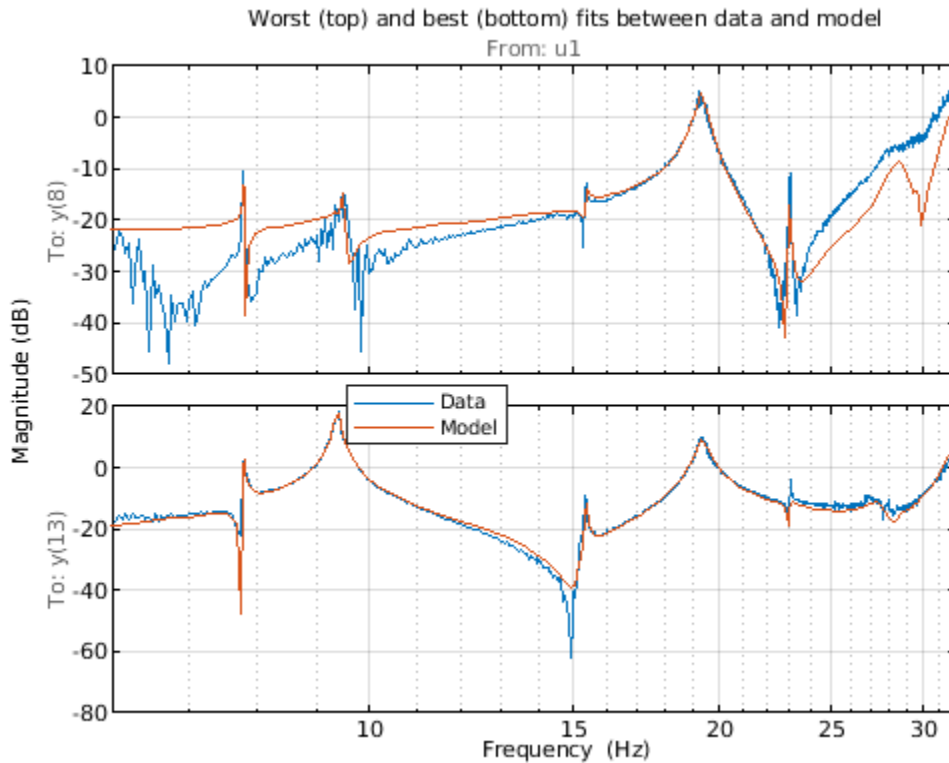
The fits achieved with model `sys1` can be improved even more by iterative nonlinear least-squares refinement of model's parameters. This can be achieved using the `ssest` command. We set up the estimation options for `ssest` using the `ssestOptions` command. This time the parameter covariance is also estimated.

```
ssOpt = ssestOptions('EstimateCovariance',true,...
   'WeightingFilter',n4Opt.WeightingFilter,...
   'SearchMethod','lm',...      % use Levenberg-Marquardt search method
   'Display','on',...
   'OutputWeight',eye(20));
sys2 = ssest(Gs, sys1, ssOpt);  % estimate state-space model (this takes several minutes)
Fit = sys2.Report.Fit.FitPercent'
```

Fit = *1×20*

    88.8715   88.6587   88.9388   89.8630   87.8597   88.1235   79.7016   82.5930   74.2458   82.0

As before we plot the worst and the best fits. We also visualize the 1-sd confidence region for the model's frequency response.

```
[~, Imin] = min(Fit);
[~, Imax] = max(Fit);
clf
h = bodeplot(Gs([Imin, Imax],:), sys2([Imin, Imax],:), opt);
showConfidence(h, 1)
xlim([6.5 35])
```

4-211

```
title('Worst (top) and best (bottom) fits between data and refined model')
grid on
legend('Data', 'Model (sys2)')
```
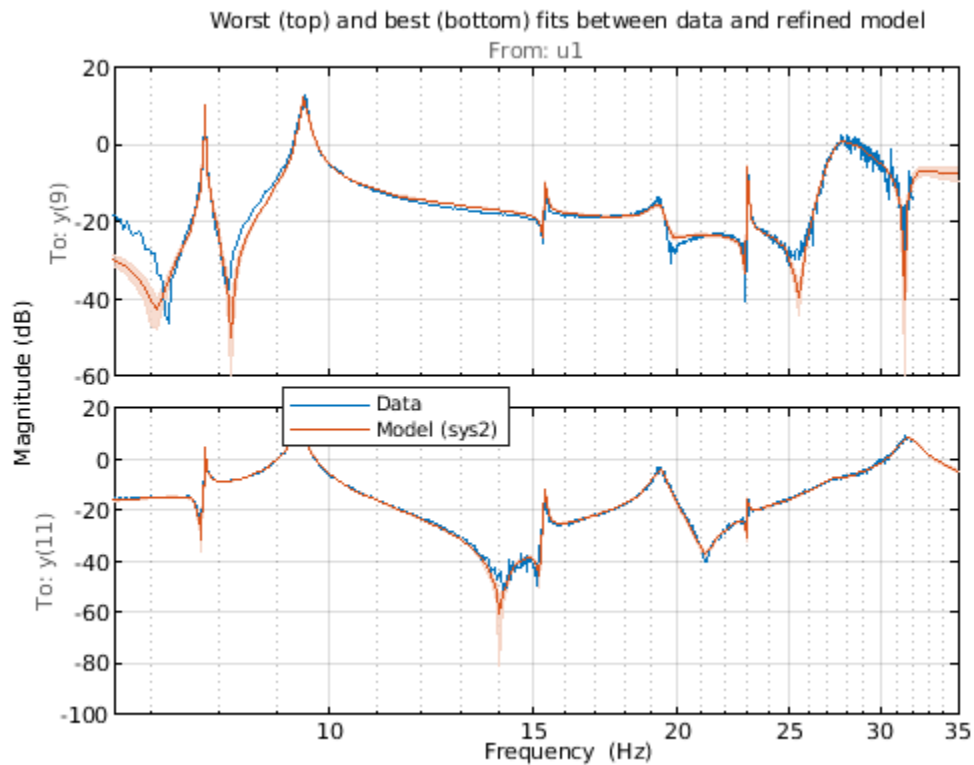

The refined state-space model `sys2` fits the FRFs quite well in the 7–20 Hz region. The uncertainty bounds are tight around most resonant locations. We estimated a 24th-order model which means that there are at most 12 oscillatory modes in the system `sys2`. Use the `modalfit` command to fetch the natural frequencies in Hz for these modes.

```
f = Gs.Frequency/2/pi;      % data frequencies (Hz)
fn = modalfit(sys2, f, 12); % natural frequencies (Hz)
disp(fn)

    7.7720
    7.7954
    9.3176
    9.4052
    9.4919
   15.3462
   19.3281
   23.0219
   27.4035
   28.6873
   31.7036
   64.2624
```

The values in `fn` suggest two frequencies very close to 7.8 Hz and three close to 9.4 Hz. An inspection of frequency responses near these frequencies reveals that the peaks location shift a little

bit across outputs. These discrepancies may be removed by better control over the experiment process, performing direct time-domain identification with input bandwidth limited to narrow range centered at these frequencies, or fitting a single oscillatory mode to the frequency response around these frequencies. These alternatives are not explored in this example.

Modal Parameter Computation

We can now use the model `sys2` to extract the modal parameters. An inspection of the FRFs indicates around 10 modal frequencies, roughly around the frequencies [5 7 10 15 17 23 27 30] Hz. We can make this assessment more concrete by using the `modalsd` command that checks the stability of modal parameters as the order of the underlying model is changed. This operation takes a long time. Hence the resulting plot is inserted directly as an image. Execute the code inside the comment block below to reproduce the figure.

```
FRF = permute(Gs.ResponseData,[3 1 2]);
f = Gs.Frequency/2/pi;
%{
 figure
 pf = modalsd(FRF,f,fs,'MaxModes',20,'FitMethod','lsrf');
%}
```
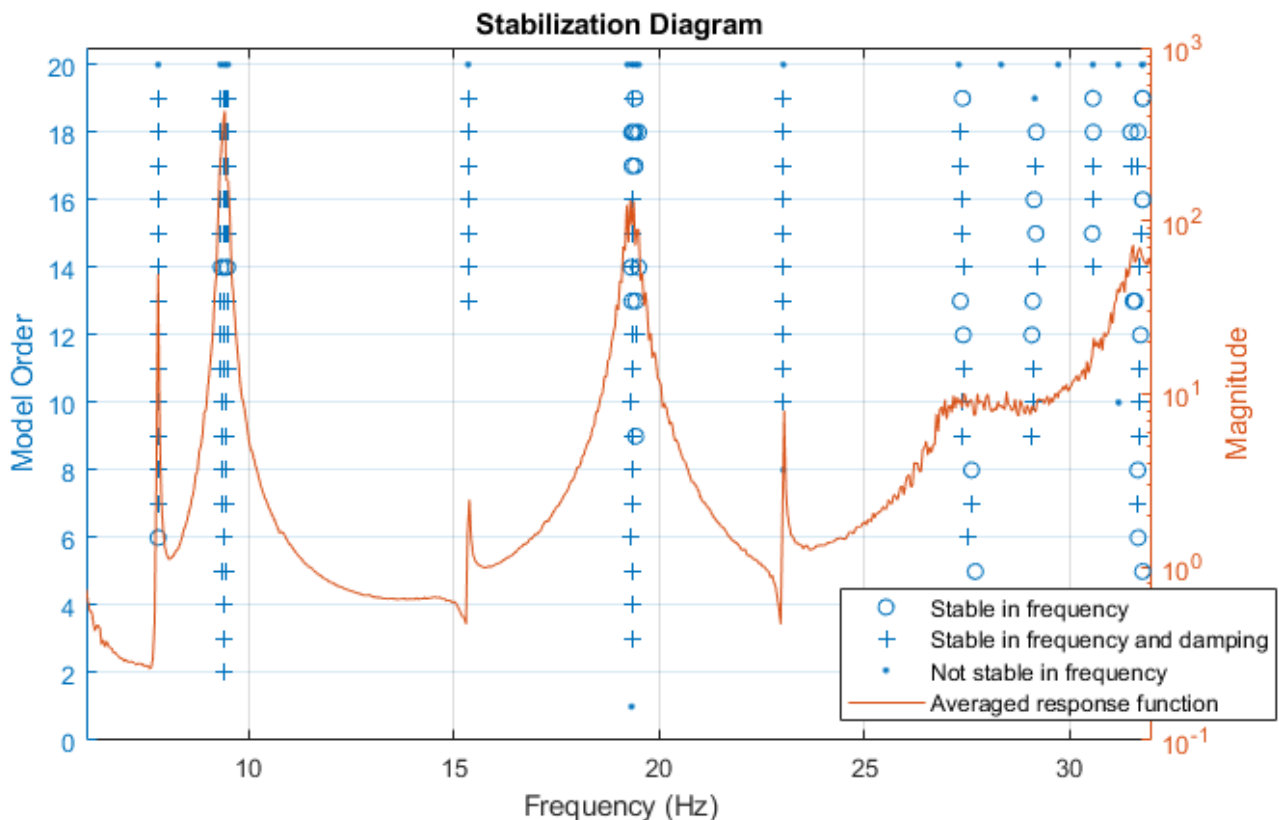

Inspection of the plot and `pf` values suggests a refined list of true natural frequencies:

```
Freq = [7.8 9.4 15.3 19.3 23.0 27.3 29.2 31.7];
```

We use the values in `Freq` vector as a guide for picking most dominant modes from the model `sys2`. This is done using the `modalfit` command.

```
[fn,dr,ms] = modalfit(sys2,f,length(Freq),'PhysFreq',Freq);
```

fn are the natural frequencies (in Hz), dr the corresponding damping coefficients and ms the normalized residuals as column vectors (one column for each natural frequency). In the process of extraction of these modal parameters, only the stable, under-damped poles of the model are used. The ms columns contain data for only the poles with positive imaginary part.

Mode Shape Visualization

To visualize the various bending modes, we need the modal parameters extracted above. In addition we need information on the location of the measurement points. These positions (x-y values) is recorded for each accelerometer in the matrix AccePos:

```
AccelPos = [...    % see figure 2
    16.63 18.48;   % nearest right of center
    16.63 24.48;
    27.90 22.22;
    27.90 28.22;
    37.17 25.97;
    37.17 31.97;
    46.44 29.71;
    46.44 35.71;
    55.71 33.46;
    55.71 39.46;  % farthest right
   -16.63 18.48; % nearest left of center
   -16.63 24.18;
   -27.90 22.22;
   -27.90 28.22;
   -37.17 25.97;
   -37.17 31.97;
   -46.44 29.71;
   -46.44 35.71;
   -55.71 33.46;
   -55.71 39.46]; % farthest left
```

The mode shapes are contained in the matrix ms where each column corresponds to the shape for one frequency. Animate the mode by superimposing mode shape amplitudes over the sensor coordinates and varying the amplitudes at the mode's natural frequency. The animation shows the bending with no damping. As an example, consider the mode at 15.3 Hz.

```
AnimateOneMode(3, fn, ms, AccelPos);
```

Mode 3. FN = 15.35 Hz
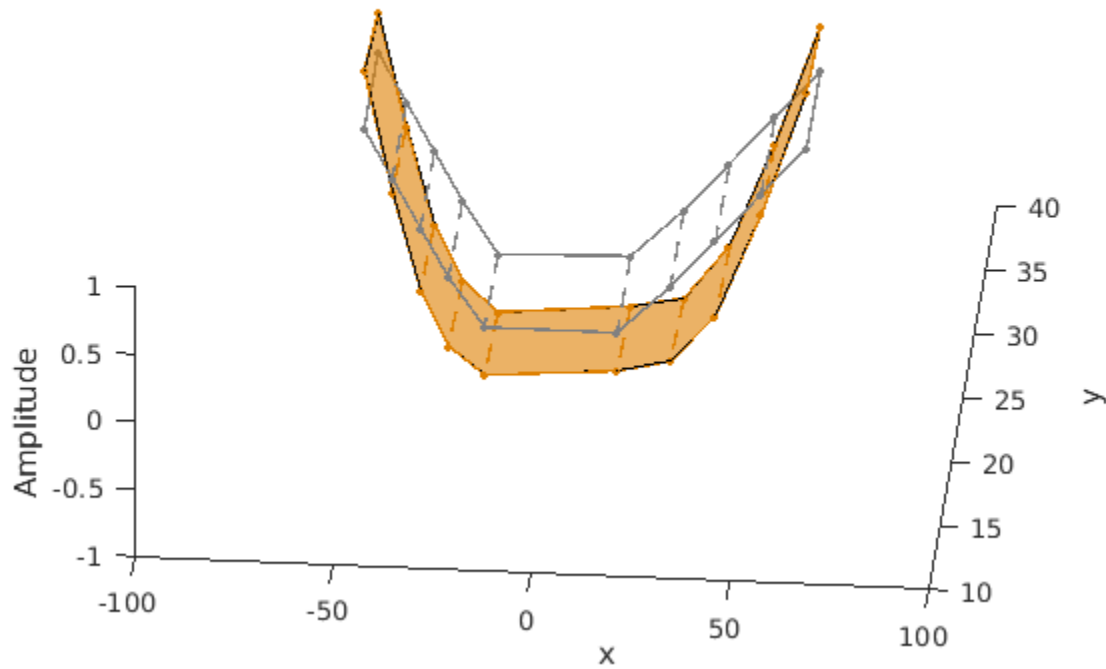
Conclusions

This example shows a parametric model identification based approach to modal parameter estimation. Using a state-space model of 24th order, 8 stable oscillatory modes in the frequency region 6–32 Hz were extracted. The modal information was combined with the knowledge of the accelerometer positions to visualize the various bending modes. Some of these modes are shown in the figures below.

Mode 6. fn = 27.42 Hz

Mode 8. fn = 31.7 Hz
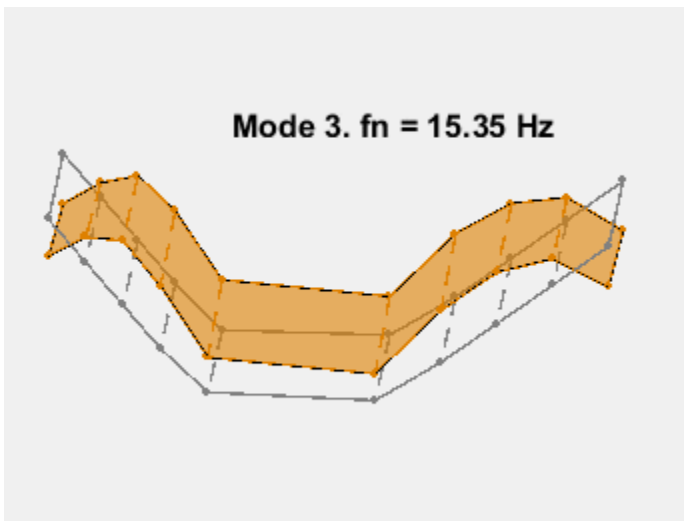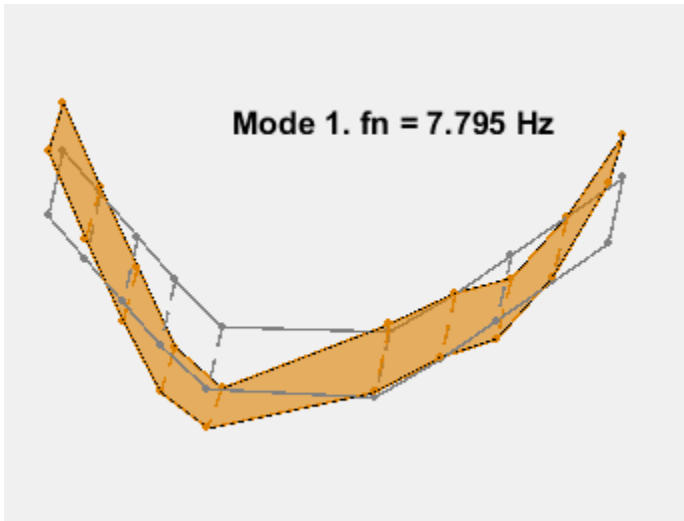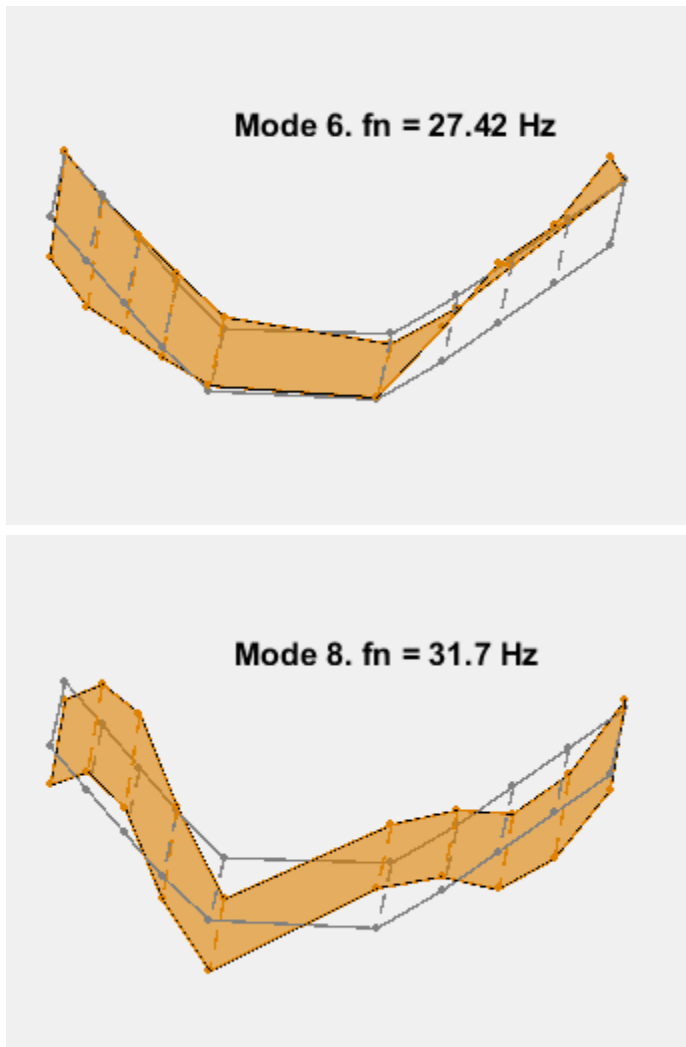
References

[1] Gupta, Abhineet, Peter J. Seiler, and Brian P. Danowsky. "Ground Vibration Tests on a Flexible Flying Wing Aircraft." *AIAA Atmospheric Flight Mechanics Conference, AIAA SciTech Forum.* (AIAA 2016-1753).

```matlab
function AnimateOneMode(ModeNum, fn, ModeShapes, AccelPos)
% Animate one mode.
% ModeNum: Index of the mode.

% Reorder the sensor locations for plotting so that a continuous,
% non-intersecting curve is traced around the body of the aircraft.
PlotOrder = [19:-2:11, 1:2:9, 10:-2:2, 12:2:20, 19];
Fwd = PlotOrder(1:10);
Aft = PlotOrder(20:-1:11);
x = AccelPos(PlotOrder,1);
y = AccelPos(PlotOrder,2);
xAft = AccelPos(Aft,1);
yAft = AccelPos(Aft,2);
xFwd = AccelPos(Fwd,1);
```

```matlab
    yFwd = AccelPos(Fwd,2);

    wn = fn(ModeNum)*2*pi;   % Mode frequency in rad/sec
    T = 1/fn(ModeNum);       % Period of modal oscillation
    Np  = 2.25;              % Number of periods to simulate
    tmax = Np*T;             % Simulate Np periods
    Nt = 100;                % Number of time steps for animation
    t = linspace(0,tmax,Nt);
    ThisMode = ModeShapes(:,ModeNum)/max(abs(ModeShapes(:,ModeNum))); % normalized for plotting
    z0 = ThisMode(PlotOrder);
    zFwd = ThisMode(Fwd);
    zAft = ThisMode(Aft);

    clf
    col1 = [1 1 1]*0.5;
    xx = reshape([[xAft, xFwd]'; NaN(2,10)],[2 20]);
    yy = reshape([[yAft, yFwd]'; NaN(2,10)],[2 20]);
    plot3(x,y,0*z0,'-', x,y,0*z0,'.', xx(:), yy(:), zeros(40,1),'--',...
        'Color',col1,'LineWidth',1,'MarkerSize',10,'MarkerEdgeColor',col1);
    xlabel('x')
    ylabel('y')
    zlabel('Amplitude')
    ht = max(abs(z0));
    axis([-100  100  10  40 -ht ht])
    view(5,55)
    title(sprintf('Mode %d. FN = %s Hz', ModeNum, num2str(fn(ModeNum),4)));

    % Animate by updating z-coordinates.
    hold on
    col2 = [0.87 0.5 0];
    h1 = plot3(x,y,0*z0,'-', x,y,0*z0,'.', xx(:), yy(:), zeros(40,1),'--',...
        'Color',col2,'LineWidth',1,'MarkerSize',10,'MarkerEdgeColor',col2);
    h2 = fill3(x,y,0*z0,col2,'FaceAlpha',0.6);
    hold off

    for k = 1:Nt
       Rot1 = cos(wn*t(k));
       Rot2 = sin(wn*t(k));
       z_t = real(z0)*Rot1 - imag(z0)*Rot2;
       zAft_t = real(zAft)*Rot1 - imag(zAft)*Rot2;
       zFwd_t = real(zFwd)*Rot1 - imag(zFwd)*Rot2;
       zz = reshape([[zAft_t, zFwd_t]'; NaN(2,10)],[2 20]);
       set(h1(1),'ZData',z_t)
       set(h1(2),'ZData',z_t)
       set(h1(3),'ZData',zz(:))
       h2.Vertices(:,3) = z_t;
       pause(0.1)
    end

end
```

Use LSTM Network for Linear System Identification

This example shows how to use long short-term memory (LSTM) neural networks to estimate a linear system and compares this approach to transfer function estimation.

In this example, you investigate the ability of an LSTM network to capture the underlying dynamics of a modeled system. To do this, you train an LSTM network on the input and output signal from a linear transfer function, and measure the accuracy of the network response to a step change.

Transfer Function

This example uses a fourth-order transfer function with mixed fast and slow dynamics and moderate damping. The moderate damping causes the system dynamics to damp out over a longer time horizon and shows the ability of an LSTM network to capture the mixed dynamics without some of the important response dynamics damping out. Construct the transfer function by specifying the poles and zeros of the system.

```
fourthOrderMdl = zpk(-4,[-9+5i;-9-5i;-2+50i;-2-50i],5e5);
[stepResponse,stepTime] = step(fourthOrderMdl);
```

Plot the step response of the transfer function.

```
plot(stepTime,stepResponse)
grid on
axis tight
title('Fourth-order mixed step response')
```


4-219
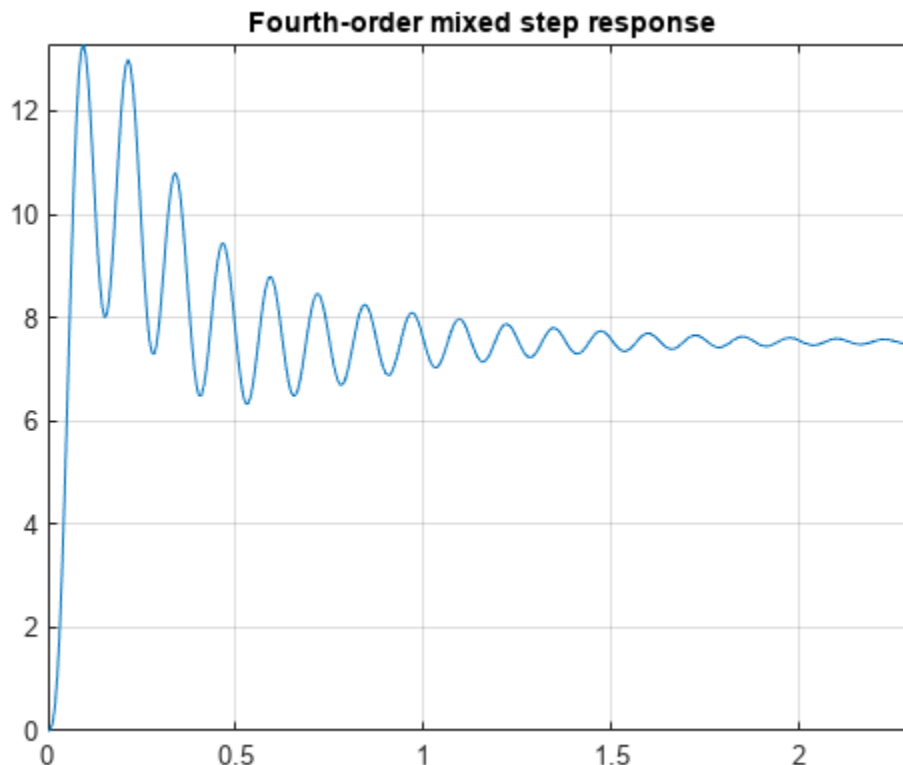
The Bode plot shows the bandwidth of the system, which is measured as the first frequency where the gain drops below 70.8%, or around 3 dB, of its DC value.

```
bodeplot(fourthOrderMdl)
```

Bode Diagram
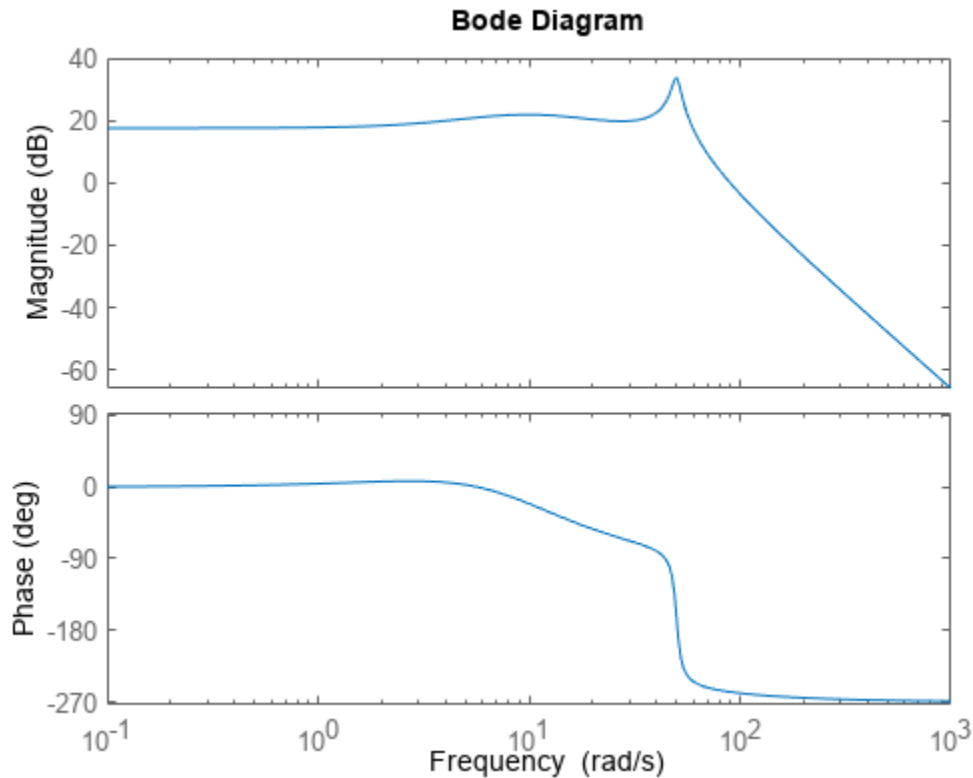

```
fb = bandwidth(fourthOrderMdl)
```

```
fb = 62.8858
```

Generate Training Data

Construct a data set of input and output signals that you can use to train an LSTM network. For the input, generate a random Gaussian signal. Simulate the response of the transfer function `fourthOrderMdl` to this input to obtain the output signal.

Gaussian Noise Data

Specify properties for the random Gaussian noise training signal.

```
signalType = 'rgs'; % Gaussian
signalLength = 5000; % Number of points in the signal
fs = 100; % Sampling frequency
signalAmplitude = 1; % Maximum signal amplitude
```

Generate the Gaussian noise signal using `idinput` and scale the result.

```
urgs = idinput(signalLength,signalType);
urgs = (signalAmplitude/max(urgs))*urgs';
```

Generate the time signal based on the sample rate.

```
trgs = 0:1/fs:length(urgs)/fs-1/fs;
```

Use the `lsim` function to generate the response of the system and store the result in `yrgs`. Transpose the simulated output so that it corresponds to the LSTM data structure, which requires row vectors and not column vectors.

```
yrgs = lsim(fourthOrderMdl,urgs,trgs);
yrgs = yrgs';
```

Similarly, create a shorter validation signal to use during network training.

```
xval = idinput(100,signalType);
yval = lsim(fourthOrderMdl,xval,trgs(1:100));
```

Create and Train Network

The following network architecture was determined by using a Bayesian optimization routine where the Bayesian optimization cost function uses independent validation data (see the accompanying `bayesianOptimizationForLSTM.mlx` for the details). Although multiple architectures may work, this optimization provides the most computationally efficient one. The optimization process also showed that as the complexity of the transfer function increases when applying LSTM to other linear transfer functions, the architecture of the network does not change significantly. Rather, the number of epochs needed to train the network increases. The number of hidden units required for modeling a system is related to how long the dynamics take to damp out. In this case there are two distinct parts to the response: a high frequency response and a low frequency response. A higher number of hidden units are required to capture the low frequency response. If a lower number of units are selected the high frequency response is still modeled. However, the estimation of the low frequency response deteriorates.

Create the network architecture.

```
numResponses = 1;
featureDimension = 1;
numHiddenUnits = 100;
maxEpochs = 1000;
miniBatchSize = 200;

Networklayers = [sequenceInputLayer(featureDimension) ...
    lstmLayer(numHiddenUnits) ...
    lstmLayer(numHiddenUnits) ...
    fullyConnectedLayer(numResponses) ...
    regressionLayer];
```

The initial learning rate impacts the success of the network. Using an initial learning rate that is too high results in high gradients, which lead to longer training times. Longer training times can lead to saturation of the fully connected layer of the network. When the network saturates, the outputs diverge and the network outputs a `NaN` value. Hence, use the default value of 0.01, which is a relatively low initial learning rate. This results in a monotonically decreasing residual and loss curves. Use a piecewise rate schedule to keep the optimization algorithm from getting trapped in local minima at the start of the optimization routine.

```
options = trainingOptions('adam', ...
    'MaxEpochs',maxEpochs, ...
    'MiniBatchSize',miniBatchSize, ...
```

```
        'GradientThreshold',10, ...
        'Shuffle','once', ...
        'Plots','training-progress',...
        'ExecutionEnvironment','gpu',...
        'LearnRateSchedule','piecewise',...
        'LearnRateDropPeriod',100,...
        'Verbose',0,...
        'ValidationData',[{xval'} {yval'}]);

loadNetwork = true; % Set to true to train the network using a parallel pool.
if loadNetwork
    load('fourthOrderMdlnet','fourthOrderNet')
else
    poolobj = parpool;
    fourthOrderNet = trainNetwork(urgs,yrgs,Networklayers,options);
    delete(poolobj)
    save('fourthOrderMdlnet','fourthOrderNet','urgs','yrgs');
end
```

Evaluate Model Performance

A network performs well if it is successful in capturing the system dynamic behavior. Evaluate the network performance by measuring the ability of the network to accurately predict the system response to a step input.

Construct a step input.

```
stepTime = 2; % In seconds
stepAmplitude = 0.1;
stepDuration = 4; % In seconds

% Construct step signal and system output.
time = (0:1/fs:stepDuration)';
stepSignal = [zeros(sum(time<=stepTime),1);stepAmplitude*ones(sum(time>stepTime),1)];
systemResponse = lsim(fourthOrderMdl,stepSignal,time);

% Transpose input and output signal for network inputs.
stepSignal = stepSignal';
systemResponse = systemResponse';
```

Use the trained network to evaluate the system response. Compare the system and estimated responses in a plot.

```
fourthOrderMixedNetStep = predict(fourthOrderNet,stepSignal);

figure
title('Step response estimation')
plot(time,systemResponse,'k', time, fourthOrderMixedNetStep)
grid on
legend('System','Estimated')
title('Fourth-Order Step')
```

Fourth-Order Step
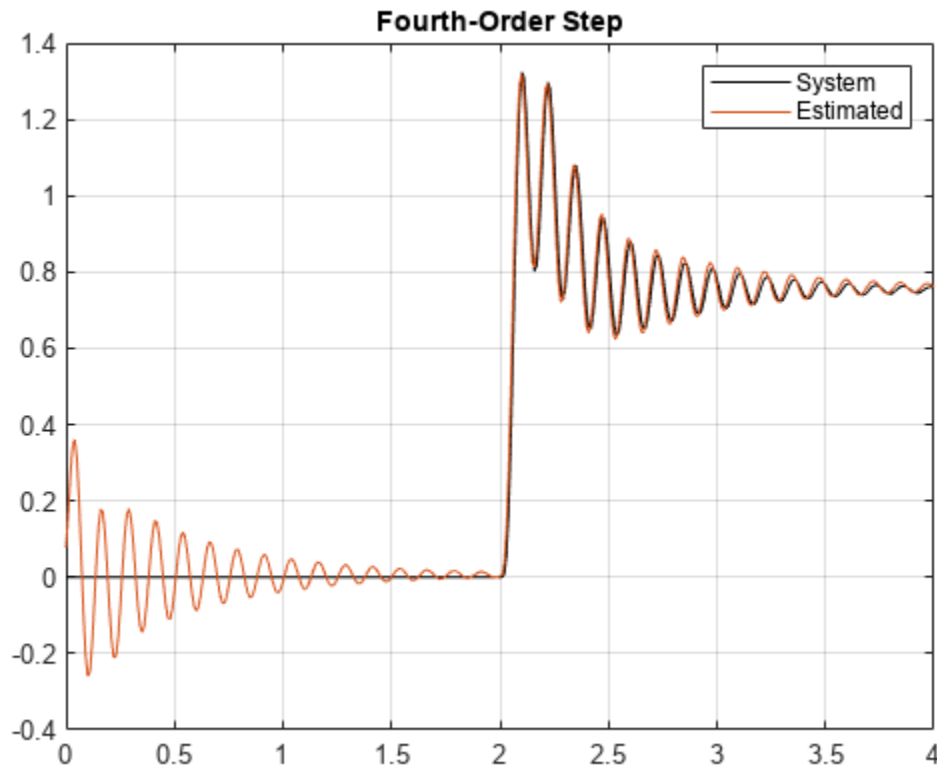
The plot shows two issues with the fit. First, the initial state of the network is not stationary, which results in transient behavior at the start of the signal. Second, the prediction of the network has a slight offset.

Initialize Network and Adjust Fit

To initialize the network state to the correct initial condition, you must update the network state to correspond to the state of the system at the start of the test signal.

You can adjust the initial state of the network by comparing the estimated response of the system at the initial condition to the actual response of the system. Use the difference between the estimation of the initial state by the network and the actual response of the initial state to correct for the offset in the system estimation.

Set Network Initial State

As the network performs estimation using a step input from 0 to 1, the states of the LSTM network (cell and hidden states of the LSTM layers) drift toward the correct initial condition. To visualize this, extract the cell and hidden state of the network at every time step using the `predictAndUpdateState` function.

Use only the cell and hidden state values prior to the step, which occurs at 2 seconds. Define a time marker for 2 seconds, and extract the values up to this marker.

```
stepMarker = time <= 2;
yhat = zeros(sum(stepMarker),1);
```

4-223

```
hiddenState = zeros(sum(stepMarker),200); % 200 LSTM units
cellState = zeros(sum(stepMarker),200);
for ntime = 1:sum(stepMarker)
    [fourthOrderNet,yhat(ntime)] = predictAndUpdateState(fourthOrderNet,stepSignal(ntime)');
    hiddenState(ntime,:) = fourthOrderNet.Layers(2,1).HiddenState;
    cellState(ntime,:) = fourthOrderNet.Layers(2,1).CellState;
end
```

Next, plot the hidden and cell states over the period before the step and confirm that they converge to fixed values.

```
figure
subplot(2,1,1)
plot(time(1:200),hiddenState(1:200,:))
grid on
axis tight
title('Hidden State')
subplot(2,1,2)
plot(time(1:200),cellState(1:200,:))
grid on
axis tight
title('Cell State')
```
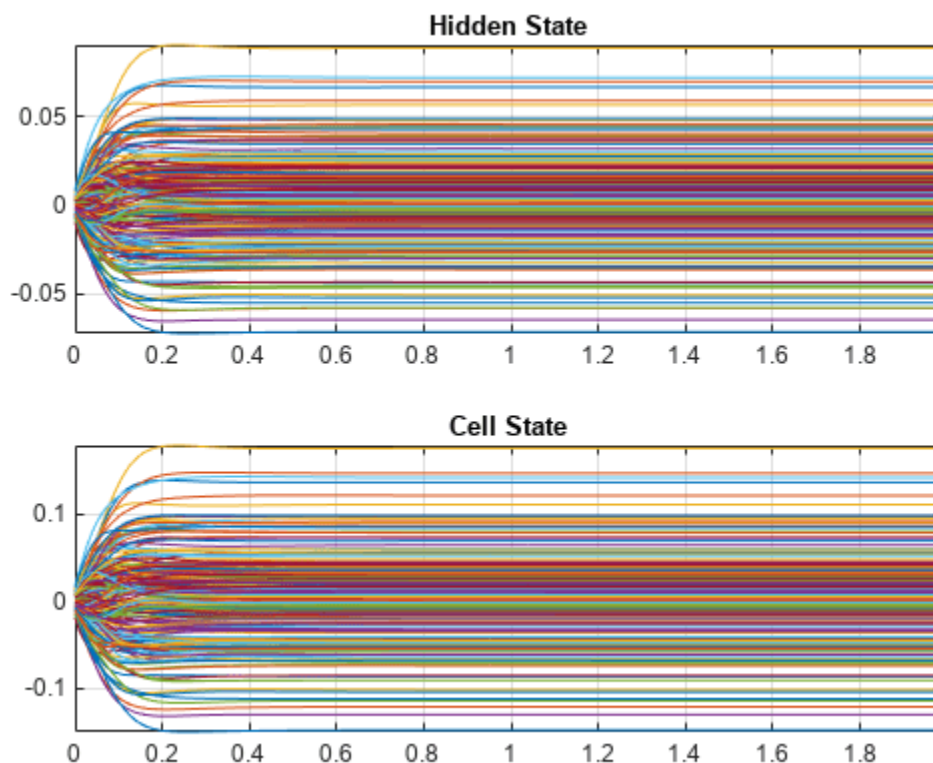

To initialize the network state for a zero input signal, choose an input signal of zero and choose the duration so that the signal is long enough for the networks to reach steady state.

```
initializationSignalDuration = 10; % In seconds
initializationValue = 0;
```

```
initializationSignal = initializationValue*ones(1,initializationSignalDuration*fs);

fourthOrderNet = predictAndUpdateState(fourthOrderNet,initializationSignal);
```

Verify that the initial condition is zero or close to zero.

```
figure
zeroMapping = predict(fourthOrderNet,initializationSignal);
plot(zeroMapping)
axis tight
```
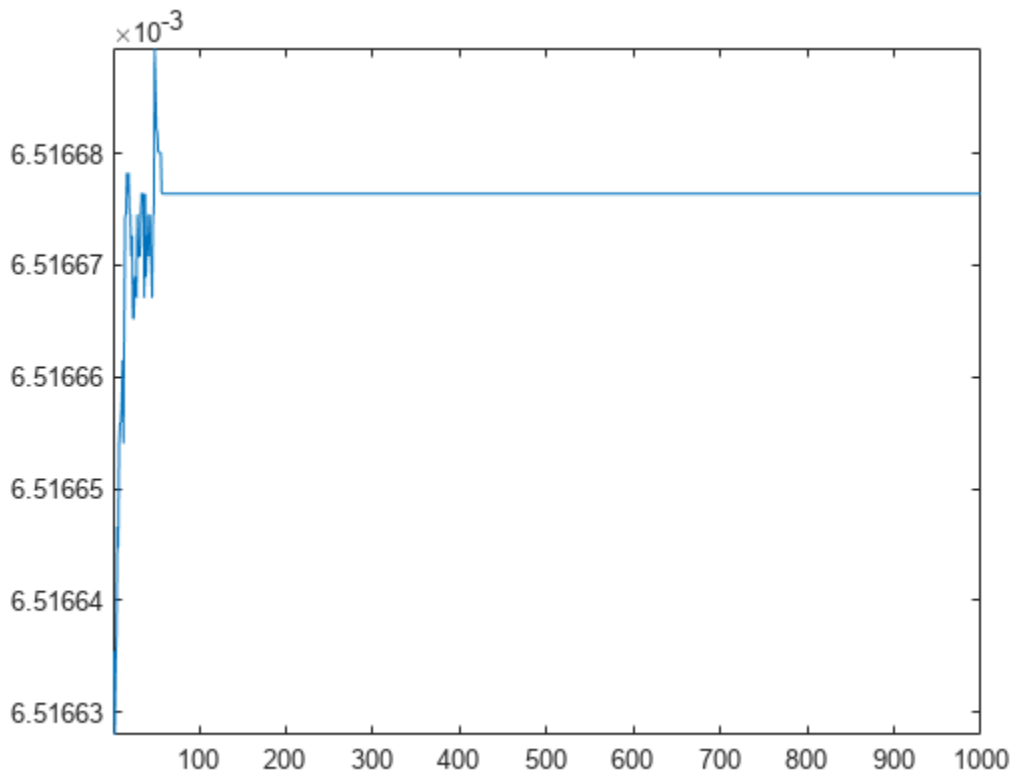

Now that the network is correctly initialized, use the network to predict the step response again and plot the results. The initial disturbance is gone.

```
fourthOrderMixedNetStep = predict(fourthOrderNet,stepSignal);

figure
title('Step response estimation')
plot(time,systemResponse,'k', ...
    time,fourthOrderMixedNetStep,'b')
grid on
legend('System','Estimated')
title('Fourth-Order Step - Adjusted State')
```

Fourth-Order Step - Adjusted State
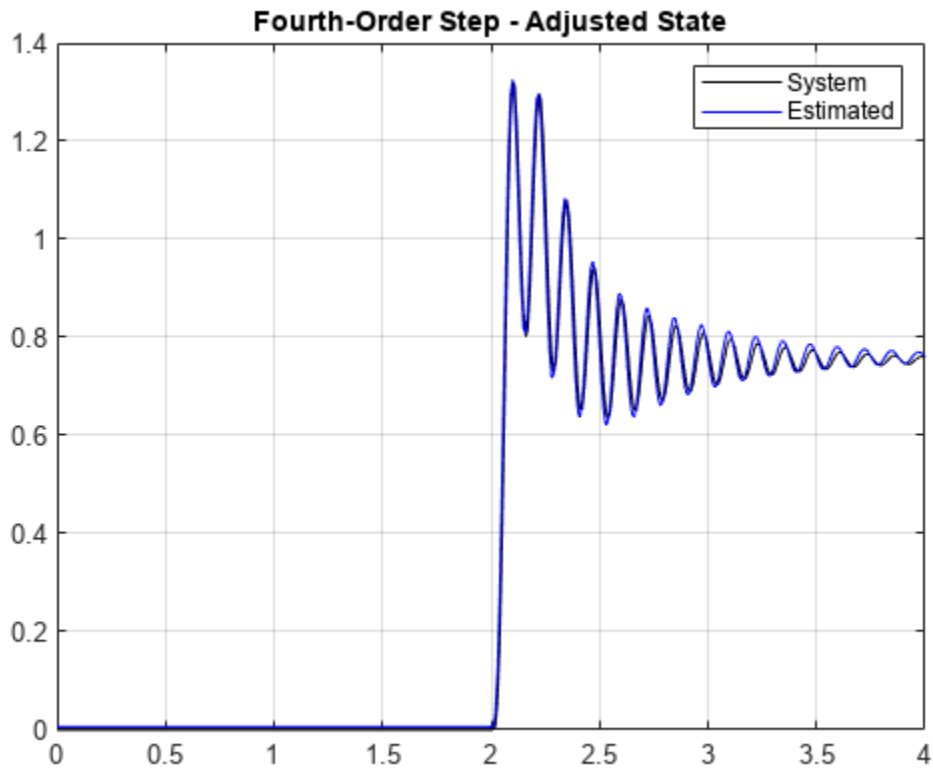
Adjust Network Offset

Even after you set the network initial state to compensate for the initial condition of the test signal, a small offset is still visible in the predicted response. This is because of the incorrect bias term that the LSTM network learned during training. You can fix the offset by using the same initialization signal that was used for updating the network states. The initialization signal is expected to map the network to zero. The offset between zero and the network estimation is the error in the bias term learned by the network. Summing the bias term calculated at each layer comes close to the bias detected in the response. Adjusting for the network bias term at the network output, however, is easier than adjusting the individual bias terms in each layer of the network.

```
bias = mean(predict(fourthOrderNet,initializationSignal));
fourthOrderMixedNetStep = fourthOrderMixedNetStep-bias;

figure
title('Step response estimation')
plot(time,systemResponse,'k',time,fourthOrderMixedNetStep,'b-')
legend('System','Estimated')
title('Fourth-Order Step - Adjusted Offset')
```

Fourth-Order Step - Adjusted Offset
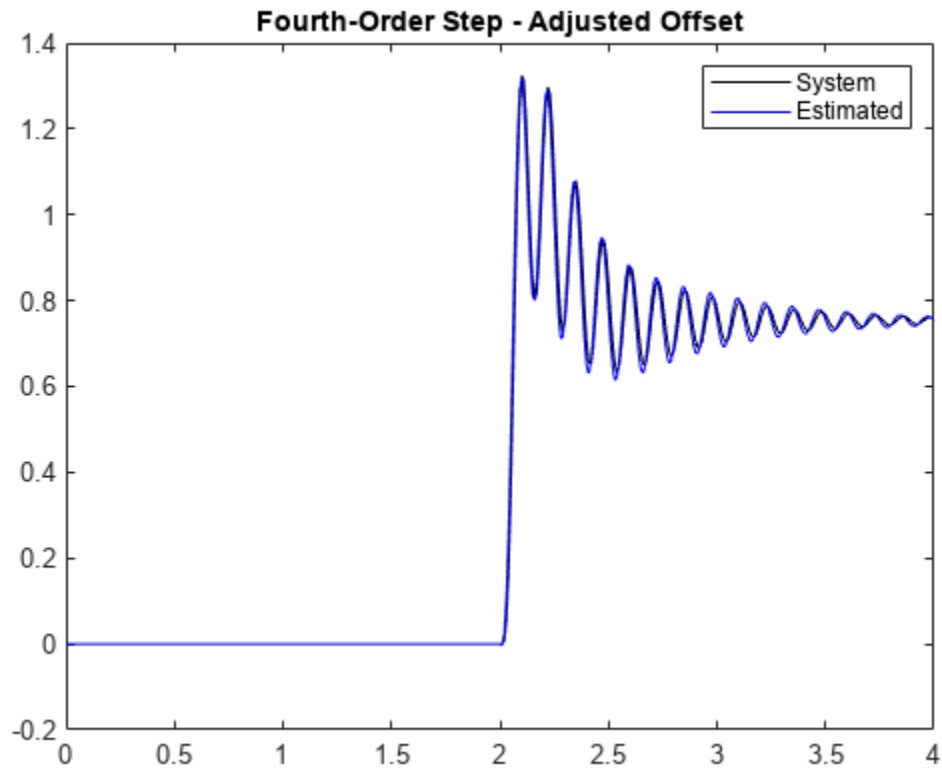
Move Outside of Training Range

All the signals used to train the network had a maximum amplitude of 1 and the step function had an amplitude of 0.1. Now, investigate the behavior of the network outside of these ranges.

Time Shift

Introduce a time shift by adjusting the time of the step. Set the time of the step to 3 seconds, 1 second longer than in the training set. Plot the resulting network output and note that the output is correctly delayed by 1 second.

```
stepTime = 3; % In seconds
stepAmplitude = 0.1;
stepDuration = 5; % In seconds
[stepSignal,systemResponse,time] = generateStepResponse(fourthOrderMdl,stepTime,stepAmplitude,ste

fourthOrderMixedNetStep = predict(fourthOrderNet,stepSignal);
bias = fourthOrderMixedNetStep(1) - initializationValue;
fourthOrderMixedNetStep = fourthOrderMixedNetStep-bias;

figure
plot(time,systemResponse,'k', time,fourthOrderMixedNetStep,'b')
grid on
axis tight
```
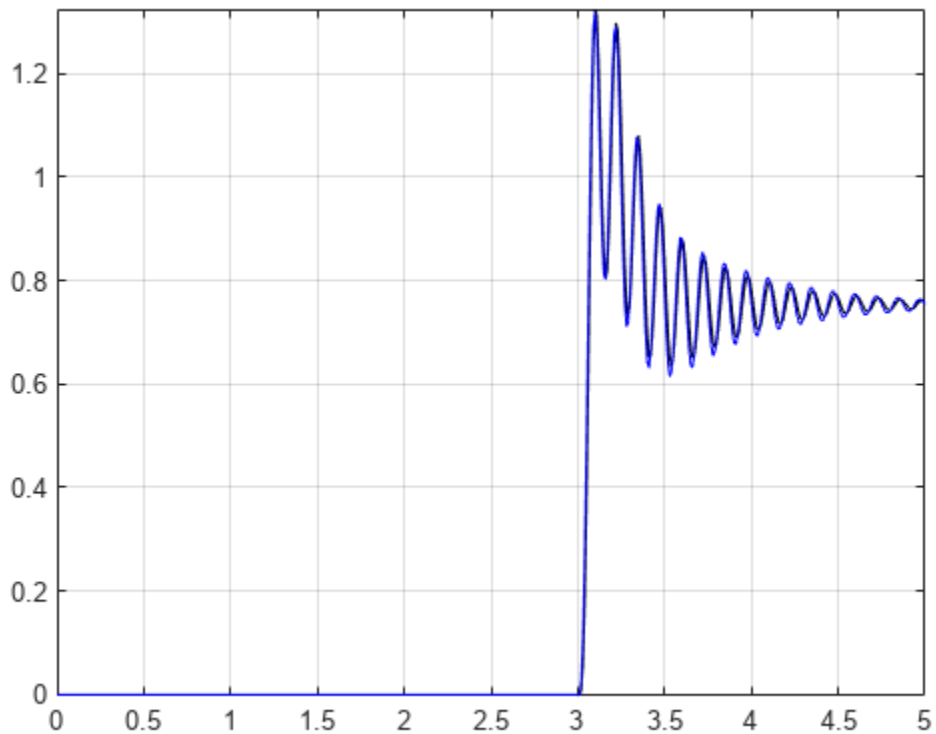
Amplitude Shift

Next, increase the amplitude of the step function to investigate the network behavior as the system inputs move outside of the range of the training data. To measure the drift outside of the training data range, you can measure the probability density function of the amplitudes in the Gaussian noise signal. Visualize the amplitudes in a histogram.

```
figure
histogram(urgs,'Normalization','pdf')
grid on
```
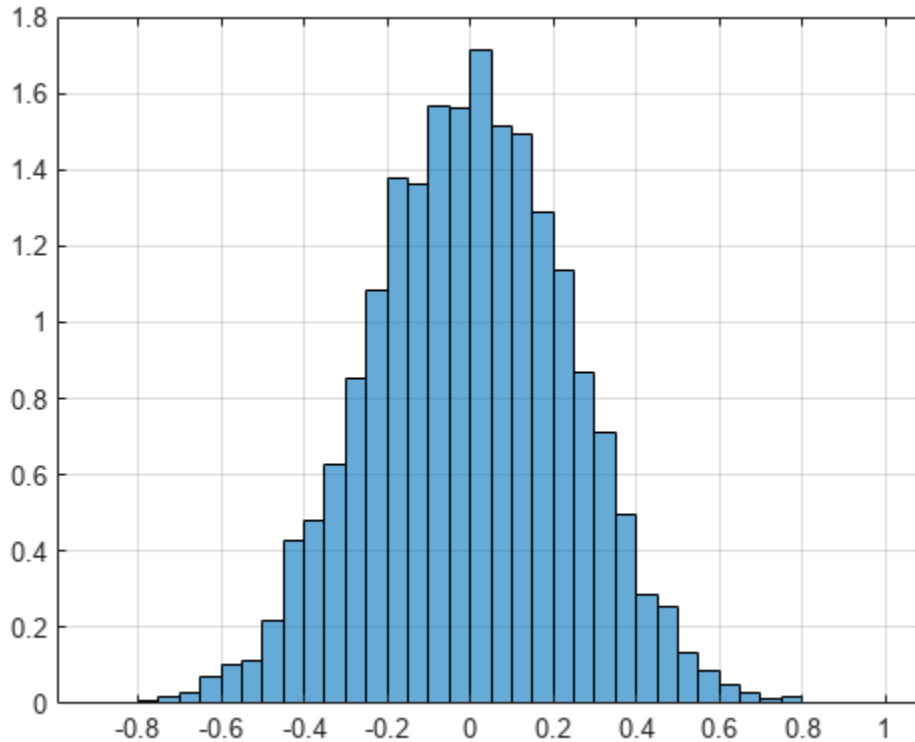
Set the amplitude of the step function according to the percentile of the distribution. Plot the error rate as a function of the percentile.

```
pValues = [60:2:98, 90:1:98, 99:0.1:99.9 99.99];
stepAmps = prctile(urgs,pValues); % Amplitudes
stepTime = 3; % In seconds
stepDuration = 5; % In seconds

stepMSE = zeros(length(stepAmps),1);
fourthOrderMixedNetStep = cell(length(stepAmps),1);
steps = cell(length(stepAmps),1);

for nAmps = 1:length(stepAmps)
    % Fourth-order mixed
    [stepSignal,systemResponse,time] = generateStepResponse(fourthOrderMdl,stepTime,stepAmps(nAmp

    fourthOrderMixedNetStep{nAmps} = predict(fourthOrderNet,stepSignal);
    bias = fourthOrderMixedNetStep{nAmps}(1) - initializationValue;
    fourthOrderMixedNetStep{nAmps} = fourthOrderMixedNetStep{nAmps}-bias;

    stepMSE(nAmps) = sqrt(sum((systemResponse-fourthOrderMixedNetStep{nAmps}).^2));
    steps{nAmps,1} = systemResponse;
end

figure
plot(pValues,stepMSE,'bo')
title('Prediction Error as a Function of Deviation from Training Rrange')
```

4-229

```
grid on
axis tight
```

Prediction Error as a Function of Deviation from Training Rrange
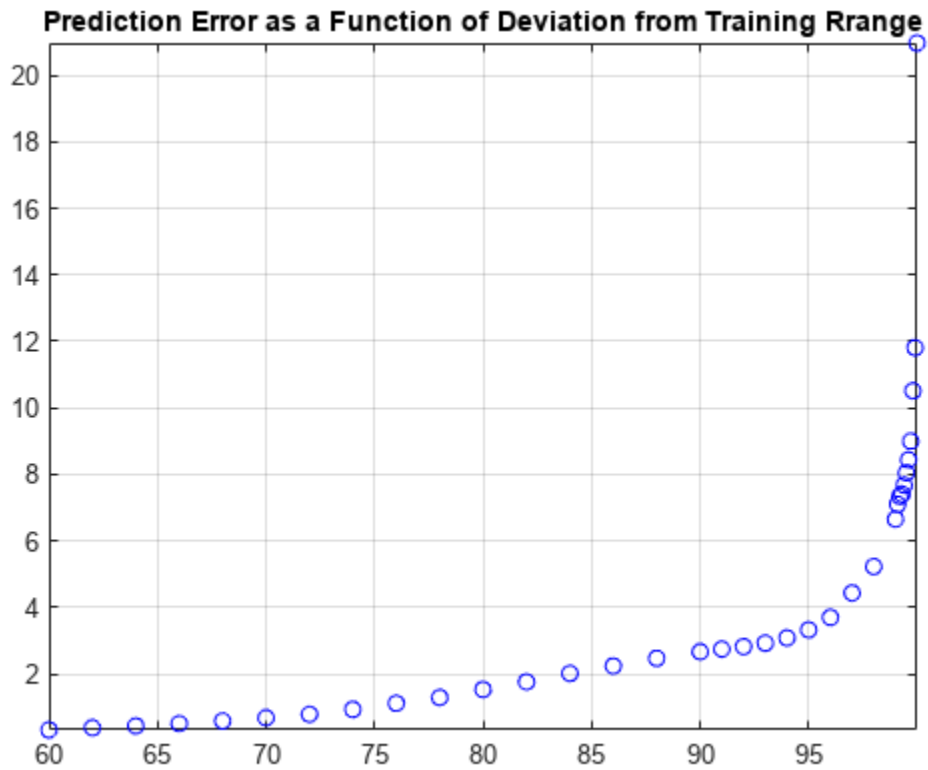

```
subplot(2,1,1)
plot(time,steps{1},'k', time,fourthOrderMixedNetStep{1},'b')
grid on
axis tight
title('Best Performance')
xlabel('time')
ylabel('System Response')
subplot(2,1,2)
plot(time,steps{end},'k', time,fourthOrderMixedNetStep{end},'b')
grid on
axis tight
title('Worst Performance')
xlabel('time')
ylabel('System Response')
```
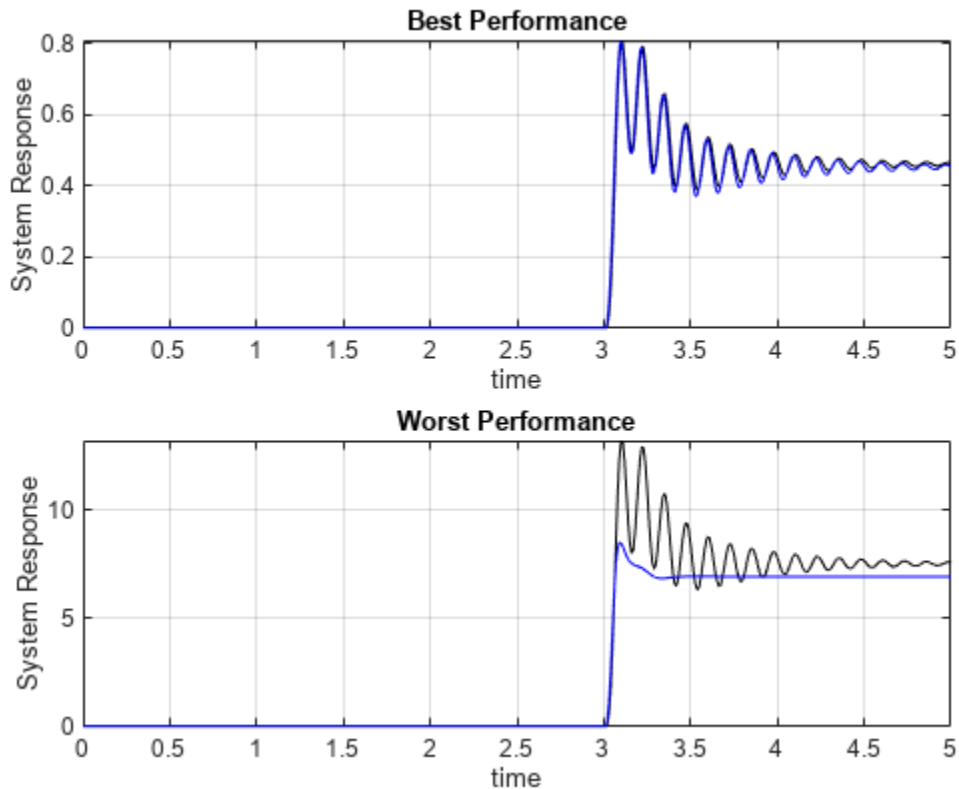
As the amplitude of the step response moves outside of the range of the training set, the LSTM attempts to estimate the average value of the response.

These results show the importance of using training data that is in the same range as the data that will be used for prediction. Otherwise, prediction results are unreliable.

Change System Bandwidth

Investigate the effect of the system bandwidth on the number of hidden units selected for the LSTM network by modeling the fourth-order mixed-dynamics transfer function with four different networks:

- Small network with 5 hidden units and a single LSTM layer
- Medium network with 10 hidden units and a single LSTM layer
- Full network with 100 hidden units and a single LSTM layer
- Deep network with 2 LSTM layers (each with 100 hidden units)

Load the trained networks.

```
load('variousHiddenUnitNets.mat')
```

Generate a step signal.

```
stepTime = 2; % In seconds
stepAmplitude = 0.1;
stepDuration = 4; % In seconds
```

```
% Construct step signal.
time = (0:1/fs:stepDuration)';

stepSignal = [zeros(sum(time<=stepTime),1);stepAmplitude*ones(sum(time>stepTime),1)];
systemResponse = lsim(fourthOrderMdl,stepSignal,time);

% Transpose input and output signal for network inputs.
stepSignal = stepSignal';
systemResponse = systemResponse';
```

Estimate the system response using the various trained networks.

```
smallNetStep = predict(smallNet,stepSignal)-smallNetZeroMapping(end);
medNetStep = predict(medNet,stepSignal)-medNetZeroMapping(end);
fullnetStep = predict(fullNet,stepSignal) - fullNetZeroMapping(end);
doubleNetStep = predict(doubleNet,stepSignal) - doubleNetZeroMapping(end);
```

Plot the estimated response.

```
figure
title('Step response estimation')
plot(time,systemResponse,'k', ...
    time,doubleNetStep,'g', ...
    time,fullnetStep,'r', ...
    time,medNetStep,'c', ...
    time,smallNetStep,'b')
grid on
legend({'System','Double Net','Full Net','Med Net','Small Net'},'Location','northwest')
title('Fourth-Order Step')
```
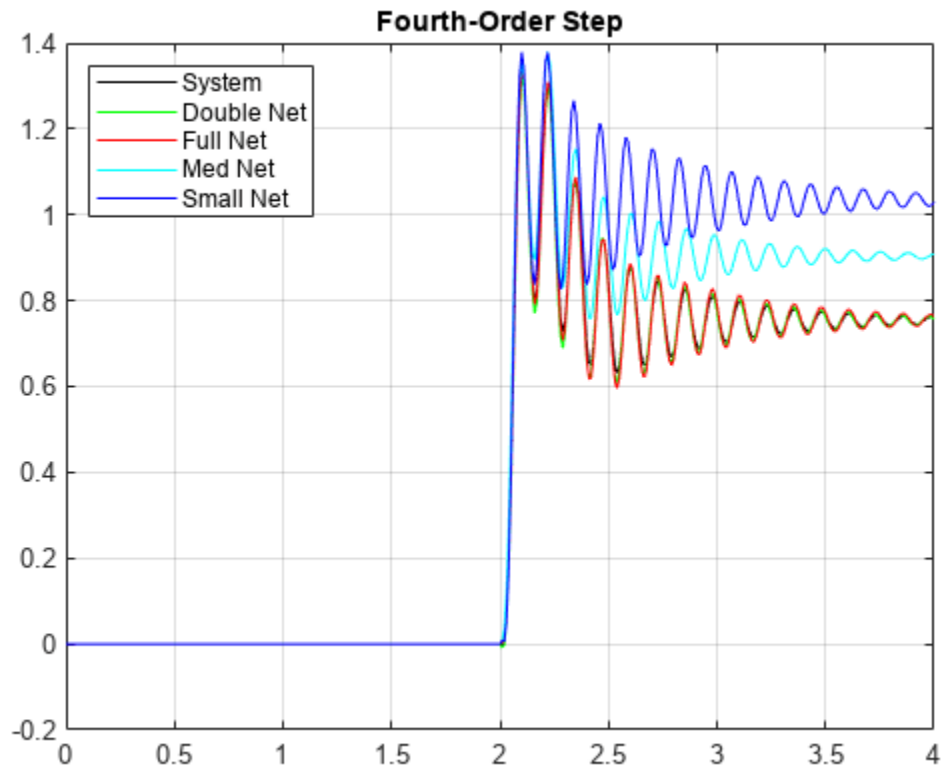
Note all the networks capture the high frequency dynamics in the response well. However, plot a moving average of the responses in order to compare the slow varying dynamics of the system. The ability of the LSTM to capture the longer term dynamics (lower frequency dynamics) of the linear system is directly related to the dynamics of the system and the number of hidden units in the LSTM. The number of layers in the LSTM is not directly related to the long-term behavior but rather adds flexibility to adjust the estimation from the first layer.

```
figure
title('Slow dynamics component')
plot(time,movmean(systemResponse,50),'k')
hold on
plot(time,movmean(doubleNetStep,50),'g')
plot(time,movmean(fullnetStep,50),'r')
plot(time,movmean(medNetStep,50),'c')
plot(time,movmean(smallNetStep,50),'b')
grid on
legend('System','Double Net','Full net','Med Net','Small Net','Location','northwest')
title('Fourth Order Step')
```

Fourth Order Step
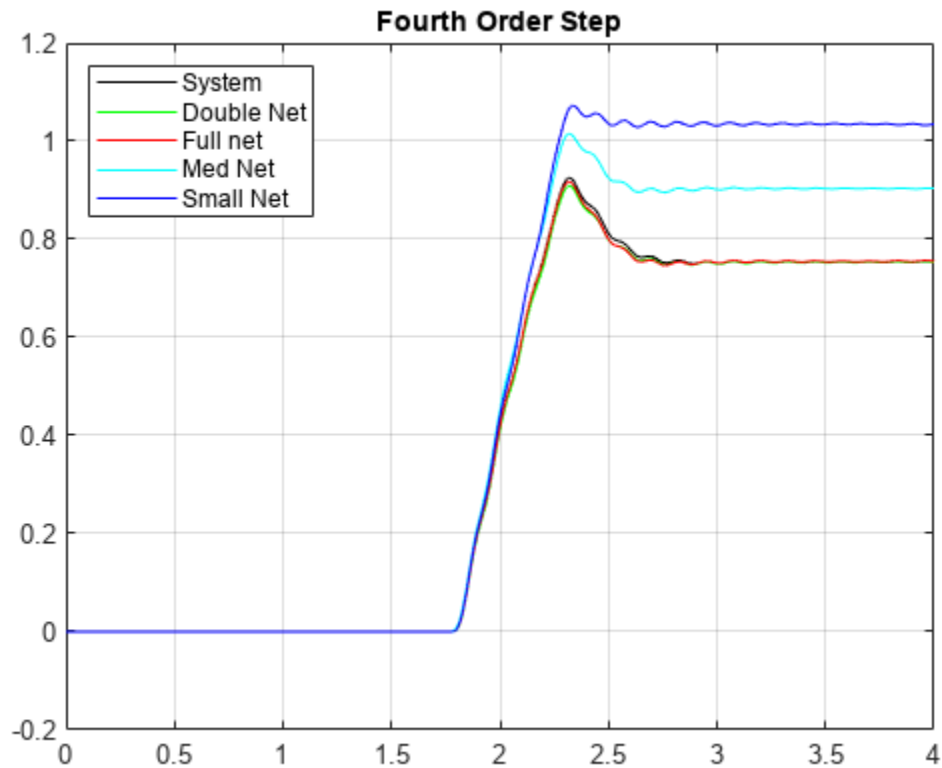
Add Noise to Measured System Response

Add random noise to the system output to explore the effect of noise on the LSTM performance. To this end, add white noise with levels of 1%, 5%, and 10% to the measured system responses. Use the noisy data to train the LSTM network. With the same noisy data sets, estimate linear models by using `tfest`. Simulate these models and use the simulated responses as the baseline for a performance comparison.

Use the same step function as before:

```
stepTime = 2; % In seconds
stepAmplitude = 0.1;
stepDuration = 4; % In seconds

[stepSignal,systemResponse,time] = generateStepResponse(fourthOrderMdl,stepTime,stepAmplitude,ste
```

Load the trained networks and estimate the system response.

```
load('noisyDataNetworks.mat')
netNoise1Step = predictAndAdjust(netNoise1,stepSignal,initializationSignal,initializationValue);
netNoise5Step = predictAndAdjust(netNoise5,stepSignal,initializationSignal,initializationValue);
netNoise10Step = predictAndAdjust(netNoise10,stepSignal,initializationSignal,initializationValue
```

A transfer function estimator (`tfest`) is used to estimate the function at the above noise levels to compare the resilience of the networks to noise (see the accompanying `noiseLevelModels.m` for more details).

```
load('noisyDataTFs.mat')
tfStepNoise1 = lsim(tfNoise1,stepSignal,time);
tfStepNoise5 = lsim(tfNoise5,stepSignal,time);
tfStepNoise10 = lsim(tfNoise10,stepSignal,time);
```

Plot the generated responses.

```
figure
plot(time,systemResponse,'k', ...
    time,netNoise1Step, ...
    time,netNoise5Step, ...
    time,netNoise10Step)
grid on
legend('System Response','1% Noise','5% Noise','10% Noise')
title('Deep LSTM with noisy data')
```
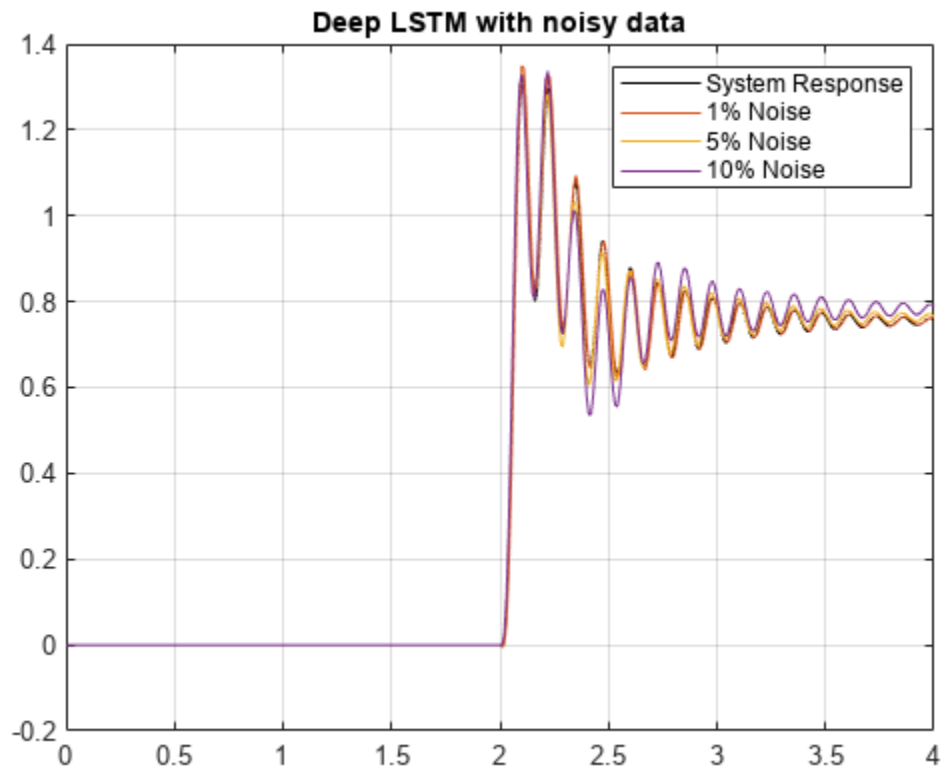

Now, plot the estimated transfer functions.

```
figure
plot(time,systemResponse,'k', ...
    time,tfStepNoise1, ...
    time,tfStepNoise5, ...
    time,tfStepNoise10)
grid on
legend('System Response','1% Noise','5% Noise','10% Noise')
title('Transfer functions fitted to noisy data')
```

Transfer functions fitted to noisy data
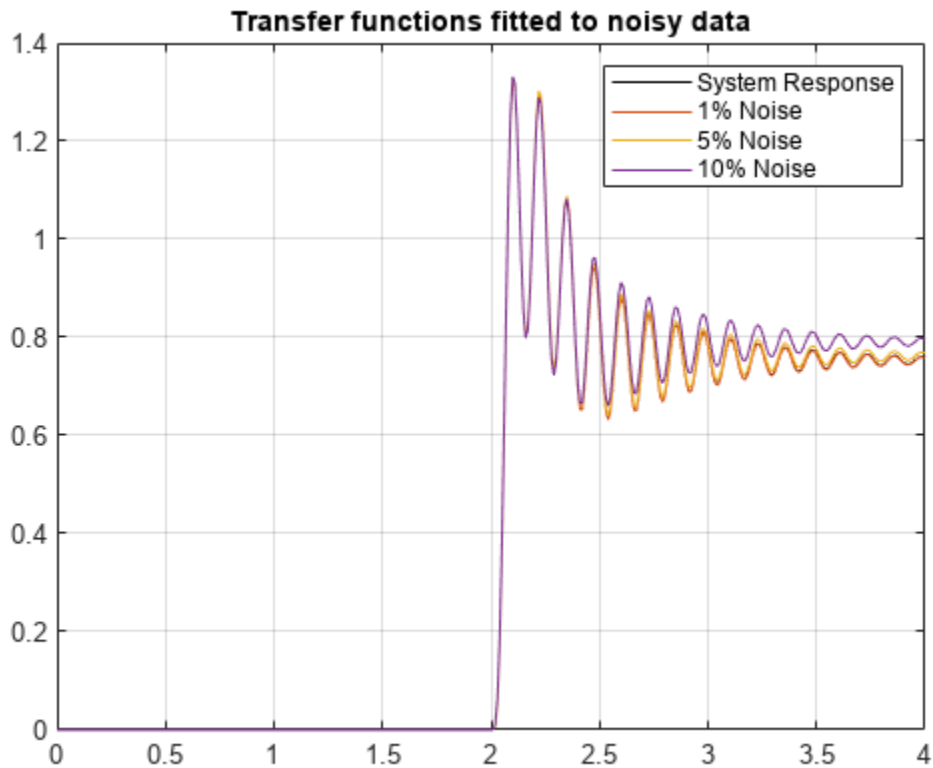
Calculate the mean squared error to better assess the performance of the different models at different noise levels.

```
msefun = @(y,yhat) mean(sqrt((y-yhat).^2)/length(y));

% LSTM errors
lstmMSE(1,:) = msefun(systemResponse,netNoise1Step);
lstmMSE(2,:) = msefun(systemResponse,netNoise5Step);
lstmMSE(3,:) = msefun(systemResponse,netNoise10Step);

% Transfer function errors
tfMSE(1,:) = msefun(systemResponse,tfStepNoise1');
tfMSE(2,:) = msefun(systemResponse,tfStepNoise5');
tfMSE(3,:) = msefun(systemResponse,tfStepNoise10');

mseTbl = array2table([lstmMSE tfMSE],'VariableNames',{'LSTMMSE','TFMSE'})
```

```
mseTbl=3×2 table
    LSTMMSE        TFMSE
    _____    _____

    1.0115e-05    8.8621e-07
    2.5577e-05    9.9064e-06
    5.1791e-05    3.6831e-05
```

Noise has a similar effect on both the LSTM and transfer-function estimation results.

Helper Functions

```matlab
function [stepSignal,systemResponse,time] = generateStepResponse(model,stepTime,stepAmp,signalDur
%Generates a step response for the given model.
%

%Check model type
modelType = class(model);
if nargin < 2
    stepTime = 1;
end

if nargin < 3
    stepAmp = 1;
end

if nargin < 4
    signalDuration = 10;
end

% Constuct step signal
if model.Ts == 0
    Ts = 1e-2;
    time = (0:Ts:signalDuration)';
else
    time = (0:model.Ts:signalDuration)';
end
stepSignal = [zeros(sum(time<=stepTime),1);stepAmp*ones(sum(time>stepTime),1)];
switch modelType
    case {'tf', 'zpk'}
        systemResponse = lsim(model,stepSignal,time);
    case 'idpoly'
        systemResponse = sim(model,stepSignal,time);
    otherwise
        error('Model passed is not supported')
end

stepSignal = stepSignal';
systemResponse = systemResponse';
end
```

See Also

5

Identifying Process Models

What Is a Process Model?

The structure of a *process model* is a simple continuous-time transfer function that describes linear system dynamics in terms of one or more of the following elements:

- Static gain K_p.
- One or more time constants T_{pk}. For complex poles, the time constant is called T_ω—equal to the inverse of the natural frequency—and the damping coefficient is ζ (`zeta`).
- Process zero T_z.
- Possible time delay T_d before the system output responds to the input (*dead time*).
- Possible enforced integration.

Process models are popular for describing system dynamics in many industries and apply to various production environments. The advantages of these models are that they are simple, support transport delay estimation, and the model coefficients have an easy interpretation as poles and zeros.

You can create different model structures by varying the number of poles, adding an integrator, or adding or removing a time delay or a zero. You can specify a first-, second-, or third-order model, and the poles can be real or complex (underdamped modes). For more information, see "Process Model Structure Specification" on page 5-36.

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

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

Such that, $Y(s) = G(s)U(s) + E(s)$, where $Y(s)$, $U(s)$, and $E(s)$ represent the Laplace transforms of the output, input, and output error, respectively. The output error, $e(t)$, is white Gaussian noise with variance λ. You can account for colored noise at the output by adding a disturbance model, $H(s)$, such that $Y(s) = G(s)U(s) + H(s)E(s)$. For more information, see the `NoiseTF` property of `idproc`.

A multi-input multi-output (MIMO) process model contains a SISO process model corresponding to each input-output pair in the system. For example, for a two-input, two-output process model:

$$Y_1(s) = G_{11}(s)U_1(s) + G_{12}(s)U_2(s) + E_1(s)$$
$$Y_2(s) = G_{21}(s)U_1(s) + G_{22}(s)U_2(s) + E_2(s)$$

Where, $G_{ij}(s)$ is the SISO process model between the ith output and the jth input. $E_1(s)$ and $E_2(s)$ are the Laplace transforms of the two output errors.

See Also

Related Examples

- "Estimate Process Models Using the App" on page 5-4
- "Estimate Process Models at the Command Line" on page 5-8
- "Data Supported by Process Models" on page 5-3
- "Process Model Structure Specification" on page 5-36

Data Supported by Process Models

You can estimate low-order (up to third order), continuous-time transfer functions using regularly sampled time- or frequency-domain `iddata` or `idfrd` data objects. The frequency-domain data may have a zero sample time.

You must import your data into the MATLAB workspace, as described in "Data Preparation".

See Also

Related Examples

- "Estimate Process Models Using the App" on page 5-4
- "Estimate Process Models at the Command Line" on page 5-8

More About

- "What Is a Process Model?" on page 5-2

Estimate Process Models Using the App

Before you can perform this task, you must have

- Imported data into the System Identification app. See "Import Time-Domain Data into the App" on page 2-13. For supported data formats, see "Data Supported by Process Models" on page 5-3.
- 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 2-5.

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

To learn more about the options in the dialog box, click **Help**.

2 If your model contains multiple inputs or multiple outputs, you can specify whether to estimate the same transfer function for all input-output pairs, or a different transfer function for each. Select the input and output channels in the **Input** and **Output** fields. The fields only appears when you have multiple inputs or outputs. For more information, see "Estimating Multiple-Input, Multi-Output Process Models" on page 5-37.

3 In the **Model Transfer Function** area, specify the model structure using the following options:

- Under **Poles**, select the number of poles, and then select `All real` or `Underdamped`.

Note You need at least two poles to allow underdamped modes (complex-conjugate pair).

- Select the **Zero** check box to include a zero, which is a numerator term other than a constant, or clear the check box to exclude the zero.

- Select the **Delay** check box to include a delay, or clear the check box to exclude the delay.
- Select the **Integrator** check box to include an integrator (self-regulating process), or clear the check box to exclude the integrator.

The **Parameter** area shows as many active parameters as you included in the model structure.

Note By default, the model **Name** is set to the acronym that reflects the model structure, as described in "Process Model Structure Specification" on page 5-36.

---

4 In the **Initial Guess** area, select `Auto-selected` to calculate the initial parameter values for the estimation. The **Initial Guess** column in the Parameter table displays `Auto`. If you do not have a good guess for the parameter values, `Auto` works better than entering an ad hoc value.

5 (Optional) If you approximately know a parameter value, enter this value in the **Initial Guess** column of the Parameter table, and press the **Enter** key. The estimation algorithm uses this value as a starting point. If you know a parameter value exactly, enter this value in the **Initial Guess** column, and press the **Enter** key. Select the corresponding **Known** check box in the table to fix its value.

If you know the range of possible values for a parameter, enter these values into the corresponding **Bounds** field to help the estimation algorithm. Press the **Enter** key after you specify the values.

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

6 In the **Disturbance Model** drop-down list, select one of the available options. For more information about each option, see "Disturbance Model Structure for Process Models" on page 5-38.

7 In the **Focus** drop-down list, select how to weigh the relative importance of the fit at different frequencies. For more information about each option, see "Assigning Estimation Weightings" on page 5-7.

8 In the **Initial condition** drop-down list, specify how you want the algorithm to treat initial states. For more information about the available options, see "Specifying Initial Conditions for Iterative Estimation Algorithms" on page 5-39.

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

---

9 In the **Covariance** drop-down list, select Estimate if you want the algorithm to compute parameter uncertainties. Effects of such uncertainties are displayed on plots as model confidence regions.

To omit estimating uncertainty, select None. Skipping uncertainty computation might reduce computation time for complex models and large data sets.

10 In the **Model Name** field, edit the name of the model or keep the default. The name of the model should be unique in the Model Board.

11 To view the estimation progress, select the **Display Progress** check box. This opens a progress viewer window in which the estimation progress is reported.

12 Click **Regularization** to obtain regularized estimates of model parameters. Specify the regularization constants in the Regularization Options dialog box. To learn more, see "Regularized Estimates of Model Parameters" on page 1-34.

13 Click **Estimate** to add this model to the Model Board in the System Identification app.

14 To stop the search and save the results after the current iteration has been completed, click **Stop Iterations**. To continue iterations from the current model, click the **Continue** button to assign current parameter values as initial guesses for the next search.

Assigning Estimation Weightings

You can specify how the estimation algorithm weighs the fit at various frequencies. In the app, set **Focus** to one of the following options:

- `Prediction` — Uses the inverse of the noise model H to weigh the relative importance of how closely to fit the data in various frequency ranges. Corresponds to minimizing one-step-ahead prediction, which typically favors the fit over a short time interval. Optimized for output prediction applications.
- `Simulation` — Uses the input spectrum to weigh the relative importance of the fit in a specific frequency range. Does not use the noise model to weigh the relative importance of how closely to fit the data in various frequency ranges. Optimized for output simulation applications.
- `Stability` — Behaves the same way as the `Prediction` option, but also forces the model to be stable. For more information about model stability, see "Unstable Models" on page 17-97.
- `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 2-96 or "Defining a Custom Filter" on page 2-96. This prefiltering applies only for estimating the dynamics from input to output. The disturbance model is determined from the estimation data.

Next Steps

- Validate the model by selecting the appropriate check box in the **Model Views** area of the System Identification app. For more information about validating models, see "Validating Models After Estimation" on page 17-2.
- Refine the model by clicking the **Value —> Initial Guess** button to assign current parameter values as initial guesses for the next search, edit the **Name** field, and click **Estimate**.
- Export the model to the MATLAB workspace for further analysis by dragging it to the **To Workspace** rectangle in the System Identification app.

See Also

Related Examples

- "Identify Low-Order Transfer Functions (Process Models) Using System Identification App"
- "Estimate Process Models at the Command Line" on page 5-8

Estimate Process 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 `frd` or `idfrd` objects. See "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-34. For supported data formats, see "Data Supported by Process Models" on page 5-3.
- 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 2-5.

Using procest to Estimate Process Models

You can estimate process models using the iterative estimation method `procest` that minimizes the prediction errors to obtain maximum likelihood estimates. The resulting models are stored as `idproc` model objects.

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

```
m = procest(data,mod_struc,opt)
```

`data` is the estimation data and `mod_struc` is one of the following:

- A character vector that represents the process model structure, as described in "Process Model Structure Specification" on page 5-36.
- A template `idproc` model. `opt` is an option set for configuring the estimation of the process model, such as handling of initial conditions, input offset and numerical search method.

Tip You do not need to construct the model object using `idproc` before estimation unless you want to specify initial parameter guesses, minimum/maximum bounds, or fixed parameter values, as described in "Estimate Process Models with Fixed Parameters" on page 5-10.

For more information about validating a process model, see "Validating Models After Estimation" on page 17-2.

You can use `procest` to refine parameter estimates of an existing process model, as described in "Refine Linear Parametric Models" on page 4-5.

For detailed information, see `procest` and `idproc`.

Estimate Process Models with Free Parameters

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

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

This process has two inputs and the response from each input is estimated by a first-order process model. All parameters are free to vary.

Load estimation data.

```
load co2data
```

Specify known sample time of 0.5 min.

```
Ts = 0.5;
```

Split data set into estimation data `ze` and validation data `zv`.

```
ze = iddata(Output_exp1,Input_exp1,Ts,...
                       'TimeUnit','min');
zv = iddata(Output_exp2,Input_exp2,Ts,...
                       'TimeUnit','min');
```

Estimate model with one pole, a delay, and a first-order disturbance component. The data contains known offsets. Specify them using the `InputOffset` and `OutputOffset` options.

```
opt = procestOptions;
opt.InputOffset = [170;50];
opt.OutputOffset = -45;
opt.Display = 'on';
opt.DisturbanceModel = 'arma1';
m = procest(ze,'p1d',opt)

m =

Process model with 2 inputs: y = G11(s)u1 + G12(s)u2
  From input "u1" to output "y1":
            Kp
  G11(s) = ---------- * exp(-Td*s)
           1+Tp1*s

        Kp = 2.6553
       Tp1 = 0.15515
        Td = 2.3175

  From input "u2" to output "y1":
            Kp
  G12(s) = ---------- * exp(-Td*s)
           1+Tp1*s

        Kp = 9.9756
       Tp1 = 2.0653
        Td = 4.9195

  An additive ARMA disturbance model exists for output "y1":
      y = G u + (C/D)e

      C(s) = s + 2.676
      D(s) = s + 0.6228

Parameterization:
    {'P1D'}    {'P1D'}
   Number of free coefficients: 8
```

```
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "ze".
Fit to estimation data: 91.07% (prediction focus)
FPE: 2.431, MSE: 2.412
```

Use dot notation to get the value of any model parameter. For example, get the value of dc gain parameter Kp .

```
m.Kp
```

```
ans = 1×2

    2.6553    9.9756
```

Estimate Process Models with Fixed Parameters

This example shows how to estimate a process model with fixed parameters.

When you know the values of certain parameters in the model and want to estimate only the values you do not know, you must specify the fixed parameters after creating the `idproc` model object. Use the following commands to prepare the data and construct a process model with one pole and a delay:

Load estimation data.

```
load co2data
```

Specify known sample time is 0.5 minutes.

```
Ts = 0.5;
```

Split data set into estimation data `ze` and validation data `zv`.

```
ze = iddata(Output_exp1,Input_exp1,Ts,...
                        'TimeUnit','min');
zv = iddata(Output_exp2,Input_exp2,Ts,...
                        'TimeUnit','min');
mod = idproc({'p1d','p1d'},'TimeUnit','min')
```

```
mod =

Process model with 2 inputs: y = G11(s)u1 + G12(s)u2
  From input 1 to output 1:
            Kp
  G11(s) = ---------- * exp(-Td*s)
           1+Tp1*s

        Kp = NaN
       Tp1 = NaN
        Td = NaN

  From input 2 to output 1:
```

```
            Kp
  G12(s) = ---------- * exp(-Td*s)
           1+Tp1*s

        Kp = NaN
       Tp1 = NaN
        Td = NaN

Parameterization:
    {'P1D'}    {'P1D'}
   Number of free coefficients: 6
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

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

Use the Structure model property to specify the initial guesses for unknown parameters, minimum/maximum parameter bounds and fix known parameters.

Set the value of Kp for the second transfer function to 10 and specify it as a fixed parameter. Initialize the delay values for the two transfer functions to 2 and 5 minutes, respectively. Specify them as free estimation parameters.

```
mod.Structure(2).Kp.Value = 10;
mod.Structure(2).Kp.Free = false;

mod.Structure(1).Tp1.Value = 2;
mod.Structure(2).Td.Value = 5;
```

Estimate Tp1 and Td only.

```
mod_proc = procest(ze,mod)

mod_proc =

Process model with 2 inputs: y = G11(s)u1 + G12(s)u2
  From input "u1" to output "y1":
            Kp
  G11(s) = ---------- * exp(-Td*s)
           1+Tp1*s

        Kp = -3.2213
       Tp1 = 2.1707
        Td = 4.44

  From input "u2" to output "y1":
            Kp
  G12(s) = ---------- * exp(-Td*s)
           1+Tp1*s

        Kp = 10
       Tp1 = 2.0764
        Td = 4.5205
```

```
Parameterization:
   {'P1D'}    {'P1D'}
  Number of free coefficients: 5
  Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "ze".
Fit to estimation data: 77.44%
FPE: 15.5, MSE: 15.39
```

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

See Also

`idproc` | `procest`

Related Examples

- "Building and Estimating Process Models Using System Identification Toolbox" on page 5-13
- "Estimate Process Models Using the App" on page 5-4
- "Loss Function and Model Quality Metrics" on page 1-46

Building and Estimating Process Models Using System Identification Toolbox

This example shows how to build simple process models using System Identification Toolbox™. Techniques for creating these models and estimating their parameters using experimental data is described. This example requires Simulink®.

Introduction

This example illustrates how to build simple process models often used in process industry. Simple, low-order continuous-time transfer functions are usually employed to describe process behavior. Such models are described by IDPROC objects which represent the transfer function in a pole-zero-gain form.

Process models are of the basic type 'Static Gain + Time Constant + Time Delay'. They may be represented as:

$$P(s) = K.e^{-T_d*s}.\frac{1 + T_z * s}{(1 + T_{p1} * s)(1 + T_{p2} * s)}$$

or as an integrating process:

$$P(s) = K.e^{-T_d*s}.\frac{1 + T_z * s}{s(1 + T_{p1} * s)(1 + T_{p2} * s)}$$

where the user can determine the number of real poles (0, 1, 2 or 3), as well as the presence of a zero in the numerator, the presence of an integrator term (`1/s`) and the presence of a time delay (`Td`). In addition, an underdamped (complex) pair of poles may replace the real poles.

Representation of Process Models Using IDPROC Objects

IDPROC objects define process models by using the letters `P` (for process model), `D` (for time delay), `Z` (for a zero) and `I` (for integrator). An integer will denote the number of poles. The models are generated by calling `idproc` with a character vector created using these letters.

For example:

```
idproc('P1') % transfer function with only one pole (no zeros or delay)
idproc('P2DIZ') % model with 2 poles, delay integrator and delay
idproc('P0ID') % model with no poles, but an integrator and a delay


ans =

Process model with transfer function:
             Kp
  G(s) = ----------
          1+Tp1*s

       Kp = NaN
      Tp1 = NaN

Parameterization:
```

```
   {'P1'}
   Number of free coefficients: 2
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.


ans =

Process model with transfer function:
                   1+Tz*s
  G(s) = Kp * ------------------ * exp(-Td*s)
              s(1+Tp1*s)(1+Tp2*s)

          Kp = NaN
         Tp1 = NaN
         Tp2 = NaN
          Td = NaN
          Tz = NaN

Parameterization:
    {'P2DIZ'}
   Number of free coefficients: 5
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.


ans =

Process model with transfer function:
          Kp
  G(s) = --- * exp(-Td*s)
           s

         Kp = NaN
         Td = NaN

Parameterization:
    {'P0DI'}
   Number of free coefficients: 2
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

Creating an IDPROC Object (Using a Simulink Model as Example)

Consider the system described by the following Simulink model:

```
open_system('iddempr1')
set_param('iddempr1/Random Number','seed','0')
```
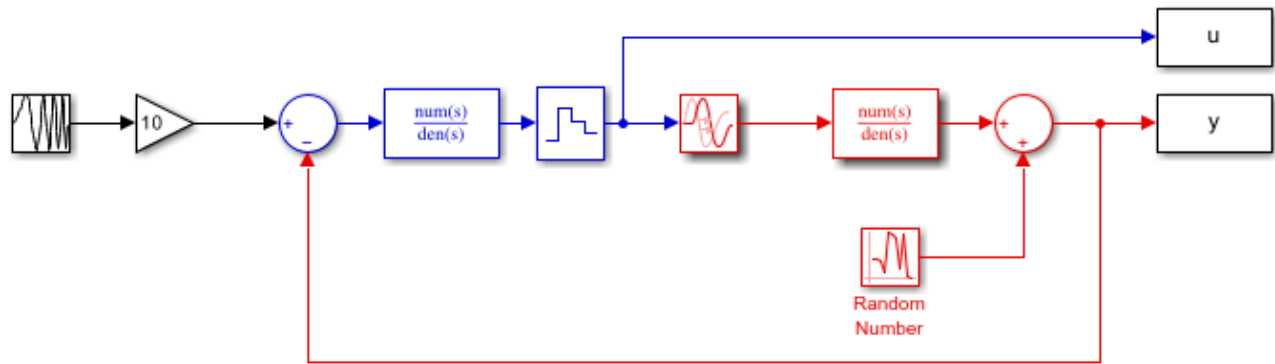
Red part: system to be modeled using IDPROC
Blue part: Controller

The red part is the system, the blue part is the controller and the reference signal is a swept sinusoid (a chirp signal). The data sample time is set to 0.5 seconds. As observed, the system is a continuous-time transfer function, and can hence be described using model objects in System Identification Toolbox, such as `idss`, `idpoly` or `idproc`.

Let us describe the system using `idpoly` and `idproc` objects. Using `idpoly` object, the system may be described as:

```
m0 = idpoly(1,0.1,1,1,[1 0.5],'Ts',0,'InputDelay',1.57,'NoiseVariance',0.01);
```

The IDPOLY form used above is useful for describing transfer functions of arbitrary orders. Since the system we are considering here is quite simple (one pole and no zeros), and is continuous-time, we may use the simpler IDPROC object to capture its dynamics:

```
m0p = idproc('p1d','Kp',0.2,'Tp1',2,'Td',1.57) % one pole+delay, with initial values
                                                % for gain, pole and delay specified.


m0p =

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

        Kp = 0.2
       Tp1 = 2
        Td = 1.57

Parameterization:
    {'P1D'}
   Number of free coefficients: 3
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

Estimating Parameters of IDPROC Models

Once a system is described by a model object, such as IDPROC, it may be used for estimation of its parameters using measurement data. As an example, we consider the problem of estimation of parameters of the Simulink model's system (red portion) using simulation data. We begin by acquiring data for estimation:

```
sim('iddempr1')
dat1e = iddata(y,u,0.5); % The IDDATA object for storing measurement data
```

Let us look at the data:

```
plot(dat1e)
```


Input-Output Data
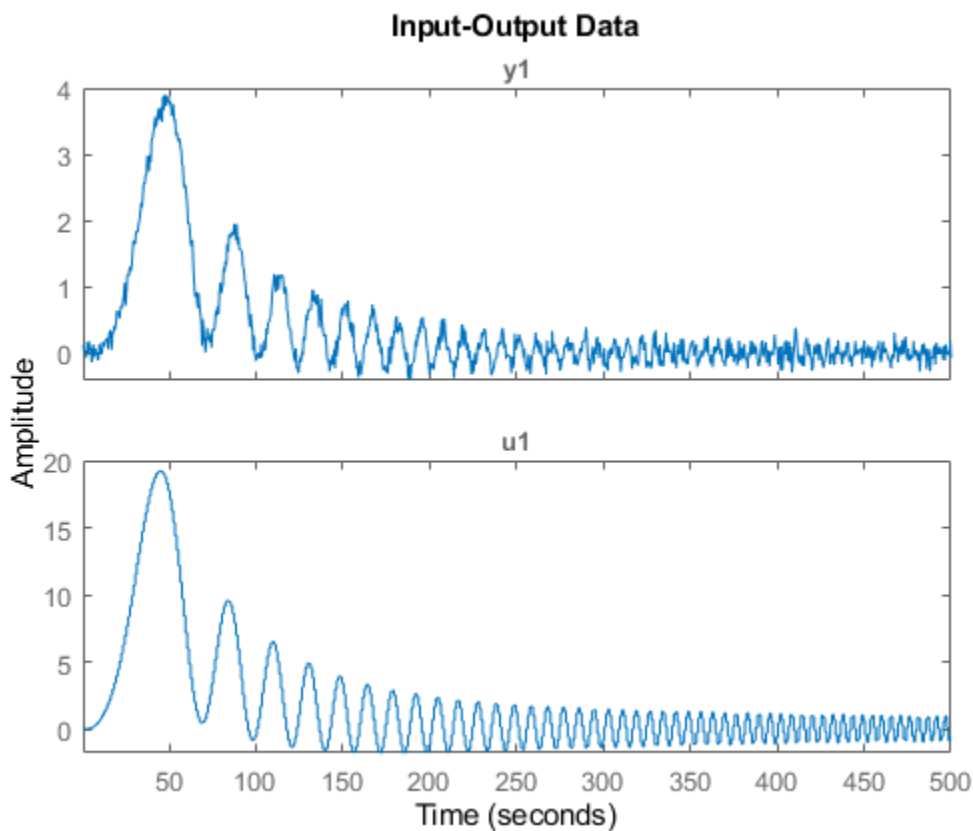
We can identify a process model using `procest` command, by providing the same structure information specified to create IDPROC models. For example, the 1-pole+delay model may be estimated by calling `procest` as follows:

```
m1 = procest(dat1e,'p1d'); % estimation of idproc model using data 'dat1e'.

% Check the result of estimation:
m1


m1 =

Process model with transfer function:
```

```
            Kp
  G(s) = ---------- * exp(-Td*s)
           1+Tp1*s

        Kp = 0.20045
       Tp1 = 2.0431
        Td = 1.499

Parameterization:
    {'P1D'}
   Number of free coefficients: 3
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "dat1e".
Fit to estimation data: 87.34%
FPE: 0.01069, MSE: 0.01062
```

To get information about uncertainties, use

`present(m1)`

```
m1 =

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

        Kp = 0.20045 +/- 0.00077275
       Tp1 = 2.0431 +/- 0.061216
        Td = 1.499 +/- 0.040854

Parameterization:
    {'P1D'}
   Number of free coefficients: 3
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Termination condition: Near (local) minimum, (norm(g) < tol)..
Number of iterations: 4, Number of function evaluations: 9

Estimated using PROCEST on time domain data "dat1e".
Fit to estimation data: 87.34%
FPE: 0.01069, MSE: 0.01062
More information in model's "Report" property.
```

The model parameters, K, Tp1 and Td are now shown with one standard deviation uncertainty range.

Computing Time and Frequency Response of IDPROC Models

The model m1 estimated above is an IDPROC model object to which all of the toolbox's model commands can be applied:

```
step(m1,m0) %step response of models m1 (estimated) and m0 (actual)
legend('m1 (estimated parameters)','m0 (known parameters)','location','northwest')
```

Step Response

From: u1 To: y1
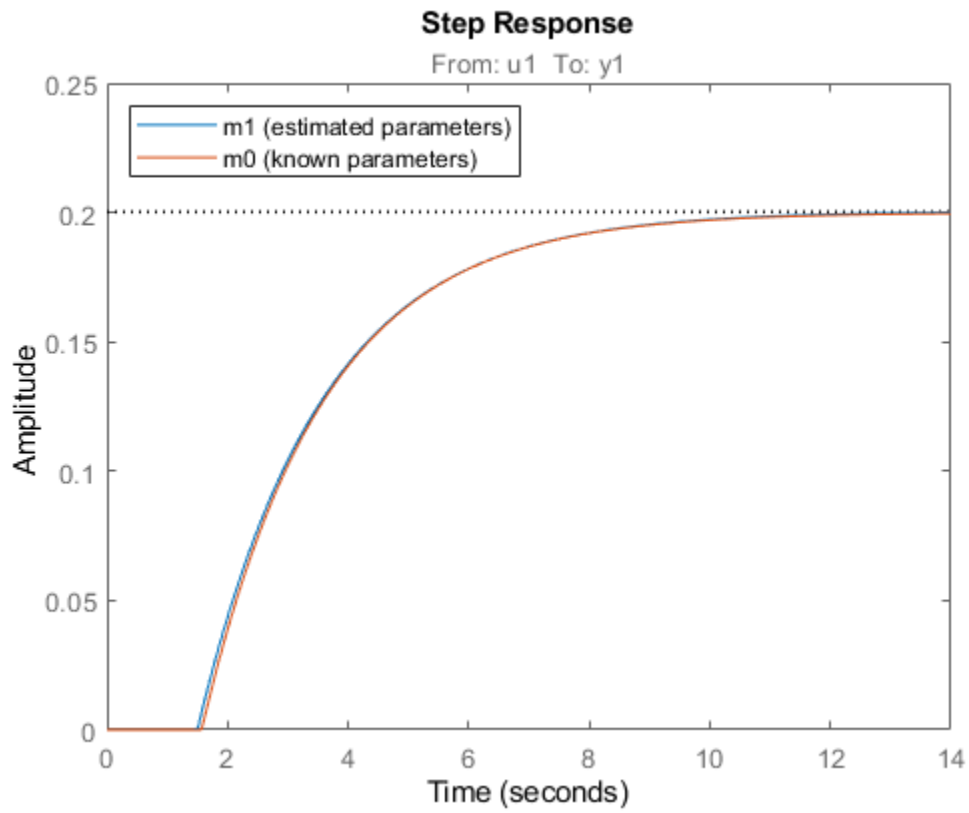
Bode response with confidence region corresponding to 3 standard deviations may be computed by doing:

```
h = bodeplot(m1,m0);
showConfidence(h,3)
```
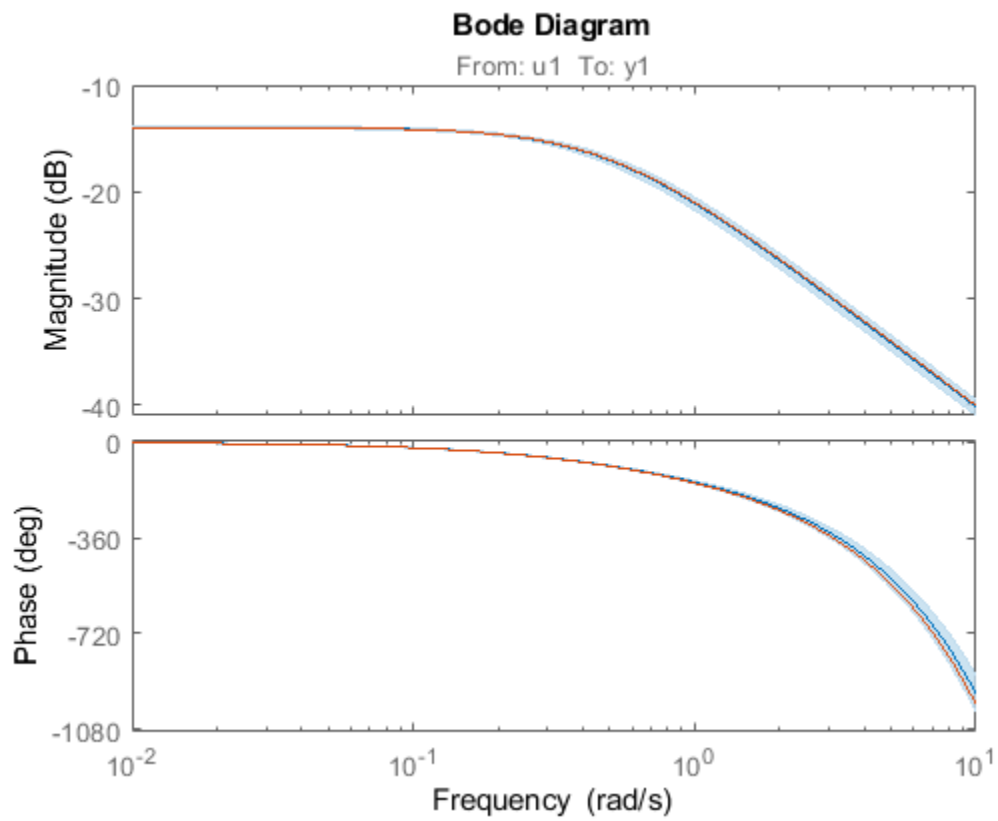
Similarly, the measurement data may be compared to the models outputs using `compare` as follows:

```
compare(dat1e,m0,m1)
```
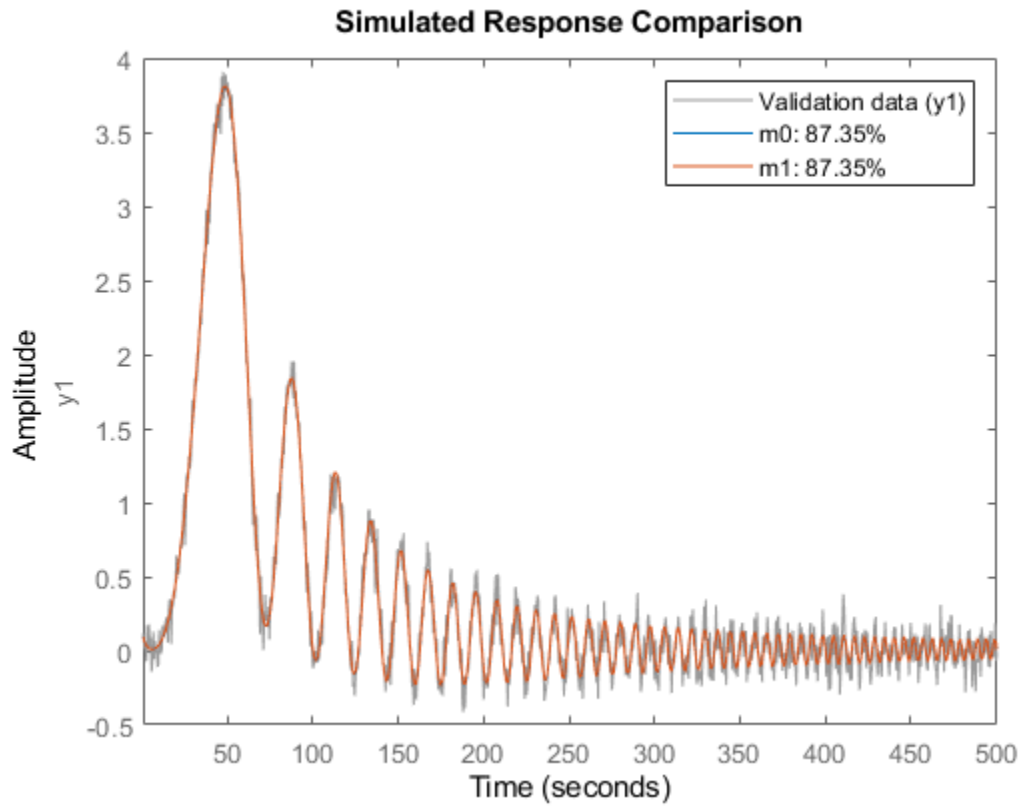
Other operations such as `sim`, `impulse`, `c2d` are also available, just as they are for other model objects.

```
bdclose('iddempr1')
```

Accommodating the Effect of Intersample Behavior in Estimation

It may be important (at least for slow sampling) to consider the intersample behavior of the input data. To illustrate this, let us study the same system as before, but without the sample-and-hold circuit:

```
open_system('iddempr5')
```
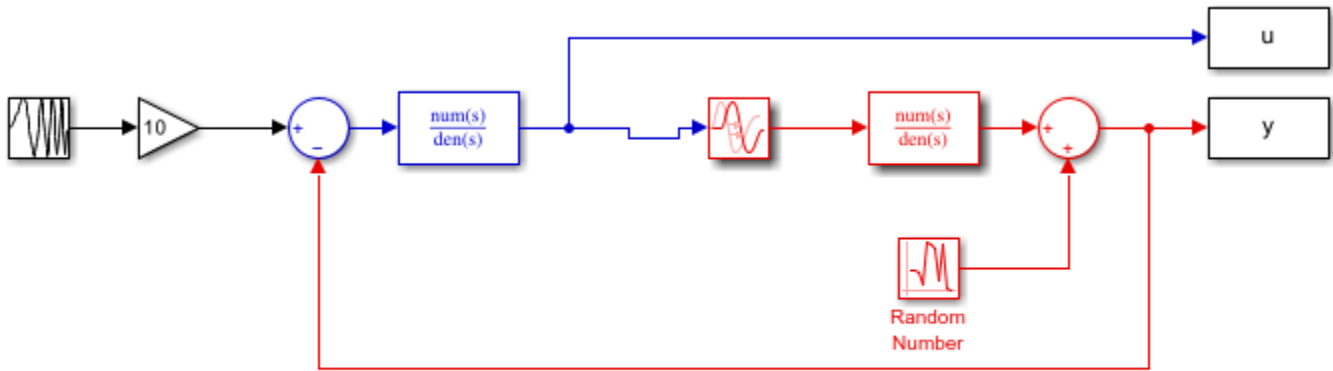
Red part: system to be modeled using IDPROC
Blue part: Controller

Simulate this system with the same sample time:

```
sim('iddempr5')
dat1f = iddata(y,u,0.5); % The IDDATA object for the simulated data
```

We estimate an IDPROC model using `data1f` while also imposing an upper bound on the allowable value delay. We will use 'lm' as search method and also choose to view the estimation progress.

```
m2_init = idproc('P1D');
m2_init.Structure.Td.Maximum = 2;
opt = procestOptions('SearchMethod','lm','Display','on');
m2 = procest(dat1f,m2_init,opt);
m2
```

```
m2 =

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

        Kp = 0.20038
       Tp1 = 2.01
        Td = 1.31

Parameterization:
    {'P1D'}
  Number of free coefficients: 3
  Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "dat1f".
Fit to estimation data: 87.26%
FPE: 0.01067, MSE: 0.01061
```

```
Process Model Identification

Estimation data: Time domain data dat1f
Data has 1 outputs, 1 inputs and 1001 samples.
Model Type:
    {'P1D'}
```

ESTIMATION PROGRESS

```
Initializing model parameters...
Initializing using polynomial estimator (BJ/OE)...
Initialization complete.

Algorithm: Levenberg-Marquardt search
-----------------------------------------------------------------------------
                     Norm of      First-order    Improvement (%)
Iteration    Cost    step         optimality     Expected  Achieved  Bisections
-----------------------------------------------------------------------------
    0     0.0106289      -            580          16.8         -         -
    1     0.0106099   0.0433          18.8         16.8       0.179       0
    2     0.0106099   0.000724       0.0282      0.0458    0.000486       0
    3     0.0106099   8.36e-06       5.21e-05    4.18e-07   4.46e-09      0
```

RESULT

```
Termination condition: Near (local) minimum, (norm(g) < tol)..
Number of iterations: 3, Number of function evaluations: 7

Status: Estimated using PROCEST
Fit to estimation data: 87.26%, FPE: 0.0106736
```

☐ Stop   Close

This model has a slightly less precise estimate of the delay than the previous one, m1:

```
[m0p.Td, m1.Td, m2.Td]
step(m0,m1,m2)
legend('m0 (actual)','m1 (estimated with ZOH)','m2 (estimated without ZOH)','location','southeast

ans =

   1.5700    1.4990    1.3100
```
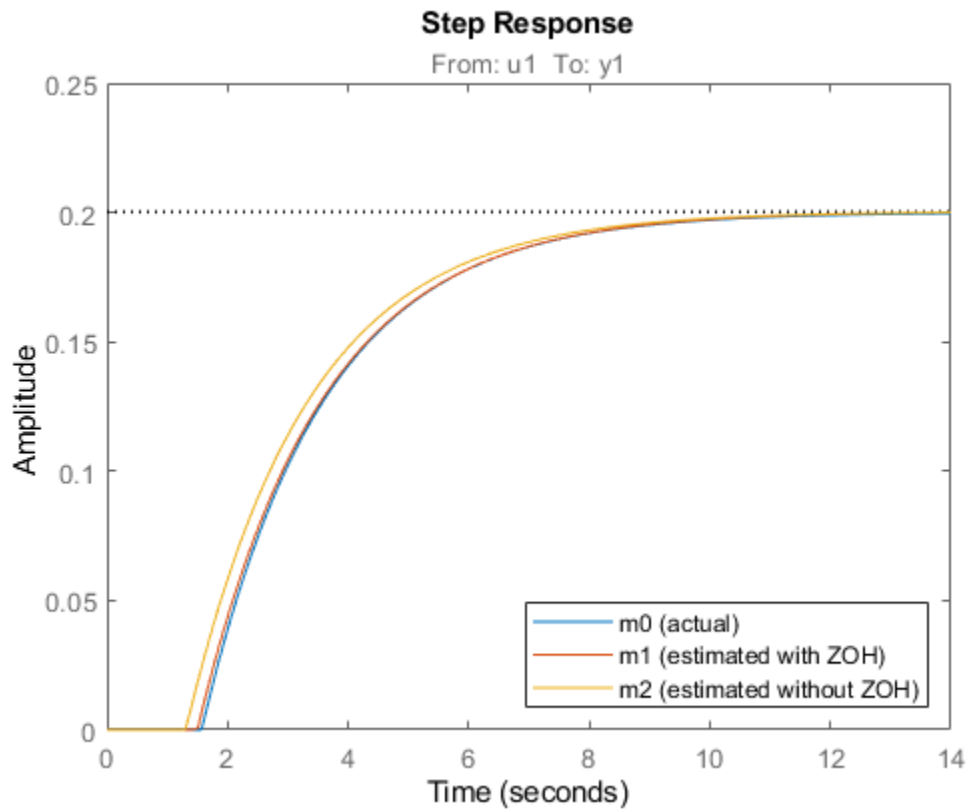
However, by telling the estimation process that the intersample behavior is first-order-hold (an approximation to the true continuous) input, we do better:

```
dat1f.InterSample = 'foh';
m3 = procest(dat1f,m2_init,opt);
```

Compare the four models m0 (true) m1 (obtained from zoh input) m2 (obtained for continuous input, with zoh assumption) and m3 (obtained for the same input, but with foh assumption)

```
[m0p.Td, m1.Td, m2.Td, m3.Td]
compare(dat1e,m0,m1,m2,m3)
```

```
ans =

   1.5700   1.4990   1.3100   1.5570
```
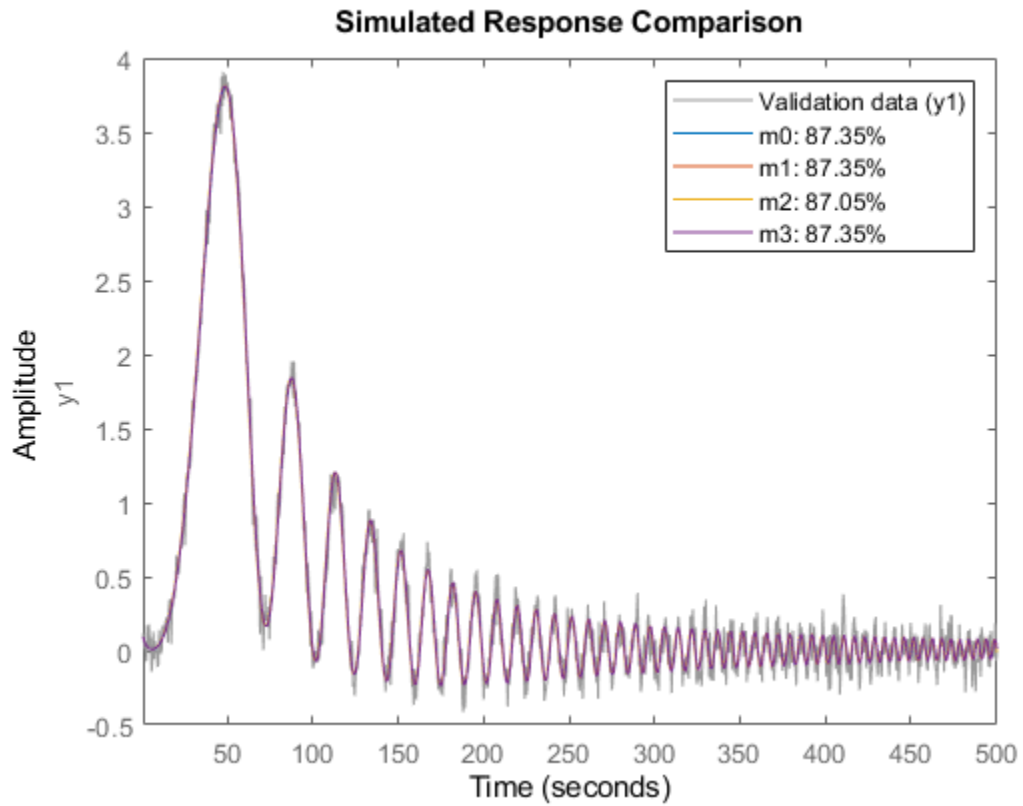
```
step(m0,m1,m2,m3)
legend('m0','m1','m2','m3')
bdclose('iddempr5')
```
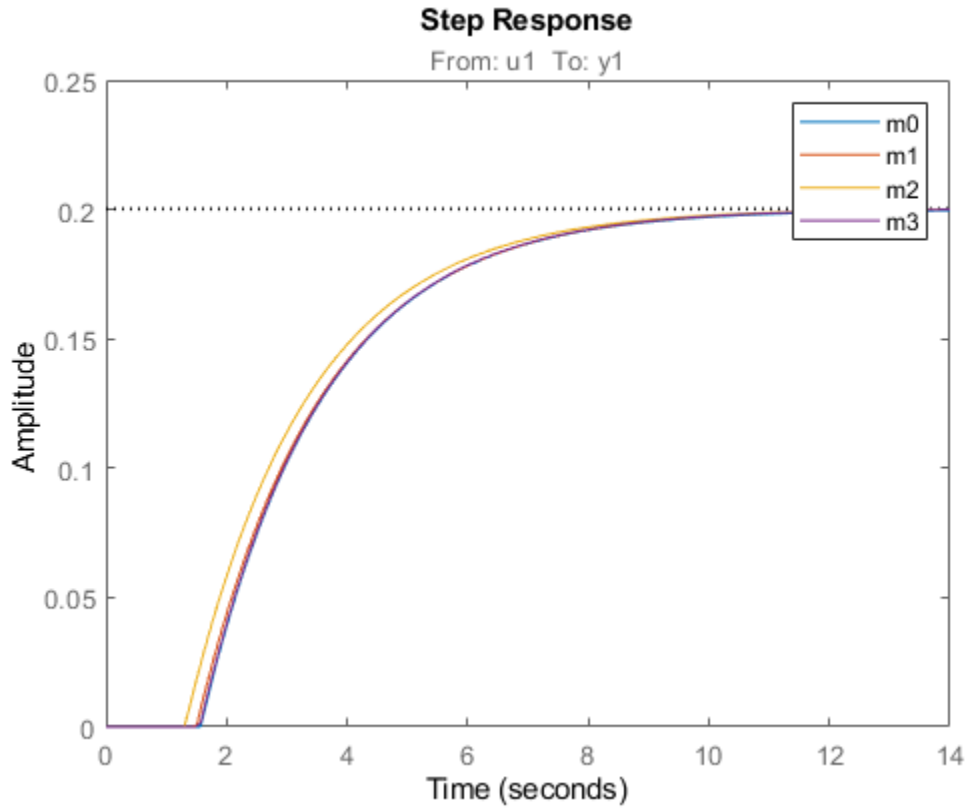
Modeling a System Operating in Closed Loop

Let us now consider a more complex process, with integration, that is operated in closed loop:

```
open_system('iddempr2')
```
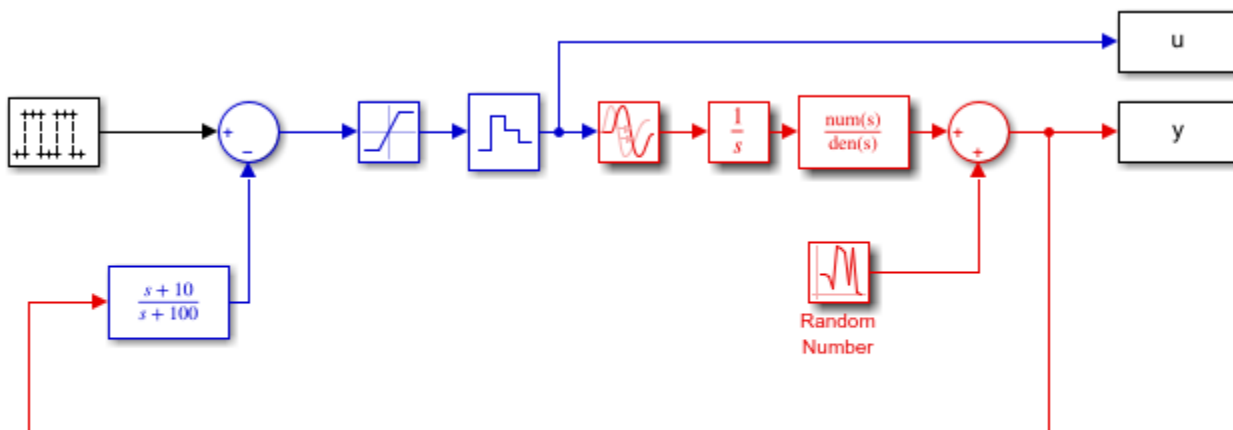

Red part: system to be modeled using IDPROC
Blue part: Controller

The true system can be represented by:

```
m0 = idproc('P2ZDI','Kp',1,'Tp1',1,'Tp2',5,'Tz',3,'Td',2.2);
```

The process is controlled by a PD regulator with limited input amplitude and a zero order hold device. The sample time is 1 second.

```
set_param('iddempr2/Random Number','seed','0')
sim('iddempr2')
dat2 = iddata(y,u,1); % IDDATA object for estimation
```

Two different simulations are made, the first for estimation and the second one for validation purposes.

```
set_param('iddempr2/Random Number','seed','13')
sim('iddempr2')
dat2v = iddata(y,u,1); % IDDATA object for validation purpose
```

Let us look at the data (estimation and validation).

```
plot(dat2,dat2v)
legend('dat2 (estimation)','dat2v (validation)')
```
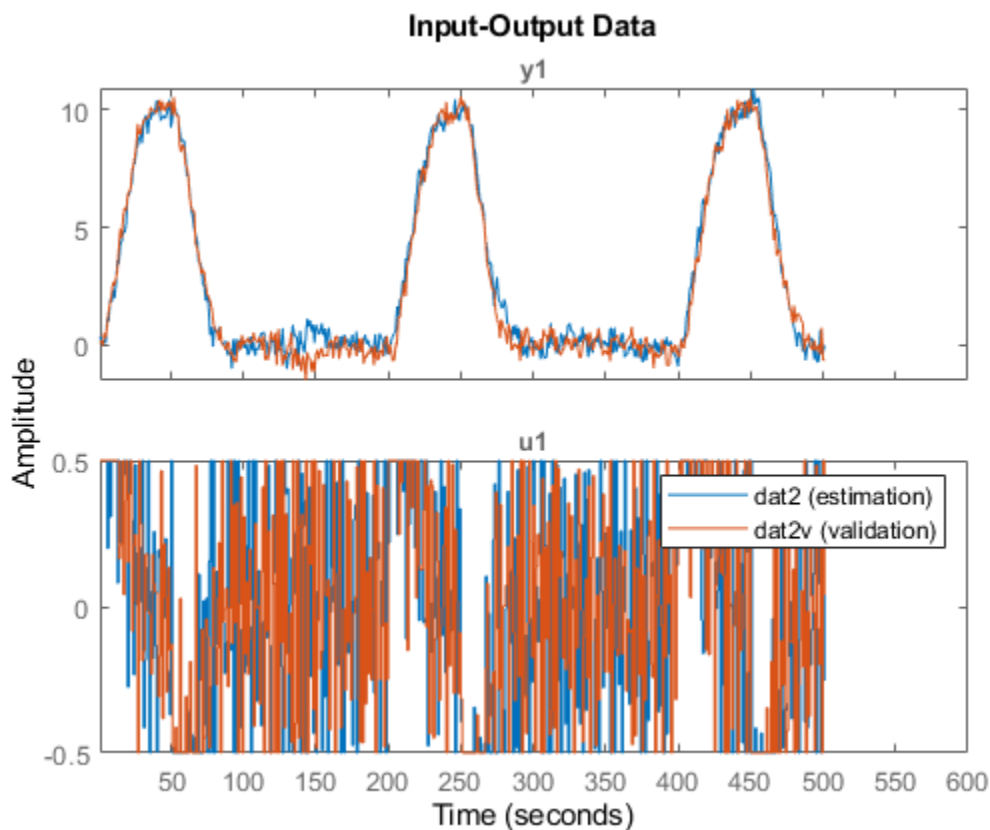

Let us now perform estimation using `dat2`.

```
Warn = warning('off','Ident:estimation:underdampedIDPROC');
m2_init = idproc('P2ZDI');
m2_init.Structure.Td.Maximum = 5;
m2_init.Structure.Tp1.Maximum = 2;
opt = procestOptions('SearchMethod','lsqnonlin','Display','on');
```

```
opt.SearchOptions.MaxIterations  = 100;
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 = 0.98412
        Tp1 = 2
        Tp2 = 1.4838
         Td = 1.713
         Tz = 0.027244

Parameterization:
    {'P2DIZ'}
   Number of free coefficients: 5
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "dat2".
Fit to estimation data: 91.51%
FPE: 0.1128, MSE: 0.1092


compare(dat2v,m2,m0) % Gives very good agreement with data
```

Simulated Response Comparison
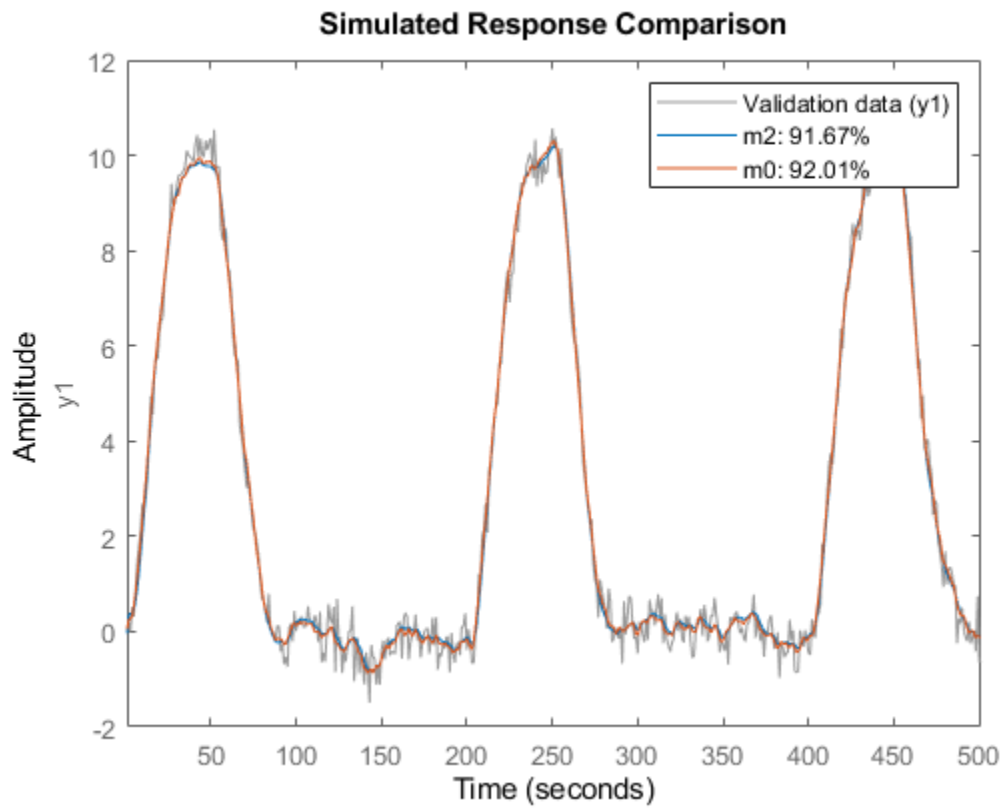

```
bode(m2,m0)
legend({'m2 (est)','m0 (actual)'},'location','west')
```

```
impulse(m2,m0)
legend({'m2 (est)','m0 (actual)'})
```

Impulse Response
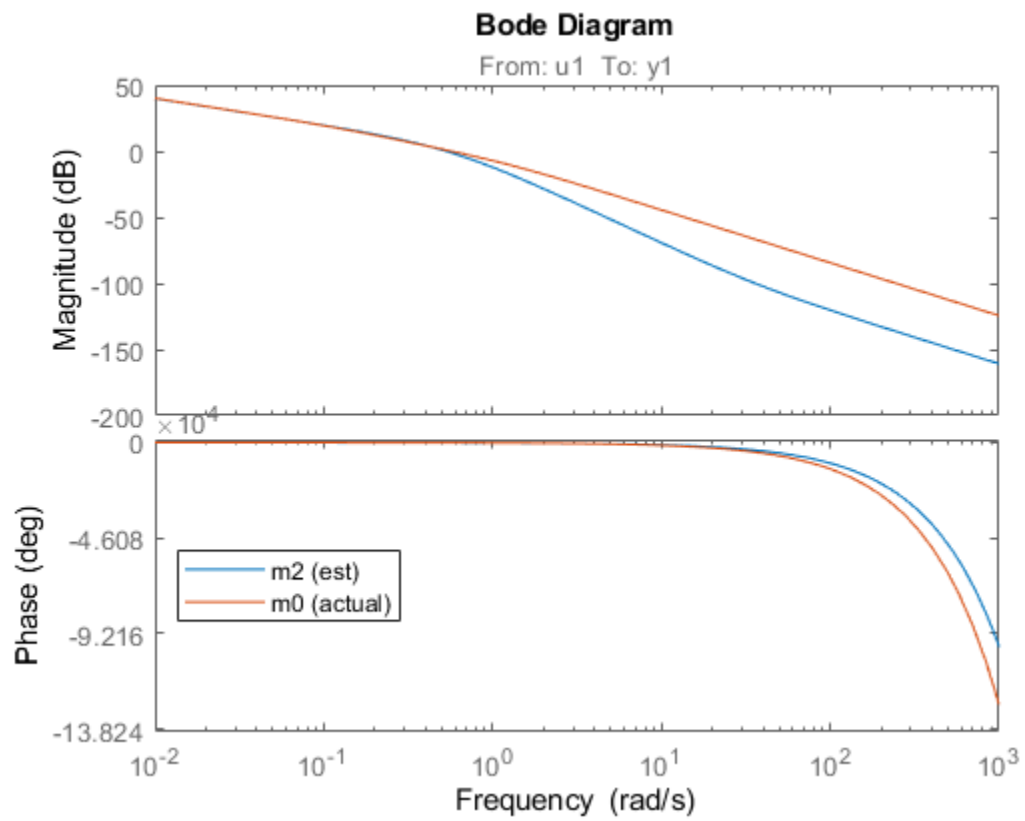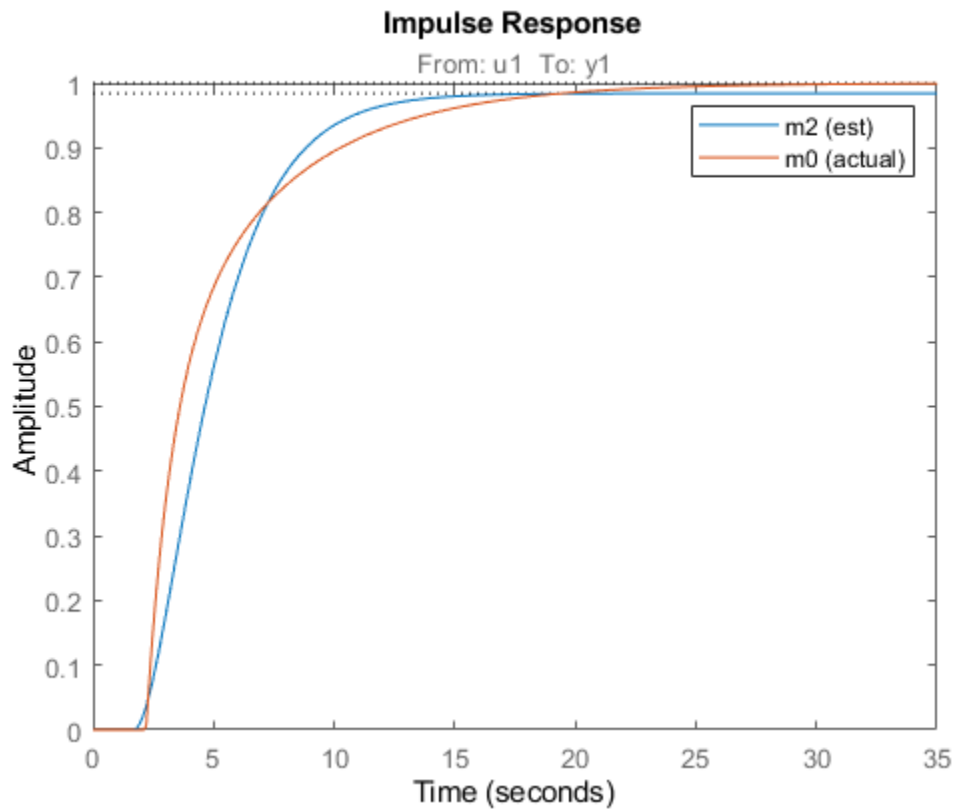
Compare also with the parameters of the true system:

```
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 = 0.98412 +/- 0.013672
        Tp1 = 2 +/- 8.2231
        Tp2 = 1.4838 +/- 10.193
         Td = 1.713 +/- 63.703
         Tz = 0.027244 +/- 65.516

Parameterization:
    {'P2DIZ'}
   Number of free coefficients: 5
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Termination condition: Change in cost was less than the specified tolerance..
Number of iterations: 3, Number of function evaluations: 4
```

```
Estimated using PROCEST on time domain data "dat2".
Fit to estimation data: 91.51%
FPE: 0.1128, MSE: 0.1092
More information in model's "Report" property.


ans =

    1.0000    0.9841
    1.0000    2.0000
    5.0000    1.4838
    2.2000    1.7130
    3.0000    0.0272
```

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 = 0.95747
        Tp1 = 1.999
        Tp2 = 0.60819
         Td = 2.314
         Tz = 0.0010561

Parameterization:
    {'P2DIZ'}
   Number of free coefficients: 5
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "dat2v".
Fit to estimation data: 90.65%
FPE: 0.1397, MSE: 0.1353


ans =

    1.0000    0.9841    0.9575
    1.0000    2.0000    1.9990
    5.0000    1.4838    0.6082
    2.2000    1.7130    2.3140
    3.0000    0.0272    0.0011
```

This model has much worse parameter values. On the other hand, it performs nearly identically to the true system m0 when tested on the other data set dat2:

```
compare(dat2,m0,m2,m2v)
```
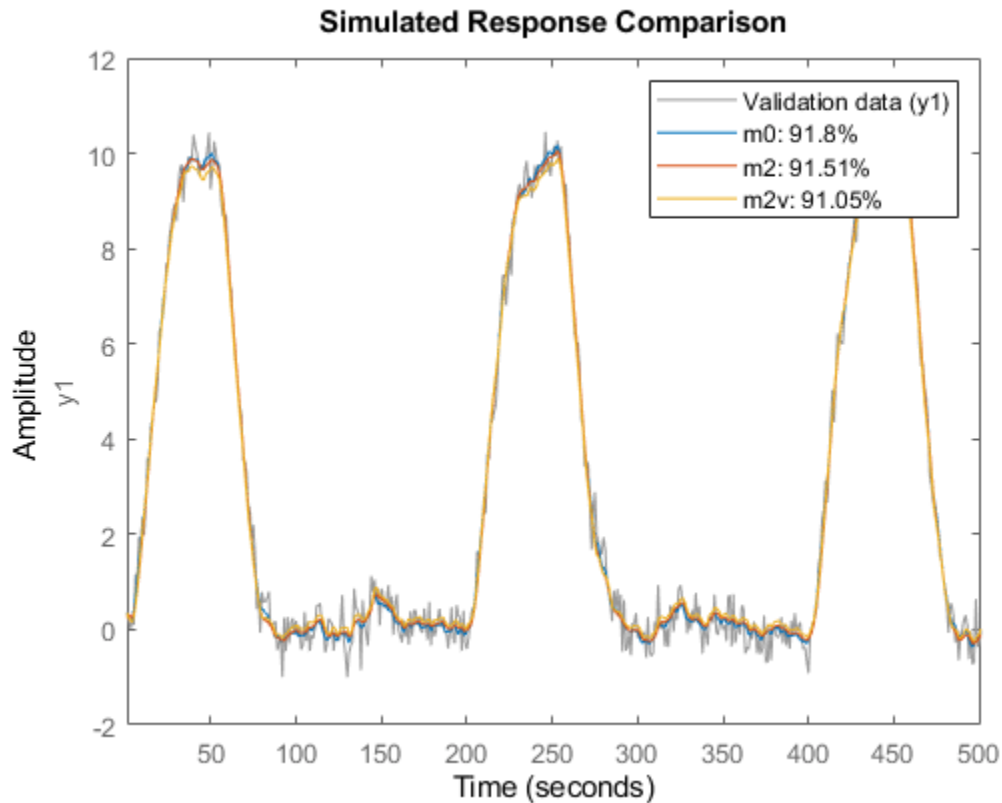

Fixing Known Parameters During Estimation

Suppose we know from other sources that one time constant is 1:

```
m2v.Structure.Tp1.Value = 1;
m2v.Structure.Tp1.Free = false;
```

We can fix this value, while estimating the other parameters:

```
m2v = procest(dat2v,m2v)
%

m2v =

Process model with transfer function:
                    1+Tz*s
  G(s) = Kp * ------------------- * exp(-Td*s)
               s(1+Tp1*s)(1+Tp2*s)

         Kp = 1.0111
        Tp1 = 1
        Tp2 = 5.3014
```

```
        Td = 2.195
        Tz = 3.231

Parameterization:
   {'P2DIZ'}
   Number of free coefficients: 4
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "dat2v".
Fit to estimation data: 92.05%
FPE: 0.09952, MSE: 0.09794
```

As observed, fixing `Tp1` to its known value dramatically improves the estimates of the remaining parameters in model `m2v`.

This also indicates that simple approximation should do well on the data:

```
m1x_init = idproc('P2D'); % simpler structure (no zero, no integrator)
m1x_init.Structure.Td.Maximum = 2;
m1x = procest(dat2v, m1x_init)
compare(dat2,m0,m2,m2v,m1x)


m1x =

Process model with transfer function:
                Kp
  G(s) = ----------------- * exp(-Td*s)
         (1+Tp1*s)(1+Tp2*s)

         Kp = -1.2555
        Tp1 = 1.0249e-06
        Tp2 = 0.078021
         Td = 1.959

Parameterization:
   {'P2D'}
   Number of free coefficients: 4
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "dat2v".
Fit to estimation data: -23.87%
FPE: 24.15, MSE: 23.77
```

Simulated Response Comparison
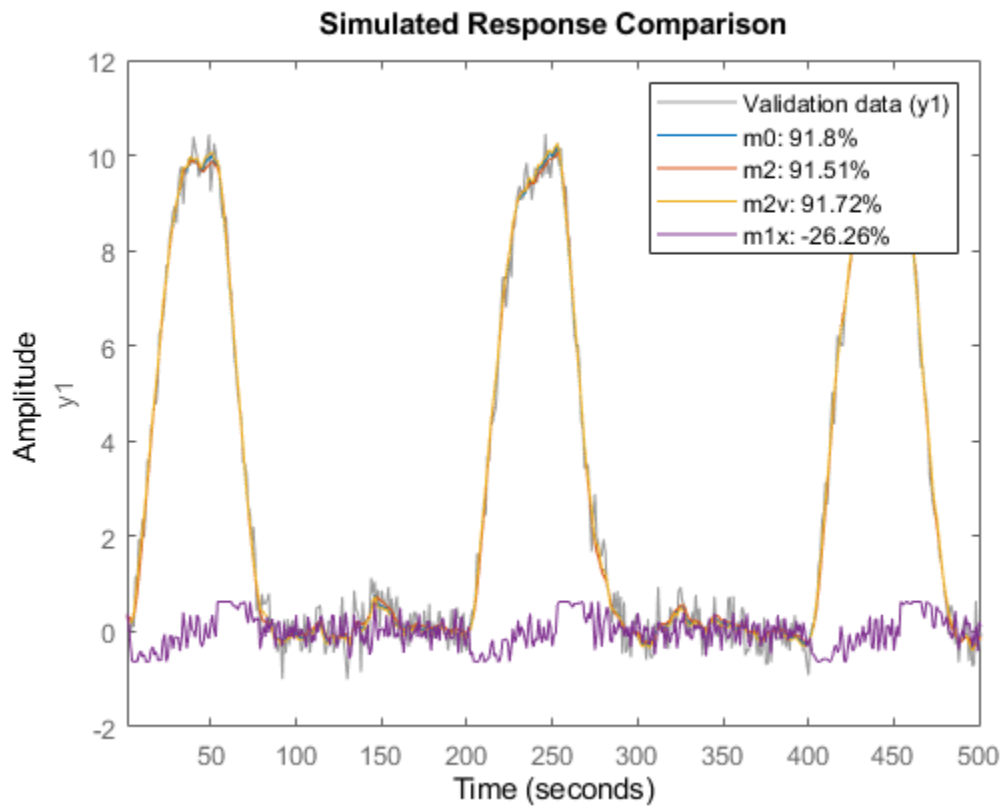
Thus, the simpler model is able to estimate system output pretty well. However, `m1x` does not contain any integration, so the open loop long time range behavior will be quite different:

```
step(m0,m2,m2v,m1x)
legend('m0','m2','m2v','m1x')
bdclose('iddempr2')
warning(Warn)
```
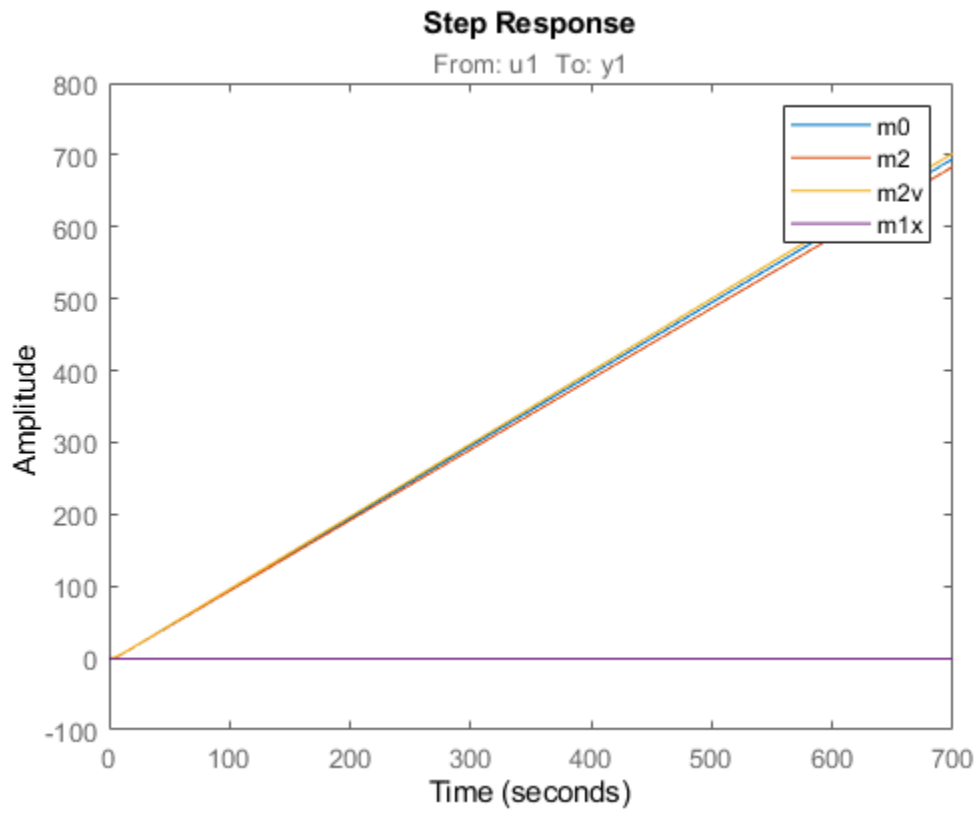
Process Model Structure Specification

This topic describes how to specify the model structure in the estimation procedures "Estimate Process Models Using the App" on page 5-4 and "Estimate Process Models at the Command Line" on page 5-8.

In the System Identification app, specify the model structure by selecting the number of real or complex poles, and whether to include a zero, delay, and integrator. The resulting transfer function is displayed in the Process Models dialog box.

At the command line, specify the model structure using an acronym that includes the following letters and numbers:

- (Required) P for a process model
- (Required) 0, 1, 2 or 3 for the number of poles
- (Optional) D to include a time-delay term e^{-sT_d}
- (Optional) Z to include a process zero (numerator term)
- (Optional) U to indicate possible complex-valued (underdamped) poles
- (Optional) I to indicate enforced integration

Typically, you specify the model-structure acronym as an argument in the estimation command procest:

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

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

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

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

- 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 "Estimate Process Models at the Command Line" on page 5-8.

See Also

More About

- "What Is a Process Model?" on page 5-2

Estimating Multiple-Input, Multi-Output Process Models

If your model contains multiple inputs, multiple outputs, or both, you can specify whether to estimate the same transfer function for all input-output pairs, or a different transfer function for each. The information in this section supports the estimation procedures "Estimate Process Models Using the App" on page 5-4 and "Estimate Process Models at the Command Line" on page 5-8.

In the System Identification app — To fit a data set with multiple inputs, or multiple outputs, or both, in the Process Models dialog box, configure the process model settings for one input-output pair at a time. Use the input and output selection lists to switch to a different input/output pair.

If you want the same transfer function to apply to all input/output pairs, select the **All same** check box. To apply a different structure to each channel, leave this check box clear, and create a different transfer function for each input.

At the command line — Specify the model structure as a cell array of character vectors in the estimation command `procest`. For example, use this command to specify the first-order transfer function for the first input, and a second-order model with a zero and an integrator for the second input:

```
m = idproc({'P1','P2ZI'})
m = procest(data,m)
```

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

See Also

More About

- "Data Supported by Process Models" on page 5-3

Disturbance Model Structure for Process Models

This section describes how to specify a noise model in the estimation procedures "Estimate Process Models Using the App" on page 5-4 and "Estimate Process Models at the Command Line" on page 5-8.

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

$$y = Gu + He$$

where *e* is white noise.

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

In the System Identification app, to specify whether to include or exclude a noise model in the Process Models dialog box, select one of the following options from the **Disturbance Model** list:

- `None` — The algorithm does not estimate a noise model (*C=D*=1). This option also sets **Focus** to `Simulation`.
- `Order 1` — Estimates a noise model as a continuous-time, first-order ARMA model.
- `Order 2` — Estimates a noise model as a continuous-time, second-order ARMA model.

At the command line, specify the disturbance model using the `procestOptions` option set. For example, use this command to estimate a first-order transfer function and a first-order noise model:

```
opt = procestOptions;
opt.DisturbanceModel = 'arma1';
model = procest(data, 'P1D', opt);
```

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

See Also
`procestOptions`

More About
- "Estimate Process Models Using the App" on page 5-4
- "Estimate Process Models at the Command Line" on page 5-8

Specifying Initial Conditions for Iterative Estimation Algorithms

You can optionally specify how the iterative algorithm treats initial conditions for estimation of model parameters. This information supports the estimation procedures "Estimate Process Models Using the App" on page 5-4 and "Estimate Process Models at the Command Line" on page 5-8.

In the System Identification app, set **Initial condition** to one of the following options:

- `Zero` — Sets all initial states to zero.
- `Estimate` — Treats the initial states as an unknown vector of parameters and estimates these states from the data.
- `Backcast` — Estimates initial states using a backward filtering method (least-squares fit).
- `U-level est` — Estimates both the initial conditions and input offset levels. For multiple inputs, the input level for each input is estimated individually. Use if you included an integrator in the transfer function.
- `Auto` — Automatically chooses one of the preceding options based on the estimation data. If the initial conditions have negligible effect on the prediction errors, they are taken to be zero to optimize algorithm performance.

At the command line, specify the initial conditions using the `InitialCondition` model estimation option, configured using the `procestOptions` command. For example, use this command to estimate a first-order transfer function and set the initial states to zero:

```
opt = procestOptions('InitialCondition','zero');
model = procest(data,'P1D',opt)
```

See Also
`procestOptions`

More About
- "Estimate Process Models Using the App" on page 5-4
- "Estimate Process Models at the Command Line" on page 5-8

6

Identifying Input-Output Polynomial Models

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:

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

The variables A, B, C, D, and F are polynomials expressed in the time-shift operator q^-1. u_i is the ith input, nu is the total number of inputs, and nk_i is the ith input delay that characterizes the transport delay. The variance of the white noise $e(t)$ is assumed to be λ. For more information about the time-shift operator, see "Understanding the Time-Shift Operator q" on page 6-2.

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:

$$A(q)y(t) = \sum_{i=1}^{nu} \frac{B_i(q)}{F_i(q)} u_i(t - nk_i) + \frac{C(q)}{D(q)} \frac{1}{1 - q^{-1}} e(t)$$

For more information, see "Different Configurations of Polynomial Models" on page 6-3.

You can estimate polynomial models using time or frequency domain data.

For estimation, you must specify the *model order* as a set of integers that represent the number of coefficients for each polynomial you include in your selected structure—*na* for A, *nb* for B, *nc* for C, *nd* for D, and *nf* for F. You must also specify the number of samples *nk* corresponding to the input delay—*dead time*—given by the number of samples before the output responds to the input.

The number of coefficients in denominator polynomials is equal to the number of poles, and the number of coefficients in the numerator polynomials is equal to the number of zeros plus 1. When the dynamics from $u(t)$ to $y(t)$ contain a delay of *nk* samples, then the first *nk* coefficients of B are zero.

For more information about the family of transfer-function models, see the corresponding section in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

Understanding the Time-Shift Operator q

The general polynomial equation is written in terms of the time-shift operator q^{-1}. To understand this time-shift operator, consider the following discrete-time difference equation:

$$y(t) + a_1 y(t - T) + a_2 y(t - 2T) =$$
$$b_1 u(t - T) + b_2 u(t - 2T)$$

where $y(t)$ is the output, $u(t)$ is the input, and T is the sample time. q^{-1} is a time-shift operator that compactly represents such difference equations using $q^{-1} u(t) = u(t - T)$:

$$y(t) + a_1 q^{-1} y(t) + a_2 q^{-2} y(t) =$$
$$b_1 q^{-1} u(t) + b_2 q^{-2} u(t)$$

or

$$A(q)y(t) = B(q)u(t)$$

In this case, $A(q) = 1 + a_1 q^{-1} + a_2 q^{-2}$ and $B(q) = b_1 q^{-1} + b_2 q^{-2}$.

Note This q description is completely equivalent to the Z-transform form: q corresponds to z.

Different Configurations of Polynomial Models

These model structures are subsets of the following general polynomial equation:

$$A(q)y(t) = \sum_{i=1}^{nu} \frac{B_i(q)}{F_i(q)} u_i(t - nk_i) + \frac{C(q)}{D(q)} e(t)$$

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 ratios.
ARIX	$$Ay = Bu + \frac{1}{1 - q^{-1}} e$$	Extends the ARX structure by including an integrator in the noise source, $e(t)$. This is useful in cases where the disturbance is not stationary.
ARMAX	$$A(q)y(t) = \sum_{i=1}^{nu} B_i(q)u_i(t - nk_i) + C(q)e(t)$$	Extends the ARX structure by providing more flexibility for modeling noise using the C parameters (a moving average of white noise). Use ARMAX when the dominating disturbances enter at the input. Such disturbances are called *load disturbances*.

Model Structure	Equation	Description
ARIMAX	$Ay = Bu + C\dfrac{1}{1 - q^{-1}}e$	Extends the ARMAX structure by including an integrator in the noise source, $e(t)$. This is useful in cases where the disturbance is not stationary.
Box-Jenkins (BJ)	$y(t) = \displaystyle\sum_{i=1}^{nu} \dfrac{B_i(q)}{F_i(q)}u_i(t - nk_i) + \dfrac{C(q)}{D(q)}e(t)$	Provides completely independent parameterization for the dynamics and the noise using rational polynomial functions. Use BJ models when the noise does not enter at the input, but is primary a measurement disturbance, This structure provides additional flexibility for modeling noise.
Output-Error (OE)	$y(t) = \displaystyle\sum_{i=1}^{nu} \dfrac{B_i(q)}{F_i(q)}u_i(t - nk_i) + e(t)$	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 app supports direct estimation of ARX, ARMAX, OE and BJ models. You can add a noise integrator to the ARX, ARMAX and BJ forms. However, you can use `polyest` to estimate all five polynomial or any subset of polynomials in the general equation. For more information about working with pem, see "Using polyest to Estimate Polynomial Models" on page 6-19.

Continuous-Time Representation of Polynomial Models

In continuous time, the general frequency-domain equation is written in terms of the Laplace transform variable s, which corresponds to a differentiation operation:

$$A(s)Y(s) = \frac{B(s)}{F(s)}U(s) + \frac{C(s)}{D(s)}E(s)$$

In the continuous-time case, the underlying time-domain model is a differential equation and the model order integers represent the number of estimated numerator and denominator coefficients. For example, $n_a=3$ and $n_b=2$ correspond to the following model:

$$A(s) = s^4 + a_1 s^3 + a_2 s^2 + a_3$$
$$B(s) = b_1 s + b_2$$

You can only estimate continuous-time polynomial models directly using continuous-time frequency-domain data. In this case, you must set the `Ts` data property to 0 to indicate that you have continuous-time frequency-domain data, and use the `oe` command to estimate an Output-Error

polynomial model. Continuous-time models of other structures such as ARMAX or BJ cannot be estimated. You can obtain those forms only by direct construction (using `idpoly`), conversion from other model types, or by converting a discrete-time model into continuous-time (`d2c`). Note that the OE form represents a transfer function expressed as a ratio of numerator (*B*) and denominator (*F*) polynomials. For such forms consider using the transfer function models, represented by `idtf` models. You can estimate transfer function models using both time and frequency domain data. In addition to the numerator and denominator polynomials, you can also estimate transport delays. See `idtf` and `tfest` for more information.

Multi-Output Polynomial Models

For a MIMO polynomial model with *ny* outputs and *nu* inputs, the relation between inputs and outputs for the *l*th output can be written as:

$$\sum_{j=1}^{ny} A_{lj}(q)y_j(t) = \sum_{i=1}^{nu} \frac{B_{li}(q)}{F_{li}(q)}u_i(t - nk_i) + \frac{C_l(q)}{D_l(q)}e_l(t)$$

The *A* polynomial array (A_{ij}; *i*=1:*ny*, *j*=1:*ny*) are stored in the `A` property of the `idpoly` object. The diagonal polynomials (A_{ii}; *i*=1:*ny*) are monic, that is, the leading coefficients are one. The off-diagonal polynomials (A_{ij}; $i \neq j$) contain a delay of at least one sample, that is, they start with zero. For more details on the orders of multi-output models, see "Polynomial Sizes and Orders of Multi-Output Polynomial Models" on page 6-21.

You can create multi-output polynomial models by using the `idpoly` command or estimate them using `ar`, `arx`, `bj`, `oe`, `armax`, and `polyest`. In the app, you can estimate such models by choosing a multi-output data set and setting the orders appropriately in the **Polynomial Models** dialog box.

See Also
`idpoly` | `ar` | `arx` | `bj` | `oe` | `armax` | `polyest`

Related Examples
- "Estimate Polynomial Models in the App" on page 6-14
- "Estimate Polynomial Models at the Command Line" on page 6-18

More About
- "Data Supported by Polynomial Models" on page 6-6

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 Analysis".
- Real data or complex data in any domain.
- Single-output and multiple-output.

You must import your data into the MATLAB workspace, as described in "Data Preparation".

Designating Data for Estimating Continuous-Time Models

To get a linear, continuous-time model of arbitrary structure for time-domain data, you can estimate a discrete-time model, and then use `d2c` to transform it to a continuous-time model.

For continuous-time frequency-domain data, you can estimate directly only Output-Error (OE) continuous-time models. Other structures include noise models, which is not supported for frequency-domain data.

> **Tip** To denote continuous-time frequency-domain data, set the data sample time to 0. You can set the sample time when you import data into the app or set the `Ts` property of the data object at the command line.

Designating Data for Estimating Discrete-Time Models

You can estimate arbitrary-order, linear state-space models for both time- or frequency-domain data.

Set the data property `Ts` to:

- `0`, for frequency response data that is measured directly from an experiment.
- Equal to the `Ts` of the original data, for frequency response data obtained by transforming time-domain `iddata` (using `spa` and `etfe`).

> **Tip** You can set the sample time when you import data into the app or set the `Ts` property of the data object at the command line.

See Also

Related Examples

- "Estimate Polynomial Models in the App" on page 6-14
- "Estimate Polynomial Models at the Command Line" on page 6-18

More About

- "What Are Polynomial Models?" on page 6-2

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 topics on the "Residual Analysis" page.

Estimating Orders and Delays in the App

The following procedure assumes that you have already imported your data into the app and performed any necessary preprocessing operations. For more information, see "Represent Data".

To estimate model orders and input delays in the System Identification app:

1   In the System Identification app, select **Estimate** > **Polynomial Models** to open the Polynomials Models dialog box.

    The ARX model is already selected by default in the **Structure** list.

    Note For time-series models, select the AR model structure.

2   Edit the **Orders** field to specify a range of poles, zeros, and delays. For example, enter the following values for *na*, *nb*, and *nk*:

    `[1:10 1:10 1:10]`

    Tip As a shortcut for entering `1:10` for each required model order, click **Order Selection**.
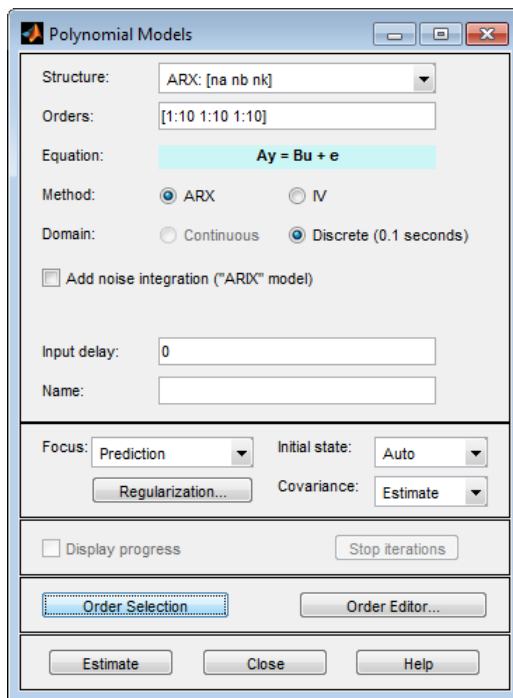
3   Click **Estimate** to open the ARX Model Structure Selection window, which displays the model performance for each combination of model parameters. The following figure shows an example plot.
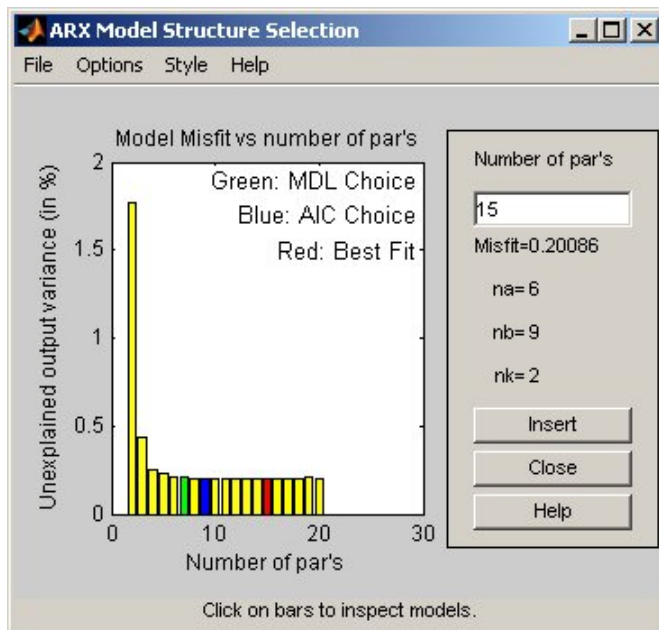
4   Select a rectangle that represents the optimum parameter combination and click **Insert** to estimates a model with these parameters. For information about using this plot, see "Selecting Model Orders from the Best ARX Structure" on page 6-11.

This action adds a new model to the Model Board in the System Identification app. The default name of the parametric model contains the model type and the number of poles, zeros, and delays. For example, `arx692` is an ARX model with n_a=6, n_b=9, and a delay of two samples.

5 Click **Close** to close the ARX Model Structure Selection window.

Note You cannot estimate model orders when using multi-output data.

After estimating model orders and delays, use these values as initial guesses for estimating other model structures, as described in "Estimate Polynomial Models in the App" on page 6-14.

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 2-37 and "Select I/O Channels and Data in idfrd Objects" on page 2-62.

For an example of estimating model orders for a multiple-input system, see "Estimating Delays in the Multiple-Input System" in *System Identification Toolbox Getting Started Guide*.

struc

The `struc` command creates a matrix of possible model-order combinations for a specified range of n_a, n_b, and n_k values.

For example, the following command defines the range of model orders and delays `na=2:5`, `nb=1:5`, and `nk=1:5`:

```
NN = struc(2:5,1:5,1:5))
```

arxstruc

The `arxstruc` command takes the output from `struc`, estimates an ARX model for each model order, and compares the model output to the measured output. `arxstruc` returns the *loss* for each model, which is the normalized sum of squared prediction errors.

For example, the following command uses the range of specified orders `NN` to compute the loss function for single-input/single-output estimation data `data_e` and validation data `data_v`:

```
V = arxstruc(data_e,data_v,NN);
```

Each row in `NN` corresponds to one set of orders:

```
[na nb nk]
```

selstruc

The `selstruc` command takes the output from `arxstruc` and opens the ARX Model Structure Selection window to guide your choice of the model order with the best performance.

For example, to open the ARX Model Structure Selection window and interactively choose the optimum parameter combination, use the following command:

```
selstruc(V);
```

For more information about working with the ARX Model Structure Selection window, see "Selecting Model Orders from the Best ARX Structure" on page 6-11.

To find the structure that minimizes Akaike's Information Criterion, use the following command:

```
nn = selstruc(V,'AIC');
```

where `nn` contains the corresponding `na`, `nb`, and `nk` orders.

Similarly, to find the structure that minimizes the Rissanen's Minimum Description Length (MDL), use the following command:

```
nn = selstruc(V,'MDL');
```

To select the structure with the smallest loss function, use the following command:

```
nn = selstruc(V,0);
```

After estimating model orders and delays, use these values as initial guesses for estimating other model structures, as described in "Using polyest to Estimate Polynomial Models" on page 6-19.

Estimating Delays at the Command Line

The `delayest` command estimates the time delay in a dynamic system by estimating a low-order, discrete-time ARX model and treating the delay as an unknown parameter.

By default, `delayest` assumes that $n_a=n_b=2$ and that there is a good signal-to-noise ratio, and uses this information to estimate n_k.

To estimate the delay for a data set `data`, type the following at the prompt:

```
delayest(data);
```

If your data has a single input, MATLAB computes a scalar value for the input delay—equal to the number of data samples. If your data has multiple inputs, MATLAB returns a vector, where each value is the delay for the corresponding input signal.

To compute the actual delay time, you must multiply the input delay by the sample time of the data.

You can also use the ARX Model Structure Selection window to estimate input delays and model order together, as described in "Estimating Model Orders at the Command Line" on page 6-10.

Selecting Model Orders from the Best ARX Structure

You generate the ARX Model Structure Selection window for your data to select the best-fit model.

For a procedure on generating this plot in the System Identification app, see "Estimating Orders and Delays in the App" on page 6-8. To open this plot at the command line, see "Estimating Model Orders at the Command Line" on page 6-10.

The following figure shows a sample plot in the ARX Model Structure Selection window.

You use this plot to select the best-fit model.
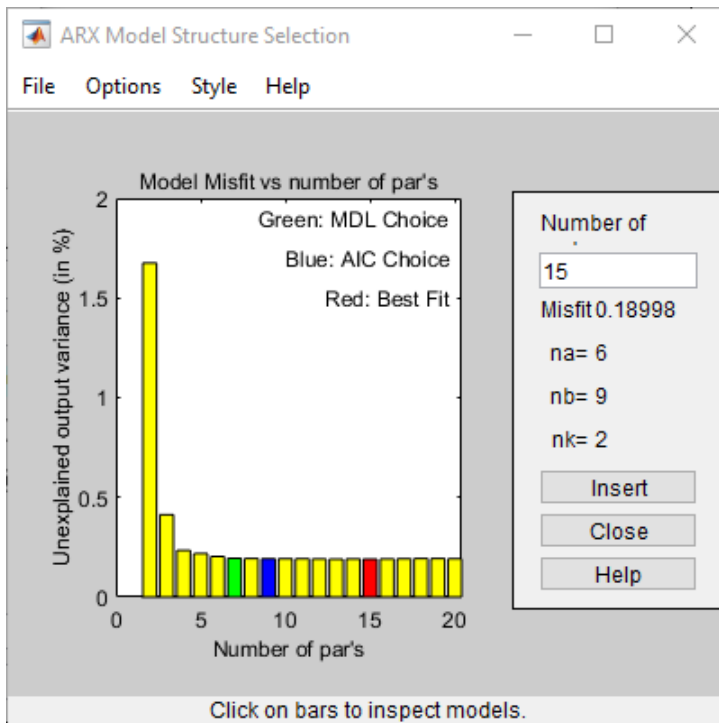
- The horizontal axis is the total number of parameters — $n_a + n_b$.
- The vertical axis, called **Unexplained output variance (in %)**, is the portion of the output not explained by the model—the ARX model prediction error for the number of parameters shown on the horizontal axis.

  The *prediction error* is the sum of the squares of the differences between the validation data output and the model one-step-ahead predicted output.
- n_k is the delay.

Three rectangles are highlighted on the plot in green, blue, and red. Each color indicates a type of best-fit criterion, as follows:

- Red — Best fit minimizes the sum of the squares of the difference between the validation data output and the model output. This rectangle indicates the overall best fit.
- Green — Best fit minimizes Rissanen MDL criterion.
- Blue — Best fit minimizes Akaike AIC criterion.

In the ARX Model Structure Selection window, click any bar to view the orders that give the best fit. The area on the right is dynamically updated to show the orders and delays that give the best fit.

For more information about the AIC criterion, see "Loss Function and Model Quality Metrics" on page 1-46.

See Also

Related Examples

- "Estimate Polynomial Models in the App" on page 6-14
- "Estimate Polynomial Models at the Command Line" on page 6-18
- "Model Structure Selection: Determining Model Order and Input Delay" on page 4-40

More About

- "What Are Polynomial Models?" on page 6-2

Estimate Polynomial Models in the App

Prerequisites

Before you can perform this task, you must have:

- Imported data into the System Identification app. See "Import Time-Domain Data into the App" on page 2-13. For supported data formats, see "Data Supported by Polynomial Models" on page 6-6.

- 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 2-5.

- Select a model structure, model orders, and delays. For a list of available structures, see "What Are Polynomial Models?" on page 6-2 For more information about how to estimate model orders and delays, see "Estimating Orders and Delays in the App" on page 6-8. For multiple-output models, you must specify order matrices in the MATLAB workspace, as described in "Polynomial Sizes and Orders of Multi-Output Polynomial Models" on page 6-21.

1 In the System Identification app, select **Estimate > Polynomial Models** to open the Polynomial Models dialog box.
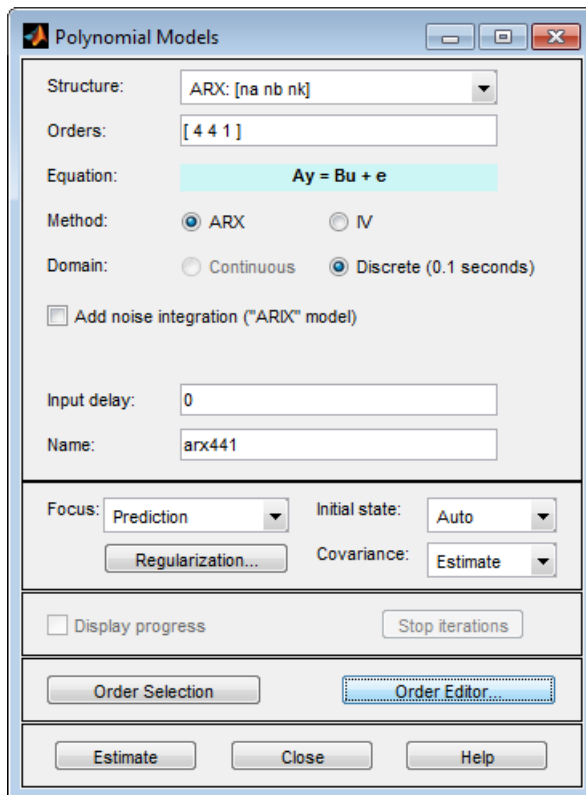
For more information on the options in the dialog box, click **Help**.

2 In the **Structure** list, select the polynomial model structure you want to estimate from the following options:

- ARX:[na nb nk]
- ARMAX:[na nb nc nk]
- OE:[nb nf nk]
- BJ:[nb nc nd nf nk]

This action updates the options in the Polynomial Models dialog box to correspond with this model structure. For information about each model structure, see "What Are Polynomial Models?" on page 6-2.

Note For time-series data, only AR and ARMA models are available. For more information about estimating time-series models, see "Time Series Analysis".

---

3 In the **Orders** field, specify the model orders and delays, as follows:

- For single-output polynomial models, enter the model orders and delays according to the sequence displayed in the **Structure** field. For multiple-input models, specify nb and nk as row vectors with as many elements as there are inputs. If you are estimating BJ and OE models, you must also specify nf as a vector.

  For example, for a three-input system, nb can be [1 2 4], where each element corresponds to an input.

- For multiple-output models, enter the model orders as described in "Polynomial Sizes and Orders of Multi-Output Polynomial Models" on page 6-21.

Tip To enter model orders and delays using the Order Editor dialog box, click **Order Editor**.

---

4 (ARX models only) Select the estimation **Method** as **ARX** or **IV** (instrumental variable method). For information about the algorithms, see "Polynomial Model Estimation Algorithms" on page 6-25.

5 (ARX, ARMAX, and BJ models only) Check the **Add noise integration** check box to add an integrator to the noise source, *e*.

6 Specify the delay using the **Input delay** edit box. The value must be a vector of length equal to the number of input channels in the data. For discrete-time estimations (any estimation using data with nonzero sample-time), the delay must be expressed in the number of lags. These delays are separate from the "in-model" delays specified by the nk order in the **Orders** edit box.

7 In the **Name** field, edit the name of the model or keep the default.

8 In the **Focus** list, select how to weigh the relative importance of the fit at different frequencies. For more information about each option, see "Assigning Estimation Weightings" on page 6-16.

9 In the **Initial state** list, specify how you want the algorithm to treat initial conditions. For more information about the available options, see "Specifying Initial Conditions for Iterative Estimation Algorithms" on page 5-39.

Tip If you get an inaccurate fit, try setting a specific method for handling initial states rather than choosing it automatically.

---

10 In the **Covariance** list, select Estimate if you want the algorithm to compute parameter uncertainties. Effects of such uncertainties are displayed on plots as model confidence regions.

To omit estimating uncertainty, select None. Skipping uncertainty computation for large, multiple-output models might reduce computation time.

11 Click **Regularization** to obtain regularized estimates of model parameters. Specify the regularization constants in the Regularization Options dialog box. To learn more, see "Regularized Estimates of Model Parameters" on page 1-34.

12 (ARMAX, OE, and BJ models only) To view the estimation progress in the MATLAB Command Window, select the **Display progress** check box. This launches a progress viewer window in which estimation progress is reported.

13 Click **Estimate** to add this model to the Model Board in the System Identification app.

14 (Prediction-error method only) To stop the search and save the results after the current iteration has been completed, click **Stop Iterations**. To continue iterations from the current model, click the **Continue iter** button to assign current parameter values as initial guesses for the next search.

Assigning Estimation Weightings

You can specify how the estimation algorithm weighs the fit at various frequencies. In the app, set **Focus** to one of the following options:

- `Prediction` — Uses the inverse of the noise model H to weigh the relative importance of how closely to fit the data in various frequency ranges. Corresponds to minimizing one-step-ahead prediction, which typically favors the fit over a short time interval. Optimized for output prediction applications.

- `Simulation` — Uses the input spectrum to weigh the relative importance of the fit in a specific frequency range. Does not use the noise model to weigh the relative importance of how closely to fit the data in various frequency ranges. Optimized for output simulation applications.

- `Stability` — Estimates the best stable model. For more information about model stability, see "Unstable Models" on page 17-97.

- `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 2-96 or "Defining a Custom Filter" on page 2-96. This prefiltering applies only for estimating the dynamics from input to output. The disturbance model is determined from the unfiltered estimation data.

Next Steps

- Validate the model by selecting the appropriate check box in the **Model Views** area of the System Identification app. For more information about validating models, see "Validating Models After Estimation" on page 17-2.

- Export the model to the MATLAB workspace for further analysis by dragging it to the **To Workspace** rectangle in the System Identification app.

Tip For ARX and OE models, you can use the exported model for initializing a nonlinear estimation at the command line. This initialization may improve the fit of the model. See "Initialize Nonlinear ARX Estimation Using Linear Model" on page 11-20, and "Initialize Hammerstein-Wiener Estimation Using Linear Model" on page 12-7.

See Also

Related Examples

- "Preliminary Step – Estimating Model Orders and Input Delays" on page 6-8
- "Estimate Polynomial Models at the Command Line" on page 6-18

More About

- "What Are Polynomial Models?" on page 6-2
- "Data Supported by Polynomial Models" on page 6-6

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 2-34. For supported data formats, see "Data Supported by Polynomial Models" on page 6-6.

- 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 2-5.

- Select a model structure, model orders, and delays. For a list of available structures, see "What Are Polynomial Models?" on page 6-2 For more information about how to estimate model orders and delays, see "Estimating Model Orders at the Command Line" on page 6-10 and "Estimating Delays at the Command Line" on page 6-11. For multiple-output models, you must specify order matrices in the MATLAB workspace, as described in "Polynomial Sizes and Orders of Multi-Output Polynomial Models" on page 6-21.

Using arx and iv4 to Estimate ARX Models

You can estimate single-output and multiple-output ARX models using the `arx` and `iv4` commands. For information about the algorithms, see "Polynomial Model Estimation Algorithms" on page 6-25.

You can use the following general syntax to both configure and estimate ARX models:

```
% Using ARX method
m = arx(data,[na nb nk],opt);
% Using IV method
m = iv4(data,[na nb nk],opt);
```

`data` is the estimation data and `[na nb nk]` specifies the model orders, as discussed in "What Are Polynomial Models?" on page 6-2.

The third input argument `opt` contains the options for configuring the estimation of the ARX model, such as handling of initial conditions and input offsets. You can create and configure the option set `opt` using the `arxOptions` and `iv4Options` commands. The three input arguments can also be followed by name and value pairs to specify optional model structure attributes such as `InputDelay`, `IODelay`, and `IntegrateNoise`.

To get discrete-time models, use the time-domain data (`iddata` object).

Note Continuous-time polynomials of ARX structure are not supported.

For more information about validating your model, see "Validating Models After Estimation" on page 17-2.

You can use `pem` or `polyest` to refine parameter estimates of an existing polynomial model, as described in "Refine Linear Parametric Models" on page 4-5.

For detailed information about these commands, see the corresponding reference page.

Tip You can use the estimated ARX model for initializing a nonlinear estimation at the command line, which improves the fit of the model. See "Initialize Nonlinear ARX Estimation Using Linear Model" on page 11-20.

Using polyest to Estimate Polynomial Models

You can estimate any polynomial model using the iterative prediction-error estimation method `polyest`. For Gaussian disturbances of unknown variance, this method gives the maximum likelihood estimate. The resulting models are stored as `idpoly` model objects.

Use the following general syntax to both configure and estimate polynomial models:

```
m = polyest(data,[na nb nc nd nf nk],opt,Name,Value);
```

where `data` is the estimation data. `na`, `nb`, `nc`, `nd`, `nf` are integers that specify the model orders, and `nk` specifies the input delays for each input.For more information about model orders, see "What Are Polynomial Models?" on page 6-2.

Tip You do not need to construct the model object using `idpoly` before estimation.

If you want to estimate the coefficients of all five polynomials, *A*, *B*, *C*, *D*, and *F*, you must specify an integer order for each polynomial. However, if you want to specify an ARMAX model for example, which includes only the *A*, *B*, and *C* polynomials, you must set `nd` and `nf` to zero matrices of the appropriate size. For some simpler configurations, there are dedicated estimation commands such as `arx`, `armax`, `bj`, and `oe`, which deliver the required model by using just the required orders. For example, `oe(data,[nb nf nk],opt)` estimates an output-error structure polynomial model.

Note To get faster estimation of ARX models, use `arx` or `iv4` instead of `polyest`.

In addition to the polynomial models listed in "What Are Polynomial Models?" on page 6-2, you can use `polyest` to model the ARARX structure—called the *generalized least-squares model*—by setting `nc=nf=0`. You can also model the ARARMAX structure—called the *extended matrix model*—by setting `nf=0`.

The third input argument, `opt`, contains the options for configuring the estimation of the polynomial model, such as handling of initial conditions, input offsets and search algorithm. You can create and configure the option set `opt` using the `polyestOptions` command. The three input arguments can also be followed by name and value pairs to specify optional model structure attributes such as `InputDelay`, `IODelay`, and `IntegrateNoise`.

For ARMAX, Box-Jenkins, and Output-Error models—which can only be estimated using the iterative prediction-error method—use the `armax`, `bj`, and `oe` estimation commands, respectively. These commands are versions of `polyest` with simplified syntax for these specific model structures, as follows:

```
m = armax(Data,[na nb nc nk]);
m = oe(Data,[nb nf nk]);
m = bj(Data,[nb nc nd nf nk]);
```

Similar to `polyest`, you can specify as input arguments the option set configured using commands `armaxOptions`, `oeOptions`, and `bjOptions` for the estimators `armax`, `oe`, and `bj` respectively. You can also use name and value pairs to configure additional model structure attributes.

Tip If your data is sampled fast, it might help to apply a lowpass filter to the data before estimating the model, or specify a frequency range for the `WeightingFilter` property during estimation. For example, to model only data in the frequency range 0-10 rad/s, use the `WeightingFilter` property, as follows:

```
opt = oeOptions('WeightingFilter',[0 10]);
m = oe(Data, [nb nf nk], opt);
```

For more information about validating your model, see "Validating Models After Estimation" on page 17-2.

You can use `pem` or `polyest` to refine parameter estimates of an existing polynomial model (of any configuration), as described in "Refine Linear Parametric Models" on page 4-5.

For more information, see `polyest`, `pem` and `idpoly`.

See Also

Related Examples
- "Estimate Models Using armax" on page 6-26
- "Preliminary Step – Estimating Model Orders and Input Delays" on page 6-8
- "Estimate Polynomial Models in the App" on page 6-14

More About
- "What Are Polynomial Models?" on page 6-2
- "Data Supported by Polynomial Models" on page 6-6
- "Loss Function and Model Quality Metrics" on page 1-46

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) = length(B{i,j})-nk(i,j).

Polynomial	Dimension	Relation Described	Orders
C,D	N_y-by-1 array of row vectors	`C{i}` and `D{i}` contain coefficients of relations between output y_i and noise e_i	`nc` and `nd` are N_y-by-1 matrices such that each entry contains the degree of the corresponding C and D polynomial, respectively.
F	N_y-by-N_u array of row vectors	`F{i,j}` contains coefficients of relations between output y_i and input u_j	`nf`: N_y-by-N_u matrix such that each entry contains the degree of the corresponding F polynomial.

For more information, see `idpoly`.

For example, consider the ARMAX set of equations for a 2 output, 1 input model:

$$y_1(t) + 0.5\ y_1(t\text{-}1) + 0.9\ y_2(t\text{-}1) + 0.1\ y_2(t\text{-}2) = u(t)\ + 5\ u(t\text{-}1)\ + 2\ u(t\text{-}2) + e_1(t) + 0.01\ e_1(t\text{-}1)$$

$$y_2(t) + 0.05\ y_2(t\text{-}1)\ + 0.3\ y_2(t\text{-}2)\ \ = 10\ u(t\text{-}2) + e_2(t) + 0.1\ e_2(t\text{-}1) + 0.02\ e_2(t\text{-}2)$$

y_1 and y_2 represent the two outputs and u represents the input variable. e_1 and e_2 represent the white noise disturbances on the outputs, y_1 and y_2, respectively. To represent these equations as an ARMAX form polynomial using `idpoly`, configure the A, B, and C polynomials as follows:

```
A = cell(2,2);
A{1,1} = [1 0.5];
A{1,2} = [0 0.9 0.1];
A{2,1} = [0];
A{2,2} = [1 0.05 0.3];

B = cell(2,1);
B{1,1} = [1 5 2];
B{2,1}  = [0 0 10];

C = cell(2,1);
C{1} = [1 0.01];
C{2} = [1 0.1 0.02];

model = idpoly(A,B,C)

model =
Discrete-time ARMAX model:
  Model for output number 1: A(z)y_1(t) = - A_i(z)y_i(t) + B(z)u(t) + C(z)e_1(t)
```

```
   A(z) = 1 + 0.5 z^-1

   A_2(z) = 0.9 z^-1 + 0.1 z^-2

   B(z) = 1 + 5 z^-1 + 2 z^-2

   C(z) = 1 + 0.01 z^-1

 Model for output number 2: A(z)y_2(t) = B(z)u(t) + C(z)e_2(t)
   A(z) = 1 + 0.05 z^-1 + 0.3 z^-2

   B(z) = 10 z^-2

   C(z) = 1 + 0.1 z^-1 + 0.02 z^-2

Sample time: unspecified

Parameterization:
   Polynomial orders:   na=[1 2;0 2]   nb=[3;1]   nc=[1;2]
   nk=[0;2]
   Number of free coefficients: 12
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

`model` is a discrete-time ARMAX model with unspecified sample-time. When estimating such models, you need to specify the orders of these polynomials as input arguments.

In the System Identification app, you can enter the matrices directly in the **Orders** field.

At the command line, define variables that store the model order matrices and specify these variables in the model-estimation command.

Tip To simplify entering large matrices orders in the System Identification app, define the variable NN=[NA NB NK] at the command line. You can specify this variable in the **Orders** field.

See Also
idpoly | ar | arx | bj | oe | armax | polyest

Related Examples
- "Estimate Polynomial Models in the App" on page 6-14
- "Estimate Polynomial Models at the Command Line" on page 6-18

More About
- "What Are Polynomial Models?" on page 6-2

Specifying Initial States for Iterative Estimation Algorithms

When you use the `pem` or `polyest` functions to estimate ARMAX, Box-Jenkins (BJ), Output-Error (OE), you must specify how the algorithm treats initial conditions.

This information supports the estimation procedures "Estimate Polynomial Models in the App" on page 6-14 and "Using polyest to Estimate Polynomial Models" on page 6-19.

In the System Identification app, for ARMAX, OE, and BJ models, set **Initial state** to one of the following options:

- `Auto` — Automatically chooses `Zero`, `Estimate`, or `Backcast` based on the estimation data. If initial states have negligible effect on the prediction errors, the initial states are set to zero to optimize algorithm performance.
- `Zero` — Sets all initial states to zero.
- `Estimate` — Treats the initial states as an unknown vector of parameters and estimates these states from the data.
- `Backcast` — Estimates initial states using a smoothing filter.

At the command line, specify the initial conditions as an estimation option. Use `polyestOptions` to configure options for the `polyest` command, `armaxOptions` for the `armax` command etc. Set the `InitialCondition` option to the desired value in the option set. For example, use this command to estimate an ARMAX model and set the initial states to zero:

```
opt = armaxOptions('InitialCondition','zero');
m = armax(data,[2 2 2 3],opt);
```

For a complete list of values for the `InitialCondition` estimation option, see the `armaxOptions` reference page.

See Also
`armaxOptions` | `arxOptions` | `bjOptions` | `oeOptions` | `polyestOptions` | `iv4Options`

Related Examples
- "Estimate Polynomial Models in the App" on page 6-14
- "Estimate Polynomial Models at the Command Line" on page 6-18

Polynomial Model Estimation Algorithms

For linear ARX and AR models, you can choose between the ARX and IV algorithms. *ARX* implements the least-squares estimation method that uses QR-factorization for overdetermined linear equations. *IV* is the *instrument variable method*. For more information about IV, see the section on variance-optimal instruments in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

The ARX and IV algorithms treat noise differently. ARX assumes white noise. However, the instrumental variable algorithm, IV, is not sensitive to noise color. Thus, use IV when the noise in your system is not completely white and it is incorrect to assume white noise. If the models you obtained using ARX are inaccurate, try using IV.

Note AR models apply to time-series data, which has no input. For more information, see "Time Series Analysis". For more information about working with AR and ARX models, see "Input-Output Polynomial Models".

See Also

ar | arx | iv4

More About

*   "What Are Polynomial Models?" on page 6-2

Estimate Models Using armax

This example shows how to estimate a linear, polynomial model with an ARMAX structure for a three-input and single-output (MISO) system using the iterative estimation method `armax`. For a summary of all available estimation commands in the toolbox, see "Model Estimation Commands" on page 1-31.

Load a sample data set `z8` with three inputs and one output, measured at 1 -second intervals and containing 500 data samples.

```
load iddata8
```

Use `armax` to both construct the `idpoly` model object, and estimate the parameters:

$$A(q)y(t) = \sum_{i=1}^{nu} B_i(q)u_i(t - nk_i) + C(q)e(t)$$

Typically, you try different model orders and compare results, ultimately choosing the simplest model that best describes the system dynamics. The following command specifies the estimation data set, $z8$, and the orders of the A, B, and C polynomials as `na`, `nb`, and `nc`, respectively. `nk` of `[0 0 0]` specifies that there is no input delay for all three input channels.

```
opt = armaxOptions;
opt.Focus = 'simulation';
opt.SearchOptions.MaxIterations = 50;
opt.SearchOptions.Tolerance = 1e-5;
na = 4;
nb = [3 2 3];
nc = 4;
nk = [0 0 0];
m_armax = armax(z8, [na nb nc nk], opt);
```

`Focus`, `Tolerance`, and `MaxIter` are estimation options that configure the estimation objective function and the attributes of the search algorithm. The `Focus` option specifies whether the model is optimized for simulation or prediction applications. The `Tolerance` and `MaxIter` search options specify when to stop estimation. For more information about these properties, see the `armaxOptions` reference page.

`armax` is a version of `polyest` with simplified syntax for the ARMAX model structure. The `armax` method both constructs the `idpoly` model object and estimates its parameters.

View information about the resulting model object.

```
m_armax
```

```
m_armax =
Discrete-time ARMAX model: A(z)y(t) = B(z)u(t) + C(z)e(t)
  A(z) = 1 - 1.284 z^-1 + 0.3048 z^-2 + 0.2648 z^-3 - 0.05708 z^-4

  B1(z) = -0.07547 + 1.087 z^-1 + 0.7166 z^-2

  B2(z) = 1.019 + 0.1142 z^-1

  B3(z) = -0.06739 + 0.06828 z^-1 + 0.5509 z^-2

  C(z) = 1 - 0.06096 z^-1 - 0.1296 z^-2 + 0.02489 z^-3 - 0.04699 z^-4
```

```
Sample time: 1 seconds

Parameterization:
   Polynomial orders:   na=4   nb=[3 2 3]   nc=4   nk=[0 0 0]
   Number of free coefficients: 16
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using ARMAX on time domain data "z8".
Fit to estimation data: 80.86% (simulation focus)
FPE: 2.888, MSE: 0.9868
```

m_armax is an `idpoly` model object. The coefficients represent estimated parameters of this polynomial model. You can use `present(m_armax)` to show additional information about the model, including parameter uncertainties.

View all property values for this model.

get(m_armax)

```
                A: [1 -1.2836 0.3048 0.2648 -0.0571]
                B: {[-0.0755 1.0870 0.7166]  [1.0188 0.1142]  [-0.0674 ... ]}
                C: [1 -0.0610 -0.1296 0.0249 -0.0470]
                D: 1
                F: {[1]  [1]  [1]}
    IntegrateNoise: 0
          Variable: 'z^-1'
           IODelay: [0 0 0]
         Structure: [1x1 pmodel.polynomial]
     NoiseVariance: 2.7984
        InputDelay: [3x1 double]
       OutputDelay: 0
         InputName: {3x1 cell}
         InputUnit: {3x1 cell}
        InputGroup: [1x1 struct]
        OutputName: {'y1'}
        OutputUnit: {''}
       OutputGroup: [1x1 struct]
             Notes: [0x1 string]
          UserData: []
              Name: ''
                Ts: 1
          TimeUnit: 'seconds'
      SamplingGrid: [1x1 struct]
            Report: [1x1 idresults.polyest]
```

The `Report` model property contains detailed information on the estimation results. To view the properties and values inside `Report`, use dot notation. For example:

m_armax.Report

```
ans =
               Status: 'Estimated using ARMAX with simulation focus'
               Method: 'ARMAX'
     InitialCondition: 'zero'
                  Fit: [1x1 struct]
           Parameters: [1x1 struct]
```

```
    OptionsUsed: [1x1 idoptions.polyest]
      RandState: [1x1 struct]
       DataUsed: [1x1 struct]
    Termination: [1x1 struct]
```

This action displays the contents of estimation report such as model quality measures (`Fit`), search termination criterion (`Termination`), and a record of estimation data (`DataUsed`) and options (`OptionsUsed`).

See Also

Related Examples

- "Estimate Polynomial Models at the Command Line" on page 6-18

More About

- "What Are Polynomial Models?" on page 6-2

Identifying State-Space Models

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. If the set of first-order differential equation is linear in the state and input variables, the model is referred to as a *linear* state space model.

Note Generally, the System Identification Toolbox documentation refers to linear state space models simply as state-space models. You can also identify nonlinear state space models using grey-box and neural state-space objects. For more information, see "Available Nonlinear Models" on page 11-10.

The linear state-space model structure is a good choice for quick estimation because it requires you to specify only one parameter, 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 difference equation. State variables *x(t)* can be reconstructed from the measured input-output data, but are not themselves measured during an experiment.

Continuous-Time Representation

It is often easier to define a parameterized state-space model in continuous time because physical laws are most often described in terms of differential equations. In continuous-time, the linear state-space description has the following form:

$$\dot{x}(t) = Fx(t) + Gu(t) + \tilde{K}w(t)$$
$$y(t) = Hx(t) + Du(t) + w(t)$$
$$x(0) = x0$$

The matrices *F*, *G*, *H*, and *D* contain elements with physical significance—for example, material constants. *x0* specifies the initial states.

Note $\tilde{K} = 0$ gives the state-space representation of an Output-Error model. For more information, see "What Are Polynomial Models?" on page 6-2.

You can estimate continuous-time state-space model using both time- and frequency-domain data.

Discrete-Time Representation

The discrete-time linear state-space model structure is often written in the *innovations form* that describes noise:

$$x(kT + T) = Ax(kT) + Bu(kT) + Ke(kT)$$
$$y(kT) = Cx(kT) + Du(kT) + e(kT)$$
$$x(0) = x0$$

where T is the sample time, $u(kT)$ is the input at time instant kT, and $y(kT)$ is the output at time instant kT.

Note $K=0$ gives the state-space representation of an Output-Error model. For more information about Output-Error models, see "What Are Polynomial Models?" on page 6-2.

Discrete-time state-space models provide the same type of linear difference relationship between the inputs and outputs as the linear ARMAX model, but are rearranged such that there is only one delay in the expressions.

You cannot estimate a discrete-time state-space model using continuous-time frequency-domain data.

The innovations form uses a single source of noise, $e(kT)$, rather than independent process and measurement noise. If you have prior knowledge about the process and measurement noise, you can use linear grey-box estimation to identify a state-space model with structured independent noise sources. For more information, see "Identifying State-Space Models with Separate Process and Measurement Noise Descriptions" on page 13-55.

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 $\widetilde{K}$ 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 $kT \leq t < (k+1)T$.

The exact relationship between K and $\widetilde{K}$ is complicated. However, for short sample time T, the following approximation works well:

$$K = \int_0^T e^{F\tau} \widetilde{K} d\tau$$

State-Space Representation of Transfer Functions

For linear models, the general model description is given by:

$$y = Gu + He$$

G is a transfer function that takes the input u to the output y. H is a transfer function that describes the properties of the additive output noise model.

The relationships between the transfer functions and the discrete-time state-space matrices on page 7-2 are given by the following equations:

$$G(q) = C(^qB + D$$

$$H(q) = C(^qK + I_{ny}$$

Here, I_{nx} is the nx-by-nx identity matrix, and nx is the number of states. I_{ny} is the ny-by-ny identity matrix, and ny is the dimension of y and e.

The state-space representation in the continuous-time case is similar.

See Also

Related Examples

- "Estimate State-Space Models in System Identification App" on page 7-16
- "Estimate State-Space Models at the Command Line" on page 7-25

More About

- "Data Supported by State-Space Models" on page 7-5
- "Supported State-Space Parameterizations" on page 7-6

Data Supported by State-Space Models

You can estimate linear state-space models from data with the following characteristics:

- Time- or frequency-domain data

  To estimate state-space models for time-series data, see "Time Series Analysis".
- Real data or complex data
- Single-output and multiple-output

To estimate state-space models, you must first import your data into the MATLAB workspace, as described in "Data Preparation".

Supported State-Space Parameterizations

System Identification Toolbox software supports the following parameterizations that indicate which parameters are estimated and which remain fixed at specific values:

- **Free parameterization** results in the estimation of all elements of the system matrices A, B, C, D, and K. See "Estimate State-Space Models with Free-Parameterization" on page 7-30.

- **Canonical parameterization** represents a state-space system in a reduced-parameter form where many entries of the A, B and C matrices are fixed to zeros and ones. The free parameters appear in only a few of the rows and columns in the system matrices A, B, C and D. The software supports companion, modal decomposition and observability canonical forms. See "Estimate State-Space Models with Canonical Parameterization" on page 7-31.

- **Structured parameterization** lets you specify the fixed values of specific parameters and exclude these parameters from estimation. You choose which entries of the system matrices to estimate and which to treat as fixed. See "Estimate State-Space Models with Structured Parameterization" on page 7-32.

- **Completely arbitrary mapping** of parameters to state-space matrices. See "Estimate Linear Grey-Box Models" on page 13-6.

See Also

- "Estimate State-Space Models with Free-Parameterization" on page 7-30
- "Estimate State-Space Models with Canonical Parameterization" on page 7-31
- "Estimate State-Space Models with Structured Parameterization" on page 7-32

Estimate State-Space Model With Order Selection

To estimate a state-space model, you must provide a value of its order, which represents the number of states. When you do not know the order, you can search and select an order using the following procedures.

Estimate Model With Selected Order in the App

You must have already imported your data into the app, as described in "Represent Data".

To estimate model orders for a specific model structure and configuration:

1   In the System Identification app, select **Estimate > State Space Models** to open the State Space Models dialog box.

2   In the **Model Structure** tab, select the **Pick best value in the range** option and specify a range in the adjacent field. The default range is `1:10`.

   This action opens the Model Order Selection window, which displays the relative measure of how much each state contributes to the input-output behavior of the model (*log of singular values of the covariance matrix*). The following figure shows an example plot. In this figure, states 1 and 2 provide the most significant contribution. The contributions to the right of state 2 drop significantly. The red bar illustrates the cutoff. The order of this bar represents the best-value recommendation, and this value appears in **Order**. You can override the recommendation by clicking on another bar or by overwriting the contents of **Order**. For information about using the Model Order Selection window, see "Using the Model Order Selection Window" on page 7-9
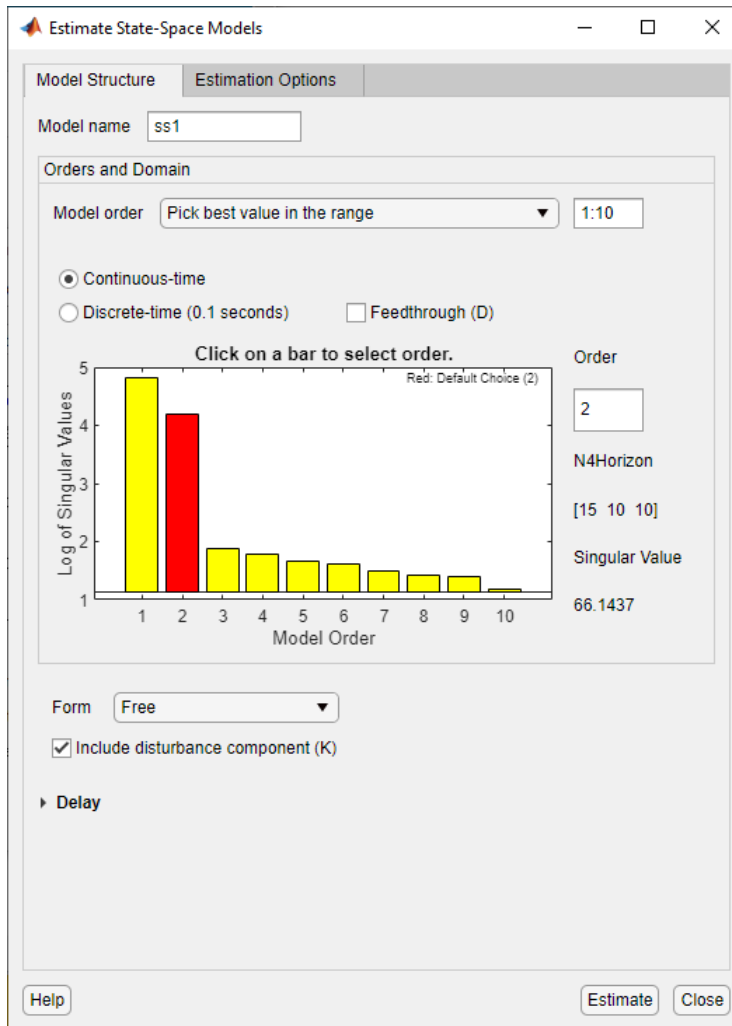
3 (Optional) Specify additional attributes of the model structure, such as input delay and feedthrough. You can also modify the estimation options in the **Estimation Options** tab. As you modify your selections, the software re-evaluates the model-order recommendation.

4 Click **Estimate**. This action adds a new model to the Model Board in the System Identification app. The default name of the model is `ss1`. You can use this model as an initial guess for estimating other state-space models, as described in "Estimate State-Space Models in System Identification App" on page 7-16.

5 Click **Close** to close the window.

Estimate Model With Selected Order at the Command Line

You can estimate a state-space model with selected order using `n4sid`, `ssest` or `ssregest`.

Use the following syntax to specify the range of model orders to try for a specific input delay:

```
m = n4sid(data,n1:n2);
```

where `data` is the estimation data set, `n1` and `n2` specify the range of orders.

The command opens the Model Order Selection window. For information about using this plot, see "Using the Model Order Selection Window" on page 7-9.

Alternatively, use `ssest` or `ssregest`:

```
m1 = ssest(data,nn)
m2 = ssregest(data,nn)
```

where `nn = [n1,n2,...,nN]` specifies the vector or range of orders you want to try.

`n4sid` and `ssregest` estimate a model whose sample time matches that of `data` by default, hence a discrete-time model for time-domain data. `ssest` estimates a continuous-time model by default. You can change the default setting by including the `Ts` name-value pair input arguments in the estimation command. For example, to estimate a discrete-time model of optimal order, assuming `Data.Ts>0`, type:

```
model = ssest(data,nn,'Ts',data.Ts);
```

or

```
model = ssregest(data,nn,'Ts',data.Ts);
```

To automatically select the best order without opening the Model Order Selection window, type `m = n4sid(data,'best')`, `m = ssest(data,'best')` or `m = ssregest(data,'best')`.

Using the Model Order Selection Window

The following figure shows a sample Model Order Selection window.

You use this plot to decide which states provide a significant relative contribution to the input-output behavior, and which states provide the smallest contribution. Based on this plot, select the rectangle that represents the cutoff for the states on the left that provide a significant contribution to the input-output behavior. The recommended choice is shown in red. To learn how to generate this plot, see "Estimate Model With Selected Order in the App" on page 7-7 or "Estimate Model With Selected Order at the Command Line" on page 7-8.

The horizontal axis corresponds to the model order n. The vertical axis, called **Log of Singular values**, shows the singular values of a covariance matrix constructed from the observed data.

For example, in the previous figure, states 1 and 2 provide the most significant contribution. However, the contributions of the states to the right of state 2 drop significantly. This sharp decrease in the log of the singular values after n=2 indicates that using two states is sufficient to get an accurate model.

Use State-Space Estimation to Reduce Model Order

Reduce the order of a Simulink model by linearizing the model and estimating a lower order model that retains model dynamics.

This example requires Simulink and the Simulink Control Design™ toolbox.

Consider the Simulink model `idF14Model`. Linearizing this model gives a ninth-order model. However, the dynamics of the model can be captured, without compromising the fit quality too much, using a lower-order model.

Obtain the linearized model.

```
load_system('idF14Model');
io = getlinio('idF14Model');
sys_lin = linearize('idF14Model',io);
```

`sys_lin` is a ninth-order state-space model with two outputs and one input.

Simulate the step response of the linearized model, and use the data to create an `iddata` object.

```
Ts = 0.0444;
t = (0:Ts:4.44)';
y = step(sys_lin,t);
```

```
data = iddata([zeros(20,2);y],[zeros(20,1); ones(101,1)],Ts);
```

`data` is an `iddata` object that encapsulates the step response of `sys_lin`.

Compare the data to the model linearization.

```
compare(data,sys_lin);
```

Simulated Response Comparison
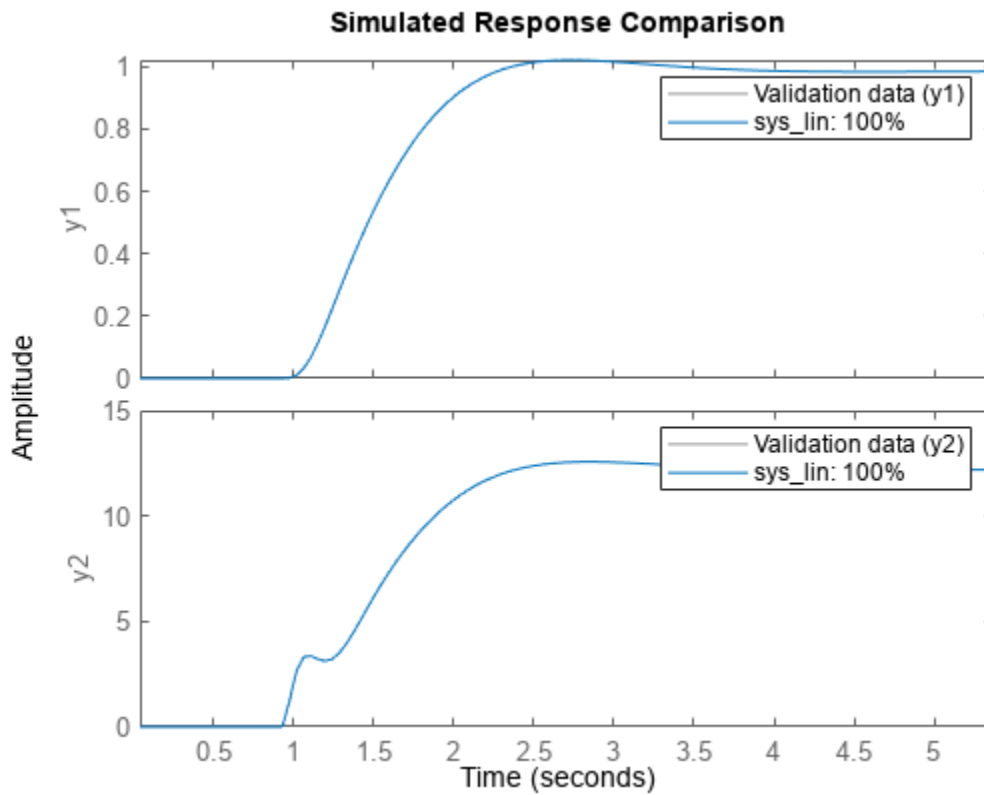
Because the data was obtained by simulating the linearized model, there is a complete match between the data and model linearization response.

Identify a state-space model with a reduced order that adequately fits the data.

Determine an optimal model order.

```
nx = 1:9;
sys1 = ssest(data,nx,'DisturbanceModel','none');
```

A plot showing the Hankel singular values (SVD) for models of the orders specified by nx appears.
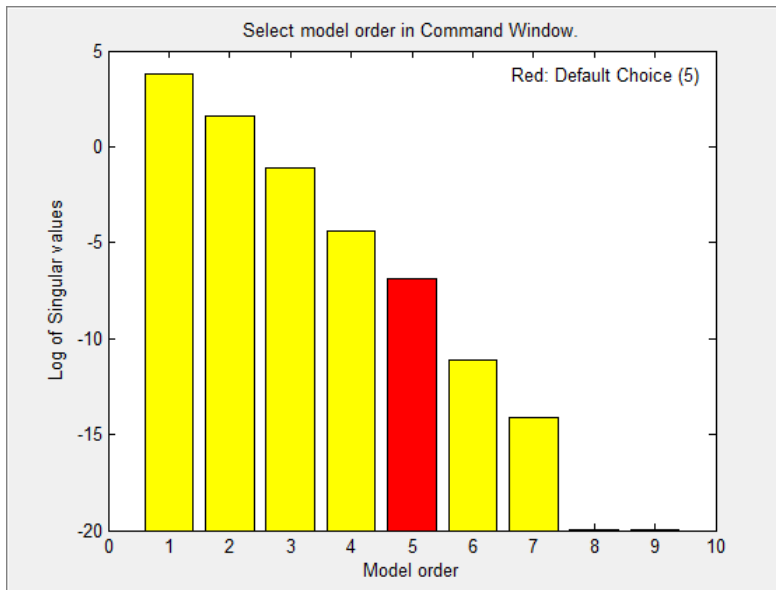
States with relatively small Hankel singular values can be safely discarded. The plot suggests using a fifth-order model.

At the MATLAB command prompt, select the model order for the estimated state-space model. Specify the model order as 5, or press **Enter** to use the default order value.

Compare the data to the estimated model.

```
compare(data,sys1);
```

Simulated Response Comparison
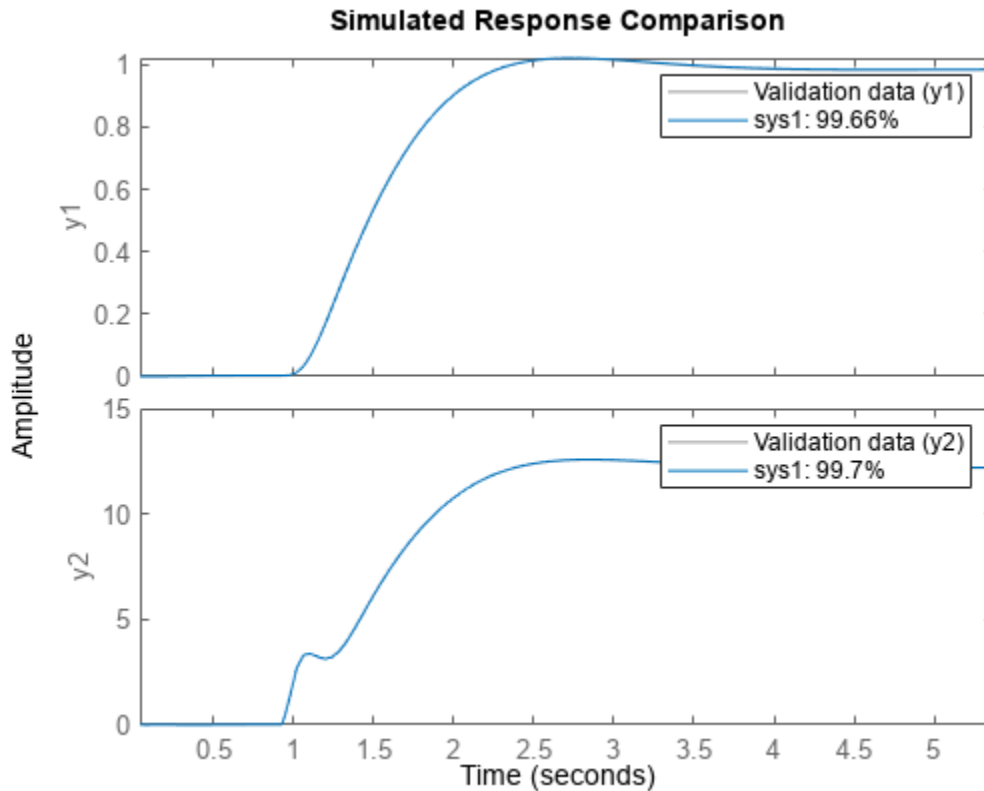
The plot displays the fit percentages for the two `sys1` outputs. The four-state reduction in model order results in a relatively small reduction in fit percentage.

Examine the stopping condition for the search algorithm.

```
sys1.Report.Termination.WhyStop
```

```
ans =
'Maximum number of iterations reached.'
```

Create an estimation options set. Specify the `'lm'` search method. Increase the maximum number of search iterations to 50 from the default maximum of 20.

```
opt = ssestOptions('SearchMethod','lm');
opt.SearchOptions.MaxIterations = 50;
opt.Display = 'on';
```

Identify a state-space model using the estimation option set and `sys1` as the estimation initialization model.

```
sys2 = ssest(data,sys1,opt);
```

Compare the response of the linearized and the estimated models.

```
compare(data,sys_lin,sys2);
```
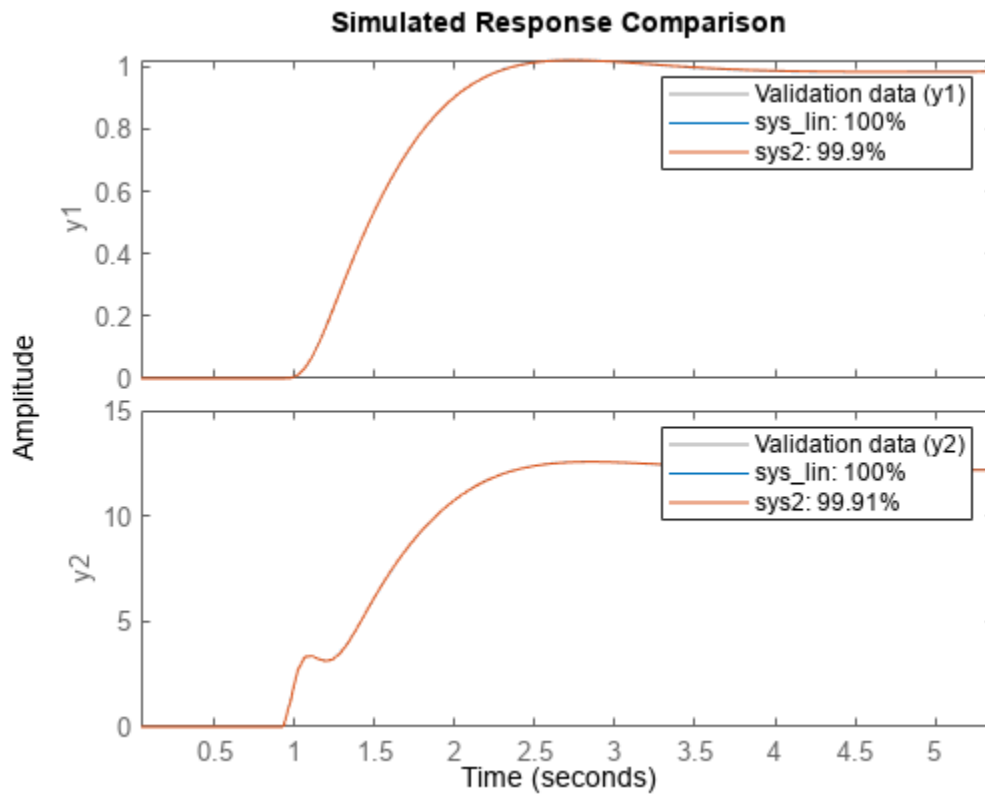
The updated option set results in better fit percentages for `sys2`.

See Also
`linearize` | `getlinio`

Estimate State-Space Models in System Identification App

Prerequisites

- Import data into the System Identification app. See "Represent Data". For supported data formats, see "Data Supported by State-Space Models" on page 7-5.
- Perform data preprocessing. To improve the accuracy of your model, you detrend your data. See "Ways to Prepare Data for System Identification" on page 2-5.

1   Select **Estimate > State Space Models**.
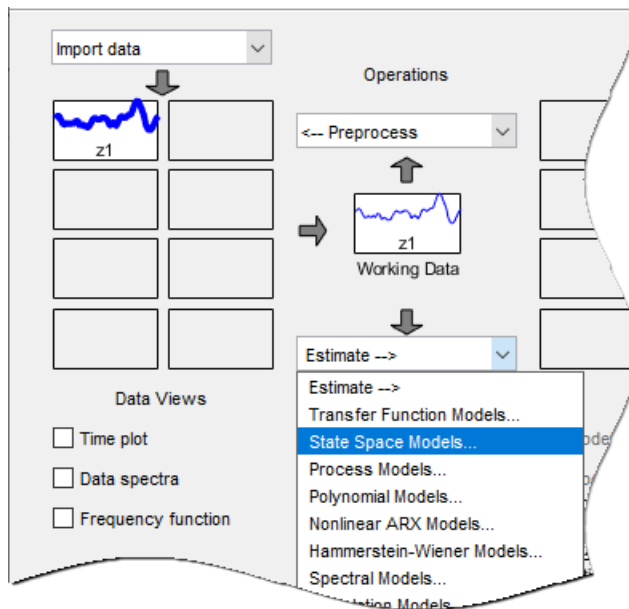
The State Space Models dialog box opens.

2 **Model name** displays the default model name. To change the name, enter a new name. The name of the model must be unique in the Model Board.

3 Select the **Specify value** option (if not already selected) and specify the model order in the edit field. Model order refers to the number of states in the state-space model.

4 Select the **Continuous-time** or **Discrete-time** option to specify the type of model to estimate.

You cannot estimate a discrete-time model if the working data is continuous-time frequency-domain data.

5 Specify the elements to include in the model structure, including feedthrough (*D* matrix) and the disturbance component (*K* matrix.) Specify the model form, such as canonical form, by selecting from the options in **Form**. To specify delays, expand the **Delay** section.

For more information about the type of state-space parameterization, see "Supported State-Space Parameterizations" on page 7-6.

6   Select the **Estimation Options** tab to select the estimation method and configure the cost function.

Select one of the methods in **Estimation method** and configure the options. For more information about these methods, see "State-Space Model Estimation Methods" on page 7-41.

Subspace (N4SID)

a In the **N4Weight** drop-down list, specify the weighting scheme used for singular-value decomposition by the N4SID algorithm.

The N4SID algorithm is used both by the subspace and `Prediction Error Minimization (PEM)` methods.

b In the **N4Horizon** field, specify the forward and backward prediction horizons used by the N4SID algorithm.

c In the **Estimation Focus** drop-down list, select whether to optimize the estimation for one-step-ahead prediction or for simulation. For more information about these options, options see "Assigning Estimation Weightings" on page 7-24.

d Select the **Allow unstable models** check box to specify whether to allow the estimation process to use parameter values that may lead to unstable models.

This option is available only when **Estimation Focus** is `Prediction`. An unstable model is delivered only if it produces a better fit to the data than other stable models computed during the estimation process.

e Select the **Estimate covariance** check box if you want the algorithm to compute parameter uncertainties.

Effects of such uncertainties are displayed on plots as model confidence regions. Skipping uncertainty computation reduces computation time for complex models and large data sets.

f Select the **Display progress** check box to open a progress viewer window during estimation.

g In the **Initial conditions** list, specify how you want the algorithm to treat initial states. For more information about the available options, see "Specifying Initial States for Iterative Estimation Algorithms" on page 7-40.

h If you want to limit the estimation process to a specific frequency range, expand **Fit Frequency Range**. Create a passband filter by specifying the minimum and maximum passband frequencies.

Prediction Error Minimization (PEM)

a   In the **N4Weight** drop-down list, specify the weighting scheme used for singular-value decomposition by the N4SID algorithm.

   The N4SID algorithm is used both by the subspace and `Prediction Error Minimization (PEM)` methods.

b   In the **N4Horizon** field, specify the forward and backward prediction horizons used by the N4SID algorithm.

c   In the **Estimation Focus** drop-down list, select whether to optimize the estimation for one-step-ahead prediction or for simulation. For more information about these options, options see "Assigning Estimation Weightings" on page 7-24.

d   Select the **Allow unstable models** check box to specify whether to allow the estimation process to use parameter values that may lead to unstable models.

   This option is available only when **Estimation Focus** is `Prediction`. An unstable model is delivered only if it produces a better fit to the data than other stable models computed during the estimation process.

e   Select the **Estimate covariance** check box if you want the algorithm to compute parameter uncertainties.

   Effects of such uncertainties are displayed on plots as model confidence regions. Skipping uncertainty computation reduces computation time for complex models and large data sets.

f   Select the **Display progress** check box to open a progress viewer window during estimation.

g In **Initial conditions**, specify how you want the algorithm to treat initial states. For more information about the available options, see "Specifying Initial States for Iterative Estimation Algorithms" on page 7-40.

h If you want to limit the estimation process to a specific frequency range, expand **Fit Frequency Range**. Create a passband filter by specifying the minimum and maximum passband frequencies.

i Expand **Search Options** to specify options for controlling the search iterations.

Search Options

In the **Search Options** section of **Estimation Options**, you can specify the following options:

- **Search method** — Method used by the iterative search algorithm. Search method is `auto` by default. Search methods include `Gauss-Newton(gn)`, `Adaptive Gauss-Newton(gna)`, `Levenberg-Marquardt(lm)`, `Trust-Region Reflective Newton (lsqnonlin)`, `Gradient Search (grad)`, `Sequential Quadratic Programming (fmincon:sqp)`, or `Interior Point (fmincon: interior-point)`. The descent direction is calculated successively at each iteration until a sufficient reduction in error is achieved.

  Output weight — 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 iterations** — Maximum number of iterations to use during search.
- **Tolerance** — Tolerance value when the iterations should terminate.
- **Outlier threshold** — Robustification of the quadratic criterion of fit.

j Expand **Regularization** to obtain regularized estimates of model parameters. Specify the regularization constants *Lambda* and *R*.
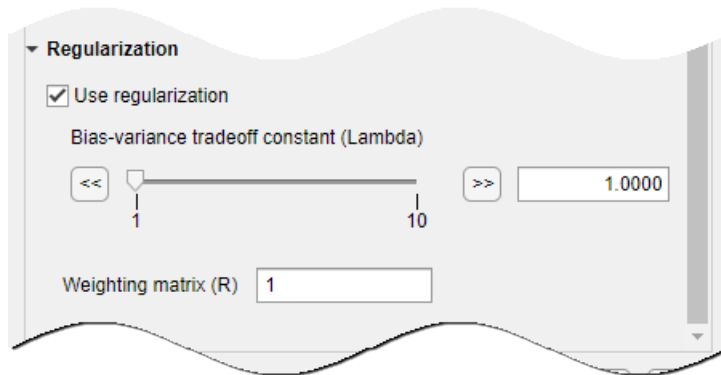
To learn more about regularization, see "Regularized Estimates of Model Parameters" on page 1-34.

Regularized Reduction
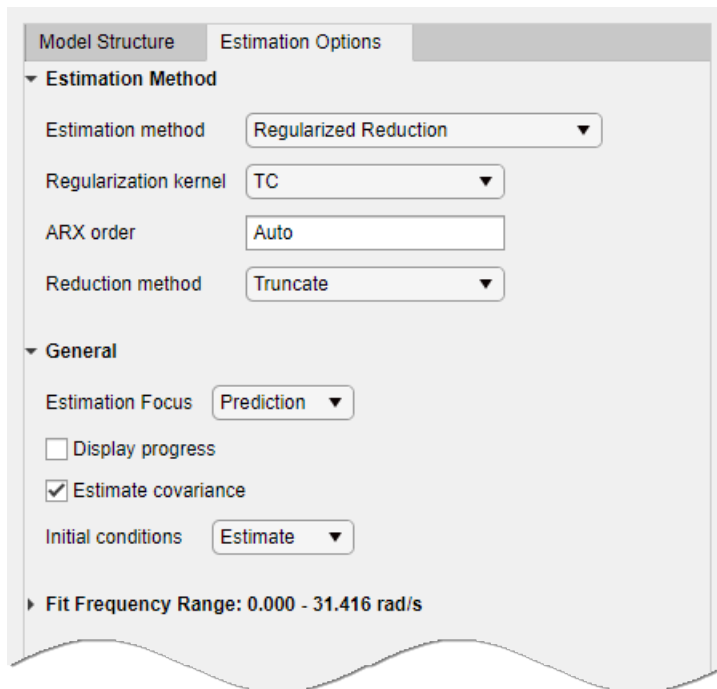
a In the **Regularization Kernel** drop-down list, select the regularizing kernel to use for regularized estimation of the underlying ARX model. To learn more, see "Regularized Estimates of Model Parameters" on page 1-34.

b In the **ARX Orders** field, specify the order of the underlying ARX model. By default, the orders are automatically computed by the estimation algorithm. If you specify a value, it is

recommended that you use a large value for `nb` order. To learn more about ARX orders, see `arx`.

c In the **Estimation Focus** drop-down list, select whether to optimize the estimation for one-step-ahead prediction or for simulation. For more information about these options, options see "Assigning Estimation Weightings" on page 7-24.

d In the **Reduction Method** drop-down list, specify the reduction method:

- `Truncate` — Discards the specified states without altering the remaining states. This method tends to product a better approximation in the frequency domain, but the DC gains are not guaranteed to match.

- `MatchDC` — Discards the specified states and alters the remaining states to preserve the DC gain.

e Select the **Estimate covariance** check box if you want the algorithm to compute parameter uncertainties.

Effects of such uncertainties are displayed on plots as model confidence regions. Skipping uncertainty computation reduces computation time for complex models and large data sets.

f Select the **Display progress** check box to open a progress viewer window during estimation.

g In **Initial conditions**, specify how you want the algorithm to treat initial states. For more information about the available options, see "Specifying Initial States for Iterative Estimation Algorithms" on page 7-40.

h If you want to limit the estimation process to a specific frequency range, expand **Fit Frequency Range**. Create a passband filter by specifying the minimum and maximum passband frequencies.

This estimation process uses parameter values that always lead to a stable model.

7 Click **Estimate** to estimate the model. A new model gets added to the System Identification app.

Next Steps

- Validate the model by selecting the appropriate response type in the **Model Views** area of the app. For more information about validating models, see "Validating Models After Estimation" on page 17-2.

- Export the model to the MATLAB workspace for further analysis by dragging it to the **To Workspace** rectangle in the app.

Assigning Estimation Weightings

You can specify both how the estimation algorithm weights the fit at various frequencies and what frequency range the app uses. In the app, set **Estimation Focus** to one of the following options:

- `Prediction` — Uses the ratio of the input spectrum U to the inverse of the noise model H to weight the relative importance of data across the full frequency range. This weighting 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 only, and not the noise model, for weighting. Optimized for output simulation applications.

You can apply a passband to limit the frequency range over which the estimation algorithm performs the fit.

For more information on estimation weighting, see the section Effects of `Focus` and `WeightingFilter` Options in "Loss Function and Model Quality Metrics" on page 1-46.

Estimate State-Space Models at the Command Line

Black Box vs. Structured State-Space Model Estimation

You can estimate state-space models in two ways at the command line, depending upon your prior knowledge of the nature of the system and your requirements.

Black Box Estimation

In this approach, you specify the model order, and, optionally, additional model structure attributes that configure the overall structure of the state-space matrices. You call `ssest`, `ssregest` or `n4sid` with data and model order as primary input arguments, and use name-value pairs to specify any additional attributes, such as model sample time, presence of feedthrough, absence of noise component, etc. You do not work directly with the coefficients of the *A*, *B*, *C*, *D*, *K*, and *X0* matrices.

Structured Estimation

In this approach, you create and configure an `idss` model that contains the initial values for all the system matrices. You use the `Structure` property of the `idss` model to specify all the parameter constraints. For example, you can designate certain coefficients of system matrices as fixed and impose minimum/maximum bounds on the values of the others. For quick configuration of the parameterization and whether to estimate feedthrough and disturbance dynamics, use `ssform`.

After configuring the `idss` model with desired constraints, you specify this model as an input argument to the `ssest` command. You cannot use `n4sid` or `ssregest` for structured estimation.

Note

- The structured estimation approach is also referred to as grey-box modeling. However, in this toolbox, the "grey box modeling" terminology is used only when referring to `idgrey` and `idnlgrey` models.

- Using the structured estimation approach, you cannot specify relationships among state-space coefficients. Each coefficient is essentially considered to be independent of others. For imposing dependencies, or to use more complex forms of parameterization, use the `idgrey` model and `greyest` estimator.

Estimating State-Space Models Using ssest, ssregest and n4sid

Prerequisites

- Represent input-output data as an `iddata` object or frequency-response data as an `frd` or `idfrd` object. See "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-34. For supported data formats, see "Data Supported by State-Space Models" on page 7-5.

- Perform data preprocessing. To improve the accuracy of results when using time-domain data, you can detrend the data or specify the input/output offset levels as estimation options. See "Ways to Prepare Data for System Identification" on page 2-5.

- Select a model order. When you do not know the model order, search and select for an order. For more information, see "Estimate Model With Selected Order at the Command Line" on page 7-8.

You can estimate continuous-time and discrete-time state-space models using the iterative estimation command `ssest` that minimizes the prediction errors to obtain maximum-likelihood values.

Use the following general syntax to both configure and estimate state-space models:

```
m = ssest(data,n,opt,Name,Value)
```

where `data` is the estimation data, `n` is the model order, and `opt` contains options for configuring the estimation of the state-space models. These options include the handling of the initial conditions, input and output offsets, estimation focus and search algorithm options. opt can be followed by name-value pair input arguments that specify optional model structure attributes such as the presence of feedthrough, the canonical form of the model, and input delay.

As an alternative to `ssest`, you can use the noniterative subspace estimators `n4sid` or `ssregest`:

```
m = n4sid(data,n,opt,Name,Value)
m = ssregest(data,n,opt,Name,Value)
```

Unless you specify the sample time as a name-value pair input argument, `n4sid` and `ssregest` estimate a discrete-time model, while `ssest` estimates a continuous-time model.

Note `ssest` uses `n4sid` to initialize the state-space matrices, and takes longer than `n4sid` to estimate a model but typically provides a better fit to data.

For information about validating your model, see "Validating Models After Estimation" on page 17-2

Choosing the Structure of A, B, C Matrices

By default, all entries of the *A*, *B*, and *C* state-space matrices are treated as free parameters. Using the `Form` name-value pair input argument of `ssest`, you can choose various canonical forms, such as the companion and modal forms, that use fewer parameters.

For more information about estimating a specific state-space parameterization, see:

- "Estimate State-Space Models with Free-Parameterization" on page 7-30
- "Estimate State-Space Models with Canonical Parameterization" on page 7-31
- "Estimate State-Space Models with Structured Parameterization" on page 7-32

Choosing Between Continuous-Time and Discrete-Time Representations

For estimation of state-space models, you have the option of switching the model sample time between zero and that of the estimation data. You can do this using the `Ts` name-value pair input argument.

- By default, `ssest` estimates a continuous-time model. If you are using data set with nonzero sample time, `data`, which includes all time domain data, you can also estimate a discrete-time model by using:

  ```
  model = ssest(data,nx,'Ts',data.Ts);
  ```

If you are using continuous-time frequency-domain data, you cannot estimate a discrete-time model.

- By default, `n4sid` and `ssregest` estimate a model whose sample time matches that of the data. Thus, for time-domain data, `n4sid` and `ssregest` deliver a discrete-time model. You can estimate a continuous-time model by using:

```
model = n4sid(data,nx,'Ts',0);
```

or

```
model = ssregest(data,nx,'Ts',0);
```

Choosing to Estimate D, K, and X0 Matrices

For state-space models with any parameterization, you can specify whether to estimate the *D*, *K* and *X0* matrices, which represent the input-to-output feedthrough, noise model and the initial states, respectively.

For state-space models with structured parameterization, you can also specify to estimate the *D* matrix. However, for free and canonical forms, the structure of the *D* matrix is set based on your choice for the `'Feedthrough'` name-value pair input argument.

D Matrix

By default, the *D* matrix is not estimated and its value is fixed to zero, except for static models.

- **Black box estimation:** Use the `Feedthrough` name-value pair input argument to denote the presence or absence of feedthrough from individual inputs. For example, in case of a two input model such that there is feedthrough from only the second input, use:

```
model = n4sid(data,n,'Feedthrough',[false true]);
```

- **Structured estimation:** Configure the values of the `init_sys.Structure.D`, where `init_sys` is an `idss` model that represents the desired model structure. To force no feedthrough for the *i*-th input, set:

```
init_sys.Structure.D.Value(:,i) = 0;
init_sys.Structure.D.Free = true;
init_sys.Structure.D.Free(:,i) = false;
```

The first line specifies the value of the *i*-th column of D as zero. The next line specifies all the elements of *D* as free, estimable parameters. The last line specifies that the *i*-th column of the *D* matrix is fixed for estimation.

Alternatively, use `ssform` with `'Feedthrough'` name-value pair.

K Matrix

K represents the noise matrix of the model, such that the noise component of the model is:.

$$\dot{x} = Ax + Ke$$
$$y_n = Cx + e$$

For frequency-domain data, no noise model is estimated and *K* is set to 0. For time-domain data, *K* is estimated by default in the black box estimation setup. y^n is the contribution of the disturbances to the model output.

- **Black box estimation:** Use the `DisturbanceModel` name-value pair input argument to indicate if the disturbance component is fixed to zero (specify `Value = 'none'`) or estimated as a free parameter (specify `Value = 'estimate'`). For example, use :

  ```
  model = n4sid(data,n,'DisturbanceModel','none');
  ```

- **Structured estimation:** Configure the value of the `init_sys.Structure.K` parameter, where `init_sys` is an `idss` model that represents the desired model structure. You can fix some *K* matrix coefficients to known values and prescribe minimum/maximum bounds for free coefficients. For example, to estimate only the first column of the *K* matrix for a two output model:

  ```
  kpar = init_sys.Structure.K;
  kpar.Free(:,1) = true;
  kpar.Free(:,2) = false;
  kpar.Value(:,2) = 0; % second column value is fixed to zero
  init_sys.Structure.K = kpar;
  ```

  Alternatively, use `ssform`.

When not sure how to easily fix or free all coefficients of *K*, initially you can omit estimating the noise parameters in *K* to focus on achieving a reasonable model for the system dynamics. After estimating the dynamic model, you can use `ssest` to refine the model while configuring the *K* parameters to be free. For example:

```
init_sys = ssest(data, n,'DisturbanceModel','none');
init_sys.Structure.K.Free = true;
sys = ssest(data,init_sys);
```

where `init_sys` is the dynamic model without noise.

To set *K* to zero in an existing model, you can set its `Value` to `0` and `Free` flag to `false`:

```
m.Structure.K.Value = 0;
m.Structure.K.Free = false;
```

X0 Matrices

The initial state vector *X0* is obtained as the by-product of model estimation. The `n4sid`, `ssest` and `ssregest` commands return the value of *X0* as their second output arguments. You can choose how to handle initial conditions during model estimation by using the `InitialState` estimation option. Use `n4sidOptions` (for `n4sid`), `ssestOptions` (for `ssest`) or `ssregestOptions` (for `ssregest`) to create the estimation option set. For example, in order to hold the initial states to zero during estimation using `n4sid`:

```
opt = n4sidOptions;
opt.InitialState = 'zero';
[m,X0] = n4sid(data,n,opt);
```

The returned `X0` variable is a zero vector of length `n`.

When you estimate models using multiexperiment data, the `X0` matrix contains as many columns as data experiments.

For a complete list of values for the `InitialStates` option, see "Specifying Initial States for Iterative Estimation Algorithms" on page 7-40.

See Also

More About

- "Loss Function and Model Quality Metrics" on page 1-46

Estimate State-Space Models with Free-Parameterization

The default parameterization of the state-space matrices *A*, *B*, *C*, *D*, and *K* is free; that is, any elements in the matrices are adjustable by the estimation routines. Because the parameterization of *A*, *B*, and *C* is free, a basis for the state-space realization is automatically selected to give well-conditioned calculations.

To estimate the disturbance model *K*, you must use time-domain data.

Suppose that you have no knowledge about the internal structure of the discrete-time state-space model. To quickly get started, use the following syntax:

```
m = ssest(data)
```

or

```
m = ssregest(data)
```

where `data` is your estimation data. `ssest` estimates a continuous-time state-space model for an automatically selected order between 1 and 10. `ssregest` estimates a discrete-time model.

To find a model of a specific order n, use the following syntax:

```
m = ssest(data,n)
```

or

```
m = ssregest(dat,n)
```

The iterative algorithm `ssest` is initialized by the subspace method `n4sid`. You can use `n4sid` directly, as an alternative to `ssest`:

```
m = n4sid(data)
```

which automatically estimates a discrete-time model of the best order in the 1:10 range.

Estimate State-Space Models with Canonical Parameterization

What Is Canonical Parameterization?

Canonical parameterization represents a state-space system in a reduced parameter form where many elements of *A*, *B* and *C* matrices are fixed to zeros and ones. The free parameters appear in only a few of the rows and columns in state-space matrices *A*, *B*, *C*, *D*, and *K*. The free parameters are identifiable — they can be estimated to unique values. The remaining matrix elements are fixed to zeros and ones.

The software supports the following canonical forms:

- **Companion form**: The characteristic polynomial appears in the rightmost column of the *A* matrix.
- **Modal decomposition form**: The state matrix *A* is block diagonal, with each block corresponding to a cluster of nearby modes.

> **Note** The modal form has a certain symmetry in its block diagonal elements. If you update the parameters of a model of this form (as a structured estimation using `ssest`), the symmetry is not preserved, even though the updated model is still block-diagonal.

- **Observability canonical form**: The free parameters appear only in select rows of the *A* matrix and in the *B* and *K* matrices.

  For more information about the distribution of free parameters in the observability canonical form, see the Appendix 4A, pp 132-134, on identifiability of black-box multivariable model structures in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999 (equation 4A.16).

For more information on canonical forms, see "State-Space Realizations" on page 7-42.

Estimating Canonical State-Space Models

You can estimate state-space models with chosen parameterization at the command line.

For example, to specify an observability canonical form, use the `'Form'` name-value pair input argument, as follows:

```
m = ssest(data,n,'Form','canonical')
```

Similarly, set `'Form'` as `'modal'` or `'companion'` to specify modal decomposition and companion canonical forms, respectively.

If you have time-domain data, the preceding command estimates a continuous-time model. If you want a discrete-time model, specify the data sample time using the `'Ts'` name-value pair input argument:

```
md = ssest(data, n,'Form','canonical','Ts',data.Ts)
```

If you have continuous-time frequency-domain data, you can only estimate a continuous-time model.

Estimate State-Space Models with Structured Parameterization

What Is Structured Parameterization?

Structured parameterization lets you exclude specific parameters from estimation by setting these parameters to specific values. This approach is useful when you can derive state-space matrices from physical principles and provide initial parameter values based on physical insight. You can use this approach to discover what happens if you fix specific parameter values or if you free certain parameters.

There are two stages to the structured estimation procedure:

1 Specify the state-space model structure, as described in "Specify the State-Space Model Structure" on page 7-32

2 Estimate the free model parameters, as described in "Estimate State-Space Models at the Command Line" on page 7-25

This approach differs from estimating models with free and canonical parameterizations, where it is not necessary to specify initial parameter values before the estimation. For free parameterization, there is no structure to specify because it is assumed to be unknown. For canonical parameterization, the structure is fixed to a specific form.

Note To estimate structured state-space models in the System Identification app, define the corresponding model structures at the command line and import them into the System Identification app.

Specify the State-Space Model Structure

To specify the state-space model structure:

1 Use `idss` to create a state-space model. For example:

```
A = [0 1; 0 -1];
B = [0; 0.28];
C = eye(2);
D = zeros(2,1);
m = idss(A,B,C,D,K,'Ts',T)
```

creates a discrete-time state-space structure, where A, B, C, D, and K specify the initial values for the free parameters. T is the sample time.

2 Use the `Structure` property of the model to specify which parameters to estimate and which to set to specific values.

More about Structure

`Structure` contains parameters for the five state-space matrices, *A*, *B*, *C*, *D*, and *K*.

For each parameter, you can set the following attributes:

- `Value` — Parameter values. For example, `sys.Structure.A.Value` contains the initial or estimated values of the *A* matrix.

NaN represents unknown parameter values.

Each property `sys.A`, `sys.B`, `sys.C`, and `sys.D` is an alias to the corresponding `Value` entry in the `Structure` property of `sys`. For example, `sys.A` is an alias to the value of the property `sys.Structure.A.Value`

- `Minimum` — Minimum value that the parameter can assume during estimation. For example, `sys.Structure.K.Minimum = 0` constrains all entries in the *K* matrix to be greater than or equal to zero.

- `Maximum` — Maximum value that the parameter can assume during estimation.

- `Free` — Boolean specifying whether the parameter is a free estimation variable. If you want to fix the value of a parameter during estimation, set the corresponding `Free = false`. For example, if *A* is a 3-by-3 matrix, `sys.Structure.A.Free = eyes(3)` fixes all of the off-diagonal entries in *A*, to the values specified in `sys.Structure.A.Value`. In this case, only the diagonal entries in *A* are estimable.

- `Scale` — Scale of the parameter's value. `Scale` is not used in estimation.

- `Info` — Structure array for storing parameter units and labels. The structure has `Label` and `Unit` fields.

  Specify parameter units and labels as character vectors. For example, `'Time'`.

For example, if you want to fix `A(1,2)=A(2,1)=0`, use:

```
m.Structure.A.Value(1,2) = 0;
m.Structure.A.Value(2,1) = 0;
m.Structure.A.Free(1,2) = false;
m.Structure.A.Free(2,1) = false;
```

The estimation algorithm only estimates the parameters in A for which `m.Structure.A.Free` is `true`.

Use physical insight, whenever possible, to initialize the parameters for the iterative search algorithm. Because it is possible that the numerical minimization gets stuck in a local minimum, try several different initialization values for the parameters. For random initialization, use `init`. When the model structure contains parameters with different orders of magnitude, try to scale the variables so that the parameters are all roughly the same magnitude.

Alternatively, to quickly configure the parameterization and whether to estimate feedthrough and disturbance dynamics, use `ssform`.

3 Use `ssest` to estimate the model, as described in "Estimate State-Space Models at the Command Line" on page 7-25.

The iterative search computes gradients of the prediction errors with respect to the parameters using numerical differentiation. The step size is specified by the `nuderst` command. The default step size is equal to 10^{-4} times the absolute value of a parameter or equal to 10^{-7}, whichever is larger. To specify a different step size, edit the `nuderst` MATLAB file.

Are Grey-Box Models Similar to State-Space Models with Structured Parameterization?

You estimate state-space models with structured parameterization on page 7-32 when you know some parameters of a linear system and need to estimate the others. These models are therefore similar to

grey-box models. However, in this toolbox, the "grey box modeling" terminology is used only when referring to `idgrey` and `idnlgrey` models. In these models, you can specify complete linear or nonlinear models with complicated relationships between the unknown parameters.

If you have independent unknown matrix elements in a linear state-space model structure, then it is easier and quicker to use state-space models with structured parameterizations. For imposing dependencies, or to use more complex forms of parameterization, use the `idgrey` model and the associated `greyest` estimator. For more information, see "Grey-Box Model Estimation".

If you want to incorporate prior knowledge regarding the state and output covariances into the estimation process, use an `idgrey` model to identify your system using a general state-space model structure. For more information, see "Identifying State-Space Models with Separate Process and Measurement Noise Descriptions" on page 13-55.

Estimate Structured Discrete-Time State-Space Models

This example shows how to estimate the unknown parameters of a discrete-time model.

In this example, you estimate $\theta_1, \theta_2, \theta_3, \theta_4, \theta_5$ in the following discrete-time model:

$$x(t+1) = \begin{bmatrix} 1 & \theta_1 \\ 0 & 1 \end{bmatrix} x(t) + \begin{bmatrix} \theta_2 \\ \theta_3 \end{bmatrix} u(t) + \begin{bmatrix} \theta_4 \\ \theta_5 \end{bmatrix} e(t)$$

$$y(t) = [1 \ 0] x(t) + e(t)$$

$$x(0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Suppose that the nominal values of the unknown parameters ($\theta_1, \theta_2, \theta_3, \theta_4, \theta_5$) are -1, 2, 3, 4, and 5, respectively.

The discrete-time state-space model structure is defined by the following equation:

$$x(kT+T) = Ax(kT) + Bu(kT) + Ke(kT)$$
$$y(kT) = Cx(kT) + Du(kT) + e(kT)$$
$$x(0) = x0$$

Construct the parameter matrices and initialize the parameter values using the nominal parameter values.

```
A = [1,-1;0,1];
B = [2;3];
C = [1,0];
D = 0;
K = [4;5];
```

Construct the state-space model object.

```
m = idss(A,B,C,D,K);
```

Specify the parameter values in the structure matrices that you do not want to estimate.

```
S = m.Structure;
S.A.Free(1,1) = false;
```

```
S.A.Free(2,:) = false;
S.C.Free = false;
m.Structure = S;
```

D is initialized, by default, as a fixed value, and K and B are initialized as free values. Suppose you want to fix the initial states to known zero values. To enforce this, configure the InitialState estimation option.

```
opt = ssestOptions;
opt.InitialState = 'zero';
```

Load estimation data.

```
load iddata1 z1;
```

Estimate the model structure.

```
m = ssest(z1,m,opt);
```

where z1 is name of the iddata object. The data can be time-domain or frequency-domain data. The iterative search starts with the nominal values in the A, B, C, D, and K matrices.

Estimate Structured Continuous-Time State-Space Models

This example shows how to estimate the unknown parameters of a continuous-time model.

In this example, you estimate $\theta_1, \theta_2, \theta_3$ in the following continuous-time model:

$$\dot{x}(t) = \begin{bmatrix} 0 & 1 \\ 0 & \theta_1 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ \theta_2 \end{bmatrix} u(t)$$

$$y(t) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x(t) + e(t)$$

$$x(0) = \begin{bmatrix} \theta_3 \\ 0 \end{bmatrix}$$

This equation represents an electrical motor, where $y_1(t) = x_1(t)$ is the angular position of the motor shaft, and $y_2(t) = x_2(t)$ is the angular velocity. The parameter $-\theta_1$ is the inverse time constant of the motor, and θ_2 / θ_1 is the static gain from the input to the angular velocity.

The motor is at rest at t=0 , but its angular position θ_3 is unknown. Suppose that the approximate nominal values of the unknown parameters are $\theta_1 = -1$ and $\theta_2 = 0.25$.

The variance of the errors in the position measurement is 0.01 , and the variance in the angular velocity measurements is 0.1 . For more information about this example, see the section on state-space models in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

The continuous-time state-space model structure is defined by the following equation:

$$\dot{x}(t) = Fx(t) + Gu(t) + \widetilde{K}w(t)$$
$$y(t) = Hx(t) + Du(t) + w(t)$$
$$x(0) = x0$$

Construct the parameter matrices and initialize the parameter values using the nominal parameter values.

```
A = [0 1;0 -1];
B = [0;0.25];
C = eye(2);
D = [0;0];
K = zeros(2,2);
x0 = [0;0];
```

The matrices correspond to continuous-time representation. However, to be consistent with the idss object property name, this example uses A, B, and C instead of F, G, and H.

Construct the continuous-time state-space model object.

```
m = idss(A,B,C,D,K,'Ts',0);
```

Specify the parameter values in the structure matrices that you do not want to estimate.

```
S = m.Structure;
S.A.Free(1,:) = false;
S.A.Free(2,1) = false;
S.B.Free(1) = false;
S.C.Free = false;
S.D.Free = false;
S.K.Free = false;
m.Structure = S;
m.NoiseVariance = [0.01 0; 0 0.1];
```

The initial state is partially unknown. Use the InitialState option of the ssestOptions option set to configure the estimation behavior of X0.

```
opt = ssestOptions;
opt.InitialState = idpar(x0);
opt.InitialState.Free(2) = false;
```

Estimate the model structure.

```
load(fullfile(matlabroot,'toolbox','ident','iddemos','data','dcmotordata'));
z = iddata(y,u,0.1);
m = ssest(z,m,opt);
```

The iterative search for a minimum is initialized by the parameters in the nominal model m . The continuous-time model is sampled using the same sample time as the data during estimation.

Simulate this system using the sample time T=0.1 for input u and the noise realization e.

```
e = randn(300,2);
u1 = idinput(300);
simdat = iddata([],u1,'Ts',0.1);
simopt = simOptions('AddNoise',true,'NoiseData',e);
y1 = sim(m,simdat,simopt);
```

The continuous system is sampled using Ts=0.1 for simulation purposes. The noise sequence is scaled according to the matrix m.NoiseVariance .

If you discover that the motor was not initially at rest, you can estimate $x_2(0)$ by setting the second element of the InitialState parameter to be free.

```
opt.InitialState.Free(2) = true;
m_new = ssest(z,m,opt);
```

Estimate State-Space Equivalent of ARMAX and OE Models

This example shows how to estimate ARMAX and OE-form models using the state-space estimation approach.

You can estimate the equivalent of multiple-output ARMAX and Output-Error (OE) models using state-space model structures:

- For an `armax` model, specify to estimate the K matrix for the state-space model.
- For an `oe` model, set $K = 0$.

Convert the resulting models into `idpoly` models to see them in the commonly defined ARMAX or OE forms.

Load measured data.

```
load iddata1 z1
```

Estimate state-space models.

```
mss_noK = n4sid(z1,2,'DisturbanceModel','none');
mss = n4sid(z1,2);
```

`mss_noK` is a second order state-space model with no disturbance model used during estimation. `mss` is also a second order state-space model, but with an estimated noise component. Both models use the measured data set `z1` for estimation.

Convert the state-space models to polynomial models.

```
mOE = idpoly(mss_noK);
mARMAX = idpoly(mss);
```

Converting to polynomial models results in the parameter covariance information for `mOE` and `mARMAX` to be lost.

You can use one of the following to recompute the covariance:

- Zero-iteration update using the same estimation data.
- `translatecov` as a Gauss approximation formula-based translation of covariance of `mss_noK` and `mss` into covariance of `mOE` and `mARMAX`.

Reestimate `mOE` and `mARMAX` for the parameters of the polynomial model using a zero iteration update.

```
opt = polyestOptions;
opt.SearchOptions.MaxIterations = 0;

mOE = polyest(z1,mOE,opt);
mARMAX = polyest(z1,mARMAX,opt);
```

The options object, `opt`, specifies a zero iteration update for `mOE` and `mARMAX`. Consequently, the model parameters remain unchanged and only their covariance information is updated.

Alternatively, you can use `translatecov` to convert the estimated models into polynomial form.

```
fcn = @(x)idpoly(x);
mOEt = translatecov(fcn,mss_noK);
mARMAXt = translatecov(fcn,mss);
```

Because `polyest` and `translatecov` use different computation algorithms, the covariance data obtained by running a zero-iteration update may not match that obtained using `translatecov`.

You can view the uncertainties of the model parameters using `present(mOE)` and `present(mARMAX)`.

You can use a state-space model with K = 0 (Output-Error (OE) form) for initializing a Hammerstein-Wiener estimation at the command line. This initialization may improve the fit of the model. See "Initialize Hammerstein-Wiener Estimation Using Linear Model" on page 12-7.

For more information about ARMAX and OE models, see "Input-Output Polynomial Models".

Specifying Initial States for Iterative Estimation Algorithms

When you estimate state-space models, you can specify how the algorithm treats initial states. This information supports the estimation procedures "Estimate State-Space Models in System Identification App" on page 7-16 and "Estimate State-Space Models at the Command Line" on page 7-25.

In the System Identification app, set **Initial state** to one of the following options:

- `Auto` — Automatically chooses `Zero`, `Estimate`, or `Backcast` based on the estimation data. If initial states have negligible effect on the prediction errors, the initial states are set to zero to optimize algorithm performance.
- `Zero` — Sets all initial states to zero.
- `Estimate` — Treats the initial states as an unknown vector of parameters and estimates these states from the data.
- `Backcast` — Estimates initial states using a backward filtering method (least-squares fit).

At the command line, specify the method for handling initial states using the `InitialState` estimation option. For example, to estimate a fourth-order state-space model and set the initial states to be estimated from the data:

```
opt = ssestOptions('InitialState','estimate');
m = ssest(data,4,opt)
```

For a complete list of values for the `InitialState` model property, see the `ssestOptions`, `n4sidOptions` and `ssregestOptions` reference pages.

Note For the `n4sid` algorithm, `'auto'` and `'backcast'` are equivalent to `'estimate'`.

State-Space Model Estimation Methods

You can estimate state-space models using one of the following estimation methods:

- *N4SID* — Noniterative, subspace method. The method works on both time-domain and frequency-domain data and is typically faster than the *SSEST* algorithm. You can choose the subspace algorithms such as `CVA`, `SSARX`, or `MOESP` using the `n4Weight` option. You can also use this method to get an initial model (see `n4sid`), and then refine the initial estimate using the iterative prediction-error method `ssest`.

  For more information about this algorithm, see [1].

- *SSEST* — Iterative method that uses *prediction error minimization* algorithm. The method works on both time-domain and frequency-domain data. For black-box estimation, the method initializes the model parameters using `n4sid` and then updates the parameters using an iterative search to minimize the prediction errors. You can also use this method for structured estimation using an initial model with initial values of one or more parameters fixed in value.

  For more information on this algorithm, see [2].

- *SSREGEST* — Noniterative method. The method works on discrete time-domain data and frequency-domain data. It first estimates a high-order regularized ARX or FIR model, converts it to a state-space model and then performs balanced reduction on it. This method provides improved accuracy on short, noisy data sets.

With all the estimation methods, you have the option of specifying how to handle initial state, delays, feedthrough behavior and disturbance component of the model.

References

[1] van Overschee, P., and B. De Moor. *Subspace Identification of Linear Systems: Theory, Implementation, Applications.* Springer Publishing: 1996.

[2] Ljung, L. *System Identification: Theory For the User*, Second Edition, Upper Saddle River, N.J: Prentice Hall, 1999.

[3] T. Chen, H. Ohlsson, and L. Ljung. "On the Estimation of Transfer Functions, Regularizations and Gaussian Processes - Revisited", *Automatica*, Volume 48, August 2012.

State-Space Realizations

A state-space realization is an implementation of a given input-output behavior. If a system is modeled by a transfer matrix $H(s)$, then a realization is a set of matrices A, B, C, D such that $H(s) = C(sI - A)^{-1}B + D$. In other words, if the system has state vector x, the system behavior can be described by the state equations

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du.$$

There are an infinite number of possible realizations of any system. A minimal realization is any realization in which A has the smallest possible dimension. That is, a given realization A, B, C, D is minimal if there is no other realization A', B', C', D' where A' has smaller dimensions than A.

A state transformation is a rotation of the state vector by an invertible matrix T such that $\widehat{x} = Tx$. State transformation yields an equivalent state-space representation of the system, with

$$\widehat{A} = TAT^{-1}$$
$$\widehat{B} = TB$$
$$\widehat{C} = CT^{-1}$$
$$\widehat{D} = D.$$

Certain minimal realizations known as canonical forms can be useful for some types of dynamic-system theory and analysis. This topic summarizes some of these canonical forms and related transformations.

Modal Form

Modal form is a diagonalized form that separates the system eigenvalues. In modal form, A is a block-diagonal matrix. The block size is typically 1-by-1 for real eigenvalues and 2-by-2 for complex eigenvalues. However, if there are repeated eigenvalues or clusters of nearby eigenvalues, the block size can be larger.

For instance, for a system with eigenvalues $(\lambda_1, \sigma \pm j\omega, \lambda_2)$, the modal A matrix is of the form

$$A_m = \begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \sigma & \omega & 0 \\ 0 & -\omega & \sigma & 0 \\ 0 & 0 & 0 & \lambda_2 \end{bmatrix}.$$

Obtaining Modal Form

Modal form is the default form returned by the `canon` command, `Hmod = canon(H)`.

When performing system identification using `ssest`, obtain modal form by setting `Form` to `modal`.

Controllable Companion Form

In companion realizations, the characteristic polynomial of the system appears explicitly in the A matrix. For a SISO system with characteristic polynomial

$$P(s) = s^n + \alpha_{n-1}s^{n-1} + \alpha_{n-2}s^{n-2} + \dots + \alpha_1 s + \alpha_0,$$

the corresponding controllable companion form has

$$A_{ccom} = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 & -\alpha_0 \\ 1 & 0 & 0 & \dots & 0 & -\alpha_1 \\ 0 & 1 & 0 & \dots & 0 & -\alpha_2 \\ 0 & 0 & 1 & \dots & 0 & -\alpha_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & -\alpha_{n-1} \end{bmatrix}, \quad B_{ccom} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

For multi-input systems, A_{ccom} has the same form, and the first column of B_{ccom} is as shown. This form does not impose a particular structure on the rest of B_{ccom} or on C_{ccom} and D_{ccom}.

Obtaining Controllable Companion Form

The command `canon(H,"companion")` computes a controllable companion-form realization of H by using the state transformation `T = ctrb(H.A,H.B)` to put the *A* matrix into companion form.

When performing system identification using commands such as `ssest` or `n4sid`, obtain companion form by setting `Form` to `companion`.

The companion transformation requires that the system be controllable from the first input. The transformation to companion form is based on the controllability matrix, which is almost always numerically singular for mid-range orders. Hence, avoid using it for computation when possible.

Observable Companion Form

A related form is obtained using the observability state transformation `T = obsv(H.A,H.B)` instead of `T = ctrb(H.A,H.B)`. This form is the dual (transpose) of controllable companion form, as follows:

$$A_{ocom} = A_{ccom}^T$$
$$B_{ocom} = C_{ccom}^T$$
$$C_{ocom} = B_{ccom}^T$$
$$D_{ocom} = D_{ccom}^T.$$

In particular,

$$A_{ocom} = \begin{bmatrix} 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 \\ -\alpha_0 & -\alpha_1 & -\alpha_2 & -\alpha_3 & \dots & -\alpha_{n-1} \end{bmatrix}, \quad C_{ocom} = [1 \ 0 \ \dots \ 0].$$

This form is sometimes known as observability canonical form [1], but it is different from observable canonical form on page 7-44.

Obtaining Observable Companion Form

When performing system identification using commands such as `ssest` or `n4sid`, obtain this form by setting `Form` to `canonical`.

When using the `canon` command, you can obtain observable companion form from the controllable companion form by performing the transpositions yourself. For example, for a state-space (`ss`) model H:

```
Hccom = canon(H,"companion");
Hocom = ss;
Hocom.A = Hccom.A';
Hocom.B = Hccom.C';
Hocom.C = Hccom.B';
Hocom.D = Hccom.D';
```

Controllable Canonical Form

For a strictly proper system with the transfer function

$$H(s) = \frac{\beta_{n-1} s^{n-1} + \ldots + \beta_1 s + \beta_0}{s^n + \alpha_{n-1} s^{n-1} + \ldots + \alpha_1 s + \alpha_0} + d_0,$$

the controllable canonical form [2] is given by:

$$A_{cont} = \begin{bmatrix} 0 & 1 & 0 & 0 & \ldots & 0 \\ 0 & 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 1 \\ -\alpha_0 & -\alpha_1 & -\alpha_2 & -\alpha_3 & \ldots & -\alpha_{n-1} \end{bmatrix}, \quad B_{cont} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix},$$

$$C_{cont} = [\beta_0 \; \beta_1 \; \ldots \; \beta_{n-1}], \qquad\qquad D_{cont} = d_0.$$

In this form, the coefficients of the characteristic polynomial appear in the last row of A_{cont}. Controllable canonical form is a minimal realization in which all model states are controllable. Like companion form and observable canonical form, it can be ill-conditioned for computation.

Obtaining Controllable Canonical Form

There is no MATLAB command for directly computing controllable canonical form. However, if you can obtain the system in the transfer-function form $H(s)$, then you can use the coefficients $\alpha_0,\ldots,\alpha_{n-1}$, $\beta_0,\ldots,\beta_{n-1}$, and d_0 to construct the controllable canonical-form matrices in MATLAB. Then, create the system with the `ss` command..

Observable Canonical Form

The observable canonical form of a system is the dual (transpose) of its controllable canonical form. In this form, the characteristic polynomial of the system appears explicitly in the last column of the A matrix. Observable canonical form can be obtained from the controllable canonical form as follows:

$$A_{obs} = A_{cont}^T$$

$$B_{obs} = C_{cont}^T$$

$$C_{obs} = B_{cont}^T$$

$$D_{obs} = D_{cont}^T.$$

Thus, for the system with transfer function

$$H(s) = \frac{\beta_{n-1}s^{n-1} + ... + \beta_1 s + \beta_0}{s^n + \alpha_{n-1}s^{n-1} + ... + \alpha_1 s + \alpha_0} + d_0,$$

the observable canonical form [2] is given by:

$$A_{obs} = \begin{bmatrix} 0 & 0 & 0 & ... & 0 & -\alpha_0 \\ 1 & 0 & 0 & ... & 0 & -\alpha_1 \\ 0 & 1 & 0 & ... & 0 & -\alpha_2 \\ 0 & 0 & 1 & ... & 0 & -\alpha_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & ... & 1 & -\alpha_{n-1} \end{bmatrix}, \quad B_{obs} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{n-1} \end{bmatrix},$$

$$C_{obs} = [0 \ 0 \ ... \ 0 \ 1], \quad\quad\quad D_{obs} = d_0.$$

Like the companion form, in this form, the coefficients of the characteristic polynomial appear in the last column of A_{obs}. Observable canonical form is a minimal realization in which all model states are observable.

Obtaining Observable Canonical Form

As with controllable canonical form, there is no MATLAB command for directly computing observable canonical form. However, if you can obtain the system in the transfer-function form $H(s)$, then you can use the coefficients $\alpha_0,...,\alpha_{n-1}$, $\beta_0,...,\beta_{n-1}$, and d_0 to construct the observable canonical-form matrices in MATLAB. Then, create the system with the `ss` command.

References

[1] Baillieul, John, "Observability Canonical Form and the Theory of Observers," lecture notes, November 15, 2012, accessed June 10, 2022, https://people.bu.edu/johnb/501Lecture19.pdf.

[2] Gillis, James T., "State Space." In *Control System Fundamentals.*, edited by William S. Levine, 2d ed. The Electrical Engineering Handbook Series. Boca Raton: CRC Press, 2011.

See Also
canon | ss | ssest | n4sid

More About
- "Scaling State-Space Models" (Control System Toolbox)

System Identification Using Eigensystem Realization Algorithm (ERA)

This example considers a 3-degree-of-freedom (DOF) system that is excited by a series of hammer strikes. The resulting displacements are recorded by sensors. The system is proportionally damped such that the dampling matrix is a linear combination of the mass and stiffness matrices. This example requires Signal Processing Toolbox™ for the section on modal analysis.
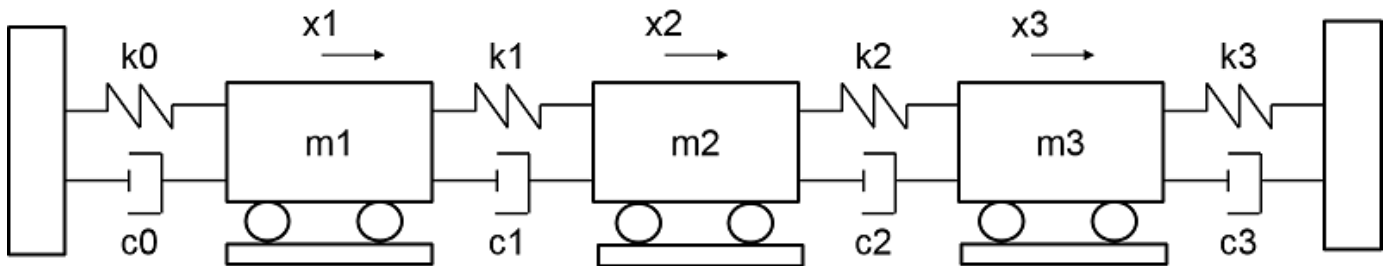
Preprocess Data

Import data for the first set of measurements. The data includes the excitation signals, reponse signals, time signals, and ground truth frequency-response functions. The response signal, denoted by Y1, gives the displacement of the first mass. The excitation signal consists of ten concatenated hammer impacts, and the response signal contains the corresponding displacement. The duration for each impact signal is 2.53 seconds. The excitation and response signals are corrupted with additive noise.

```
load modaldata XhammerMISO1 YhammerMISO1 fs;
rng('default');

% Add noise to excitation and response signals
XhammerMISO1 = XhammerMISO1 + randn(size(YhammerMISO1))/1250;
YhammerMISO1 = YhammerMISO1 + randn(size(YhammerMISO1))/1e11;

% Define outputs from function
t = (0:size(XhammerMISO1,1)-1)/fs';
X1 = 1e2*XhammerMISO1;
Y1 = 1e2*YhammerMISO1;
X0 = X1(:,1);
Y0 = Y1(:,1);
```

Visualize the first excitation and the response channel of the measurement.

```
subplot(2,1,1)
plot(t,X0)
xlabel('Time (s)')
ylabel('Force (N)')
grid on;
title('Excitation and Response for a 3DOF System')
subplot(2,1,2)
plot(t,Y0)
xlabel('Time (s)')
ylabel('Displacement (m)')
grid on;
```

Excitation and Response for a 3DOF System
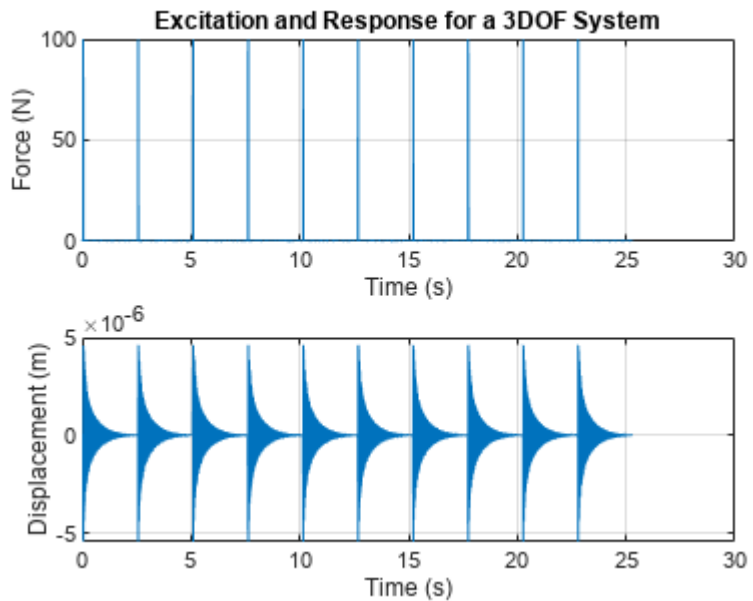
Identify System Using ERA

Create `iddata` objects for estimation and validation data.

```
Ts = 1/fs; % sample time
estimationData = iddata(Y0(1:1000), X0(1:1000), 1/fs);
validationData = iddata(Y0(1001:2000), X0(1001:2000), 1/fs);
```

Visualize the estimation data.

```
figure
plot(estimationData)
```


Input-Output Data
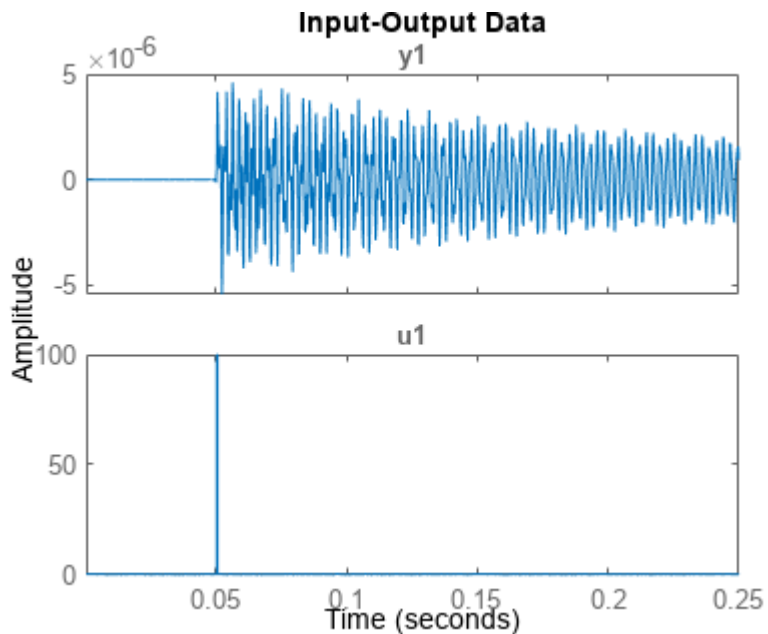
The plot of the input data shows the presence of an input delay. Remove the delay from the data.

```
[~,inputDelay] = max(estimationData.InputData);
estimationData = estimationData(inputDelay:end);
```

The `era` function requires data to be passed as a `timetable` or numeric matrix. Convert the `iddata` object into a `timetable`.

```
L = length(estimationData.InputData);
t = seconds(estimationData.Tstart + (0:L-1)'*Ts);
y1 = estimationData.OutputData;
tt = timetable(t,y1);
```

Use `era` to estimate a state-space model using this estimation data.

```
order = 6;
sys = era(tt, order, 'Feedthrough', true)

sys =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + B u(t) + K e(t)
        y(t) = C x(t) + D u(t) + e(t)

  A =
               x1          x2          x3          x4          x5          x6
    x1      0.8339     -0.5523   -0.001099    0.003053   -0.001497   0.0004633
    x2      0.5523      0.8323    -0.00714   -0.001638   -0.002676   0.0008873
    x3   -0.001099     0.00714      0.2293      0.9709  -0.0009399  -0.0006184
    x4   -0.003053   -0.001638     -0.9709      0.2289    0.006094   -0.002101
    x5   -0.001497    0.002676  -0.0009399   -0.006094     -0.5463     -0.8302
    x6  -0.0004633   0.0008873   0.0006184   -0.002101      0.8302     -0.5465

  B =
               u1
    x1   -0.001403
    x2   0.0007461
    x3   -0.001515
    x4  -0.0001808
    x5   -0.000818
    x6  -0.0002384

  C =
               x1          x2          x3          x4          x5          x6
    y1  -3.508e-07  -1.865e-07  -3.787e-07     4.52e-08  -2.045e-07   5.959e-08

  D =
               u1
    y1   1.852e-10

  K =
           y1
    x1     0
    x2     0
    x3     0
    x4     0
    x5     0
    x6     0
```

```
Sample time: 0.00025 seconds

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: yes
   Disturbance component: none
   Number of free coefficients: 49
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using the Eigensystem Realization Algorithm

Model Properties
```

Compare the response of the estimated system with the validation data.

```
compare(validationData,sys);
```
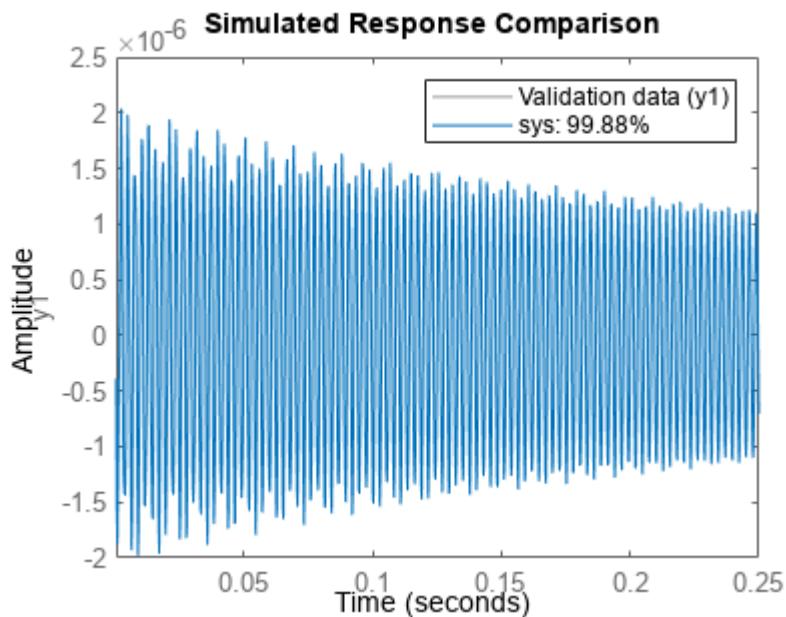

Compare the performance of the estimated model against another state-space model estimated using `ssest`.

```
sys2 = ssest(estimationData, order, 'Feedthrough', true)

sys2 =
  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        x3        x4        x5        x6
     x1     4.99      2323     66.74    -241.3     35.31    -5.076
     x2    -2341    -7.761     300.1    -11.19      11.2     25.28
     x3   -49.75      -311    -5.595     -5360     30.91      15.2
     x4    270.4     6.758      5333    -15.04    -1.474    -64.71
     x5   -52.95    -16.58     14.17     34.15     3.021      8548
```

```
  x6   -41.44   -15.95     16.8    134.7    -8674   -51.74

   B =
             u1
   x1  0.005996
   x2   0.03915
   x3   -0.2075
   x4   -0.1881
   x5     1.007
   x6   -0.9297

   C =
             x1          x2          x3          x4          x5          x6
   y1   2.998e-05  -2.043e-05   1.586e-05   8.501e-06  -2.459e-06  -2.867e-06

   D =
             u1
   y1   5.33e-09

   K =
             y1
   x1    9.881e+06
   x2   -8.603e+06
   x3    4.435e+06
   x4    2.032e+07
   x5   -7.865e+07
   x6    4.392e+07

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: yes
   Disturbance component: estimate
   Number of free coefficients: 55
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on time domain data "estimationData".
Fit to estimation data: 99.6% (prediction focus)
FPE: 5.081e-17, MSE: 4.773e-17

Model Properties
```

```matlab
compare(validationData,sys,sys2);
legend('Validation data','ERA','SSEST');
```
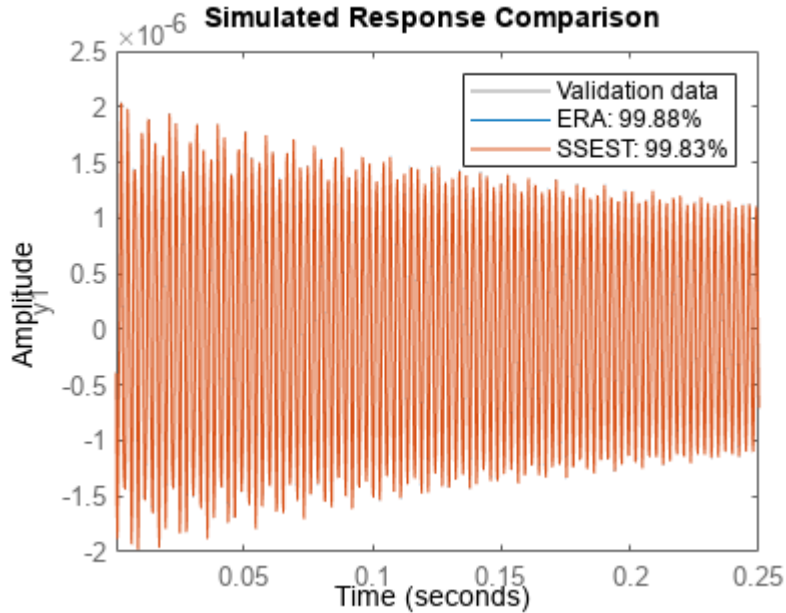
The two models have a similar fit.

Modal Analysis

Perform a modal analysis of the state-space model estimated using `era` by using the `modalfit` function from Signal Processing Toolbox. `modalfit` returns the damping ratios `zeta` and the damped natural frequencies `fd`.

```
[~,f] = modalfrf(sys);
[fd, zeta] = modalfit(sys,f,3);
```

View `zeta`.

```
zeta
```

```
zeta = 3×1

    0.0008
    0.0018
    0.0028
```

View `fd`.

```
fd
```

```
fd = 3×1
10³ ×

    0.3727
    0.8525
    1.3706
```

Compare these results with the modes obtained from the model estimated using `ssest`.

```
[frf,f] = modalfrf(sys2);
[fd, zeta] = modalfit(sys2,f,3)

fd = 3×1
10³ ×

     0.3727
     0.8525
     1.3706


zeta = 3×1

     0.0008
     0.0018
     0.0029
```

There is a close match between the modes obtained from `modalfit` for the second estimated model and those obtained for the model estimated using `era`.

Reduced-order modeling comparison

Compare the performance of `era` and `ssest` when using a lower model order for system identification.

```
order = 4;
sys = era(tt, order, 'Feedthrough', true);
sys2 = ssest(estimationData, order, 'Feedthrough', true);
```

Compare the performance of both models as before.

```
compare(validationData,sys,sys2);
legend('Validation data','ERA','SSEST');
```
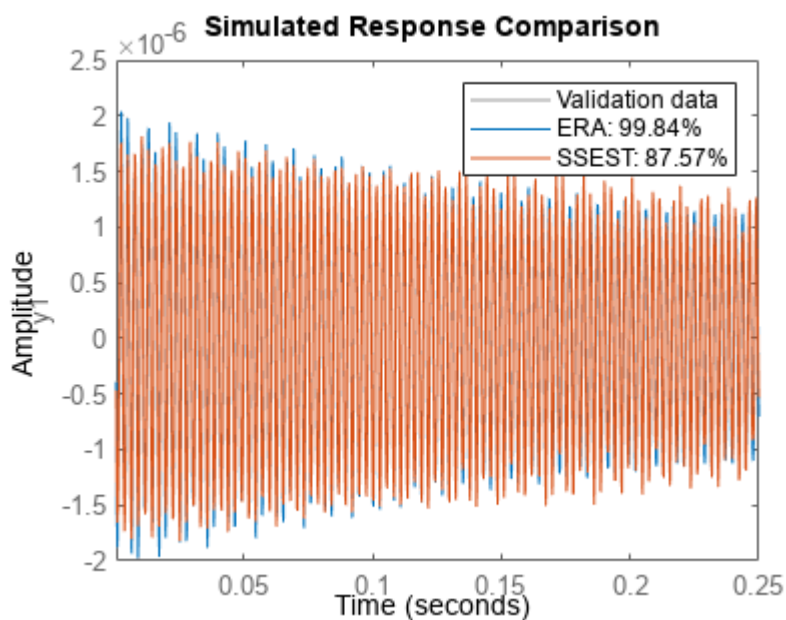
For the reduced-order model, the ERA model has a significantly better fit than the SSEST model. This result demonstrates the capability of the ERA approach in obtaining state-space models from noisy impulse response data.

See Also

era | iddata | timetable | modalfit | modalfrf

8

Identifying Transfer Function Models

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*.

If you take the Z-transform, the transfer function has the form:

$$Y(z^{-1}) = \frac{num(z^{-1})}{den(z^{-1})}U(z^{-1}) + E(z^{-1})$$

$$num(z^{-1}) = b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots$$

$$den(z^{-1}) = 1 + a_1 z^{-1} + a_2 z^{-2} + \ldots$$

Where, $Y(z^{1})$, $U(z^{1})$ and $E(z^{1})$ represent the Z-transforms of the output, input and noise, respectively. z^{1} is the Z-transform of the lag operator.

Delays

In continuous-time, input and transport delays are of the form:

$$Y(s) = \frac{num(s)}{den(s)}e^{-s\tau}U(s) + E(s)$$

Where τ represents the delay.

In discrete-time:

$$y(t) = \frac{num}{den}u(t - \tau) + e(t)$$

where *num* and *den* are polynomials in the lag operator `q^(-1)`.

Multi-Input Multi-Output Models

A single-input single-output (SISO) continuous transfer function has the form $G(s) = \frac{num(s)}{den(s)}$. The corresponding transfer function model can be represented as:

$$Y(s) = G(s)U(s) + E(s)$$

A multi-input multi-output (MIMO) transfer function contains a SISO transfer function corresponding to each input-output pair in the system. For example, a continuous-time transfer function model with two inputs and two outputs has the form:

$$Y_1(s) = G_{11}(s)U_1(s) + G_{12}(s)U_2(s) + E_1(s)$$
$$Y_2(s) = G_{21}(s)U_1(s) + G_{22}(s)U_2(s) + E_2(s)$$

Where, $G_{ij}(s)$ is the SISO transfer function between the ith output and the jth input. $E_1(s)$ and $E_2(s)$ are the Laplace transforms of the noise corresponding to the two outputs.

The representation of discrete-time MIMO transfer function models is analogous.

See Also
idtf | tfest

More About

Data Supported by Transfer Function Models

You can estimate transfer function models from data with the following characteristics:

- Real data or complex data
- Single-output and multiple-output
- Time- or frequency-domain data

  Note that you cannot use time-series data for transfer function model identification.

You must first import your data into the MATLAB workspace, as described in "Data Preparation".

See Also

More About

- "Estimate Transfer Function Models in the System Identification App" on page 8-5
- "Estimate Transfer Function Models at the Command Line" on page 8-10

Estimate Transfer Function Models in the System Identification App

Use the app to set model configuration and estimation options for estimating a transfer function model.

Prerequisites

- Import data into the System Identification app. See "Represent Data". For supported data formats, see "Data Supported by Transfer Function Models" on page 8-4.

- 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 2-5.

1 In the System Identification app, select **Estimate > Transfer Function Models**
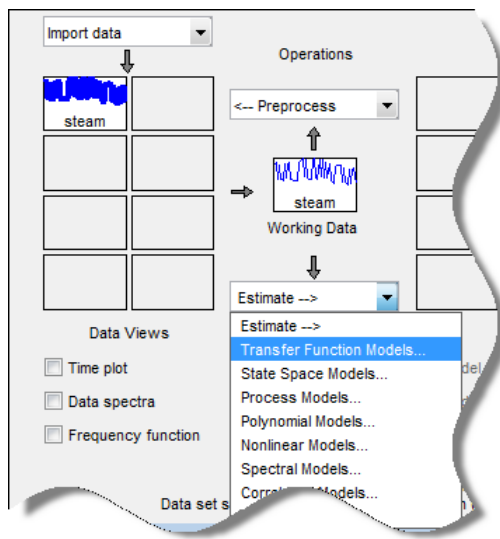
The Transfer Functions dialog box opens.
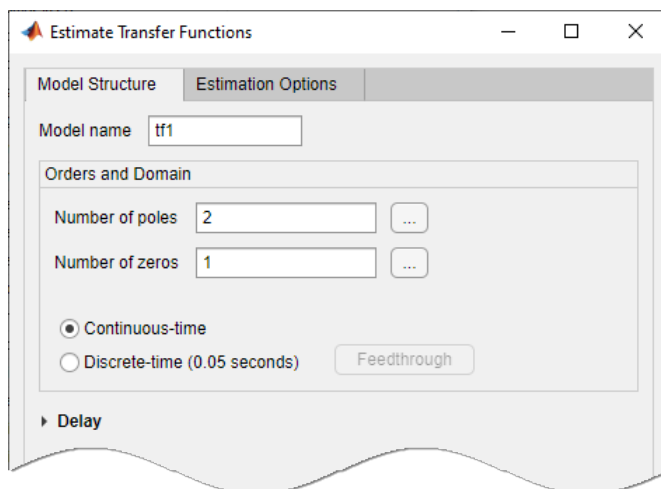
Tip For more information on the options in the dialog box, click **Help**.

2   In the **Number of poles** and **Number of zeros** fields, specify the number of poles and zeros of the transfer function as nonnegative integers.

Multi-Input, Multi-Output Models

For systems that are multiple input, multiple output, or both:

- To use the same number of poles or zeros for all the input/output pairs, specify a scalar.
- To use a different number of poles and zeros for the input/output pairs, specify an *ny*-by-*nu* matrix. *ny* is the number of outputs and *nu* is the number of inputs.

Alternatively, click ⌐···⌐.
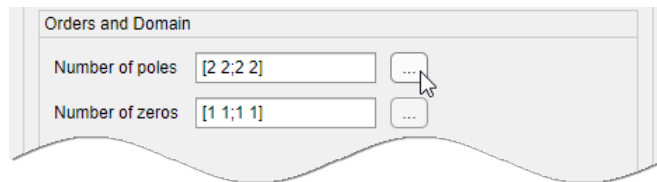
The Transfer Function Orders dialog box opens where you specify the number of poles and zeros for each input/output pair, as well as whether to include feedthrough if you have specified your model as **Discrete-time**. Use the **Output** list to select an output.
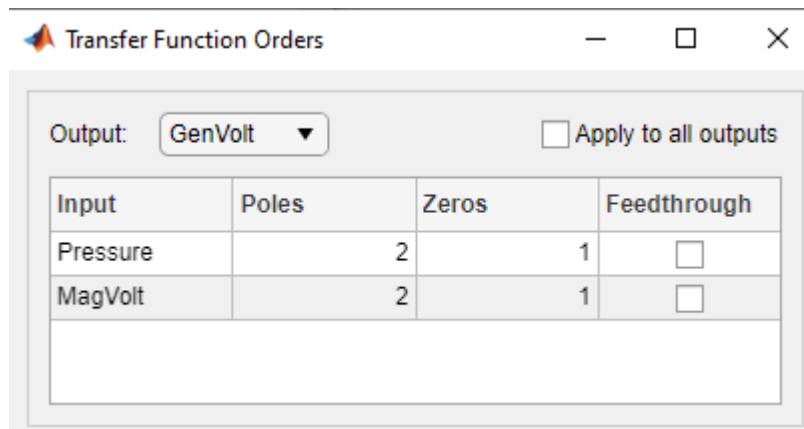
3   Select **Continuous-time** or **Discrete-time** to specify whether the model is a continuous- or discrete-time transfer function.

For discrete-time models, the number of poles and zeros refers to the roots of the numerator and denominator polynomials expressed in terms of the lag variable q^-1.

4   (For discrete-time models only) Specify whether to estimate the model feedthrough. Select the **Feedthrough** check box.
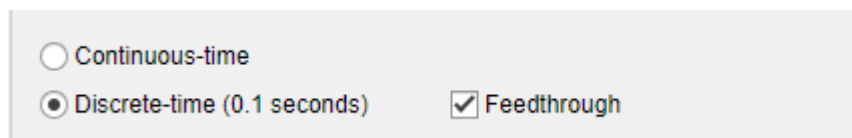
A discrete-time model with 2 poles and 3 zeros takes the following form:

$$H(z^{-1}) = \frac{b0 + b1z^{-1} + b2z^{-2} + b3z^{-3}}{1 + a1z^{-1} + a2z^{-2}}$$

When the model has direct feedthrough, b0 is a free parameter whose value is estimated along with the rest of the model parameters b1, b2, b3, a1, a2. When the model has no feedthrough, b0 is fixed to zero.

Multi-Input, Multi-Output Models

For models that are multi input, multi output or both, click **Feedthrough**.
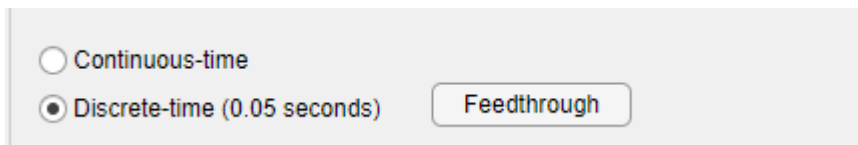
The Model Orders dialog box opens, where you specify to estimate the feedthrough for each input/output pair separately. Use the **Output** list to select an output.
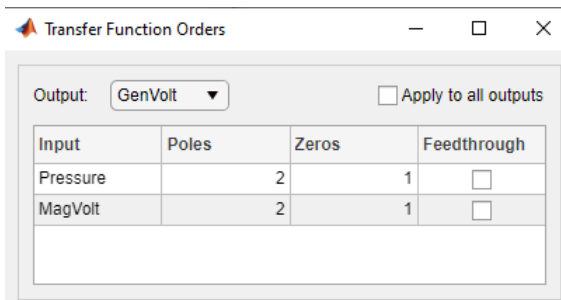
5   Expand the **Delay** section to specify nominal values and constraints for transport delays for different input/output pairs.
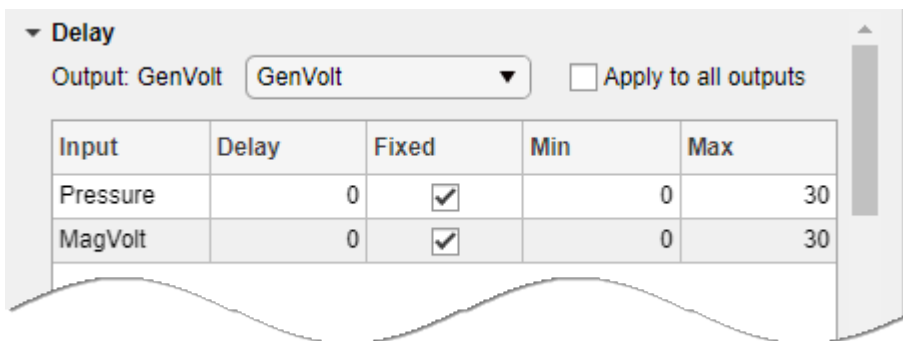
Use the **Output** list to select an output. Select the **Fixed** check box to specify a transport delay as a fixed value. Specify its nominal value in the **Delay** field.

6   Expand the **Estimation Options** section to specify estimation options.
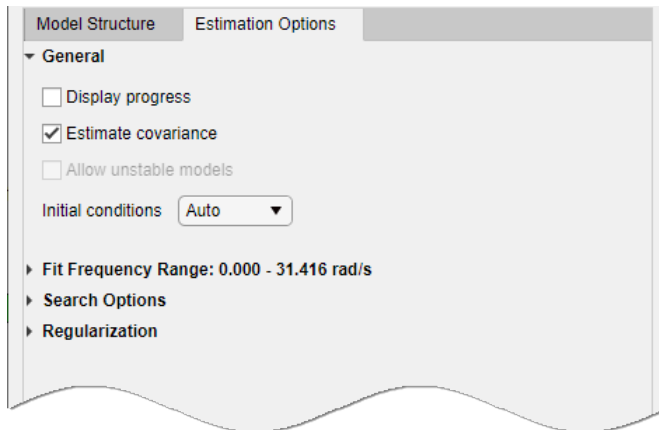
- Select **Display progress** to view the progress of the optimization.
- Select **Estimate covariance** to estimate the covariance of the transfer function parameters.
- (For frequency-domain data only) Specify whether to allow the estimation process to use parameter values that may lead to unstable models. Select the **Allow unstable models** option. An unstable model is delivered only if it produces a better fit to the data than other stable models computed during the estimation process.
- Specify how to treat the initial conditions in the **Initial condition** list. For more information, see "Specifying Initial Conditions for Iterative Estimation of Transfer Functions" on page 8-17.
- Expand **Fit Frequency Range** and set the range sliders to the desired passband to specify the frequency range over which the transfer function model must fit the data. By default the entire frequency range (0 to Nyquist frequency) is covered.

7 Expand **Search Options** to specify options for controlling the search iterations.

Search Options

In the **Search Options** section of **Estimation Options**, you can specify the following options:

- **Search method** — Method used by the iterative search algorithm. Search method is `auto` by default. Search methods include `Gauss-Newton(gn)`, `Adaptive Gauss-Newton(gna)`, `Levenberg-Marquardt(lm)`, `Trust-Region Reflective Newton (lsqnonlin)`, `Gradient Search (grad)`, `Sequential Quadratic Programming (fmincon:sqp)`, or `Interior Point (fmincon: interior-point)`. The descent direction is calculated successively at each iteration until a sufficient reduction in error is achieved.

    Output weight — 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 iterations** — Maximum number of iterations to use during search.
- **Tolerance** — Tolerance value when the iterations should terminate.
- **Outlier threshold** — Robustification of the quadratic criterion of fit.

8 Expand **Regularization** to obtain regularized estimates of model parameters. Specify the regularization constants *Lambda* and *R*.
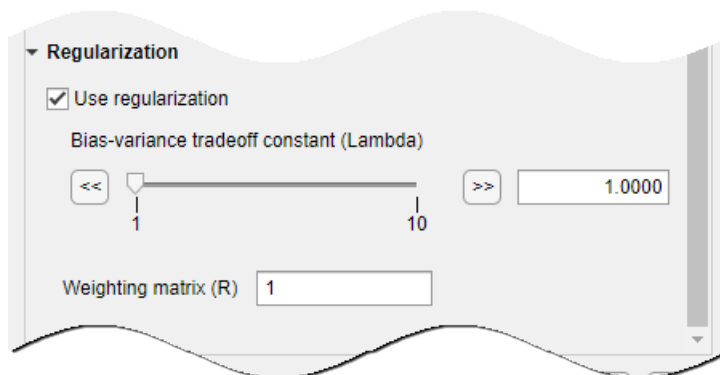
To learn more about regularization, see "Regularized Estimates of Model Parameters" on page 1-34.

9 Click **Estimate** to estimate the model. A new model gets added to the System Identification app.

Next Steps

- Validate the model by selecting the appropriate check box in the **Model Views** area of the System Identification app. For more information about validating models, see "Validating Models After Estimation" on page 17-2.

- Export the model to the MATLAB workspace for further analysis. Drag the model to the **To Workspace** rectangle in the System Identification app.

Estimate Transfer Function Models at the Command Line

General workflow for estimating 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 2-34. For supported data formats, see "Data Supported by Transfer Function Models" on page 8-4.
- 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 2-5.

  Alternatively, you can specify the input/output offset for the data using an estimation option set. Use `tfestOptions` to create the estimation option set. Use the `InputOffset` and `OutputOffset` name and value pairs to specify the input/output offset.

Estimate continuous-time and discrete-time transfer function models using `tfest`. The output of `tfest` is an `idtf` object, which represents the identified transfer function.

The general workflow in estimating a transfer function model is:

1 Create a data object (`iddata` or `idfrd`) that captures the experimental data.
2 (Optional) Specify estimation options using `tfestOptions`.
3 (Optional) Create a transfer function model that specifies the expected model structure and any constraints on the estimation parameters.
4 Use `tfest` to identify the transfer function model, based on the data.
5 Validate the model. See "Model Validation".

See Also

Related Examples
- "Troubleshoot Frequency-Domain Identification of Transfer Function Models" on page 8-18

Transfer Function Structure Specification

You can use a priori knowledge of the expected transfer function model structure to initialize the estimation. The `Structure` property of an `idtf` model contains parameters that allow you to specify the values and constraints for the numerator, denominator and transport delays.

For example, specify a third-order transfer function model that contains an integrator and has a transport delay of at most 1.5 seconds:

```
init_sys = idtf([nan nan],[1 2 1 0]);
init_sys.Structure.IODelay.Maximum = 1.5;
init_sys.Structure.Denominator.Free(end) = false;
```

`int_sys` is an `idtf` model with three poles and one zero. The denominator coefficient for the `s^0` term is zero and implies that one of the poles is an integrator.

`init_sys.Structure.IODelay.Maximum = 1.5` constrains the transport delay to a maximum of 1.5 seconds. The last element of the denominator coefficients (associated with the `s^0` term) is not a free estimation variable. This constraint forces one of the estimated poles to be at `s = 0`.

For more information regarding configuring the initial parameterization of an estimated transfer function, see `Structure` in `idtf`.

Estimate Transfer Function Models by Specifying Number of Poles

This example shows how to identify a transfer function containing a specified number of poles for given data.

Load time-domain system response data and use it to estimate a transfer function for the system.

```
load iddata1 z1;
np = 2;
sys = tfest(z1,np);
```

z1 is an `iddata` object that contains time-domain, input-output data.

np specifies the number of poles in the estimated transfer function.

sys is an `idtf` model containing the estimated transfer function.

To see the numerator and denominator coefficients of the resulting estimated model `sys`, enter:

```
sys.Numerator;
sys.Denominator;
```

To view the uncertainty in the estimates of the numerator and denominator and other information, use `tfdata`.

Estimate Transfer Function Models with Transport Delay to Fit Given Frequency-Response Data

This example shows how to identify a transfer function to fit a given frequency response data (FRD) containing additional phase roll off induced by input delay.

This example requires a Control System Toolbox™ license.

Obtain frequency response data.

For this example, use `bode` to obtain the magnitude and phase response data for the following system:

$$H(s) = e^{-.5s}\frac{s + 0.2}{s^3 + 2s^2 + s + 1}$$

Use 100 frequency points, ranging from 0.1 rad/s to 10 rad/s, to obtain the frequency response data. Use `frd` to create a frequency-response data object.

```
freq = logspace(-1,1,100);
[mag, phase] = bode(tf([1 .2],[1 2 1 1],'InputDelay',.5),freq);
data = frd(mag.*exp(1j*phase*pi/180),freq);
```

`data` is an `iddata` object that contains frequency response data for the described system.

Estimate a transfer function using `data`. Specify an unknown transport delay for the identified transfer function.

```
np = 3;
nz = 1;
iodelay = NaN;
sys = tfest(data,np,nz,iodelay);
```

`np` and `nz` specify the number of poles and zeros in the identified transfer function, respectively.

`iodelay` specifies an unknown transport delay for the identified transfer function.

`sys` is an `idtf` model containing the identified transfer function.

Estimate Transfer Function Models with Prior Knowledge of Model Structure and Constraints

This example shows how to estimate a transfer function model when the structure of the expected model is known and apply constraints to the numerator and denominator coefficients.

Load time-domain data.

```
load iddata1 z1;
z1.y = cumsum(z1.y);
```

`cumsum` integrates the output data of `z1`. The estimated transfer function should therefore contain an integrator.

Create a transfer function model with the expected structure.

```
init_sys = idtf([100 1500],[1 10 10 0]);
```

`int_sys` is an `idtf` model with three poles and one zero. The denominator coefficient for the `s^0` term is zero which indicates that `int_sys` contains an integrator.

Specify constraints on the numerator and denominator coefficients of the transfer function model. To do so, configure fields in the `Structure` property:

```
init_sys.Structure.Numerator.Minimum = eps;
init_sys.Structure.Denominator.Minimum = eps;
init_sys.Structure.Denominator.Free(end) = false;
```

The constraints specify that the numerator and denominator coefficients are nonnegative. Additionally, the last element of the denominator coefficients (associated with the `s^0` term) is not an estimable parameter. This constraint forces one of the estimated poles to be at `s = 0`.

Create an estimation option set that specifies using the Levenberg–Marquardt search method.

```
opt = tfestOptions('SearchMethod','lm');
```

Estimate a transfer function for `z1` using `init_sys` and the estimation option set.

```
sys = tfest(z1,init_sys,opt);
```

`tfest` uses the coefficients of `init_sys` to initialize the estimation of `sys`. Additionally, the estimation is constrained by the constraints you specify in the `Structure` property of `init_sys`. The resulting `idtf` model `sys` contains the parameter values that result from the estimation.

Estimate Transfer Function Models with Unknown Transport Delays

This example shows how to estimate a transfer function model with unknown transport delays and apply an upper bound on the unknown transport delays.

Create a transfer function model with the expected numerator and denominator structure and delay constraints.

For this example, the experiment data consists of two inputs and one output. Both transport delays are unknown and have an identical upper bound. Additionally, the transfer functions from both inputs to the output are identical in structure.

```
init_sys = idtf(NaN(1,2),[1, NaN(1,3)],'IODelay',NaN);
init_sys.Structure(1).IODelay.Free = true;
init_sys.Structure(1).IODelay.Maximum = 7;
```

`init_sys` is an `idtf` model describing the structure of the transfer function from one input to the output. The transfer function consists of one zero, three poles and a transport delay. `NaN` indicates unknown coefficients.

`init_sys.Structure(1).IODelay.Free = true` indicates that the transport delay is not fixed.

`init_sys.Structure(1).IODelay.Maximum = 7` sets the upper bound for the transport delay to 7 seconds.

Specify the transfer function from both inputs to the output.

```
init_sys = [init_sys,init_sys];
```

Load time-domain system response data and detrend the data.

```
load co2data;
Ts = 0.5;
data = iddata(Output_exp1,Input_exp1,Ts);
T = getTrend(data);
T.InputOffset = [170,50];
T.OutputOffset = mean(data.y(1:75));
data = detrend(data, T);
```

Identify a transfer function model for the measured data using the specified delay constraints.

```
sys = tfest(data,init_sys);
```

`sys` is an `idtf` model containing the identified transfer function.

Estimate Transfer Functions with Delays

This example shows how to estimate transfer function models with I/O delays.

The `tfest` command supports estimation of IO delays. In the simplest case, if you specify `NaN` as the value for the `IODelay` input argument, `tfest` estimates the corresponding delay value.

```
load iddata1 z1
sys = tfest(z1,2,2,NaN); % 2 poles, 2 zeros, unknown transport delay
```

If you want to assign an initial guess to the value of delay or prescribe bounds for its value, you must first create a template `idtf` model and configure `IODelay` using the model's `Structure` property:

```
sys0 = idtf([nan nan nan],[1 nan nan]);
sys0.Structure.IODelay.Value = 0.1; % initial guess
sys0.Structure.IODelay.Maximum = 1; % maximum allowable value for delay
sys0.Structure.IODelay.Free = true; % treat delay as estimatable quantity
sys = tfest(z1,sys0);
```

If estimation data is in the time-domain, the delays are not estimated iteratively. If a finite initial value is specified, that value is retained as is with no iterative updates. The same is true of discrete-time frequency domain data. Thus in the example above, if `data` has a nonzero sample time, the estimated value of delay in the returned model `sys` is 0.1 (same as the initial guess specified for `sys0`). The delays are updated iteratively only for continuous-time frequency domain data. If, on the other hand, a finite initial value for delay is not specified (e.g., `sys0.Structure.IODelay.Value = NaN`), then a value for delay is determined using the `delayest` function, regardless of the nature of the data.

Determination of delay as a quantity independent of the model's poles and zeros is a difficult task. Estimation of delays becomes especially difficult for multi-input or multi-output data. It is strongly recommended that you perform some investigation to determine delays before estimation. You can use functions such as `delayest`, `arxstruc`, `selstruc` and impulse response analysis to determine delays. Often, physical knowledge of the system or dedicated transient tests (how long does it take for a step change in input to show up in a measured output?) will reveal the value of transport delays. Use the results of such analysis to assign initial guesses as well as minimum and maximum bounds on the estimated values of delays.

Specifying Initial Conditions for Iterative Estimation of Transfer Functions

If you estimate transfer function models using `tfest`, you can specify how the algorithm treats initial conditions.

In the System Identification app, set **Initial condition** to one of the following options:

- `auto` — Automatically chooses `Zero`, `Estimate`, or `Backcast` based on the estimation data. If initial conditions have negligible effect on the prediction errors, the initial conditions are set to zero to optimize algorithm performance.
- `Zero` — Sets all initial conditions to zero.
- `Estimate` — Treats the initial conditions as an estimation parameters.
- `Backcast` — Estimates initial conditions using a backward filtering method (least-squares fit).

At the command line, specify the initial conditions by using an estimation option set. Use `tfestOptions` to create the estimation option set. For example, create an options set that sets the initial conditions to zero:

```
opt = tfestOptions('InitialCondition','zero');
```

See Also
`tfest` | `tfestOptions`

More About
- "Estimate Transfer Function Models in the System Identification App" on page 8-5
- "Estimate Transfer Function Models at the Command Line" on page 8-10

Troubleshoot Frequency-Domain Identification of Transfer Function Models

This example shows how to perform and troubleshoot the identification of a SISO system using frequency-response data (FRD). The techniques explained here can also be applied to MIMO models and frequency-domain input-output data.

When you use the `tfest` command to estimate a SISO transfer function model from the frequency-response data, the estimation algorithm minimizes the following least-squares loss (cost) function:

$$\underset{G(\omega)}{\text{minimize}} \sum_{k=1}^{N_f} |W(\omega_k)(G(\omega_k) - f(\omega_k))|^2$$

Here `W` is a frequency-dependent weight that you specify, `G` is the transfer function that is to be estimated, `f` is the measured frequency-response data, and w is the frequency. `Nf` is the number frequencies at which the data is available. $G(w) - f(w)$ is the frequency-response error.

In this example, you first estimate the model without preprocessing the data or using estimation options to specify a weighting filter. You then apply these troubleshooting techniques to improve the model estimation.

Estimate the Model Without Preprocessing and Filtering

Load the measured continuous-time frequency response data.

```
load troubleshooting_example_data Gfrd;
```

`Gfrd` is an `idfrd` object that stores the data.

Estimate an initial transfer function model with 11 poles and 10 zeros by using the estimation data.

```
Gfit = tfest(Gfrd,11,10);
```

Plot the frequency-response magnitude of the estimated model and the measured frequency-response data.

```
bodemag(Gfrd,Gfit);
ylim([-100 0])
legend('Measured','Estimated')
```
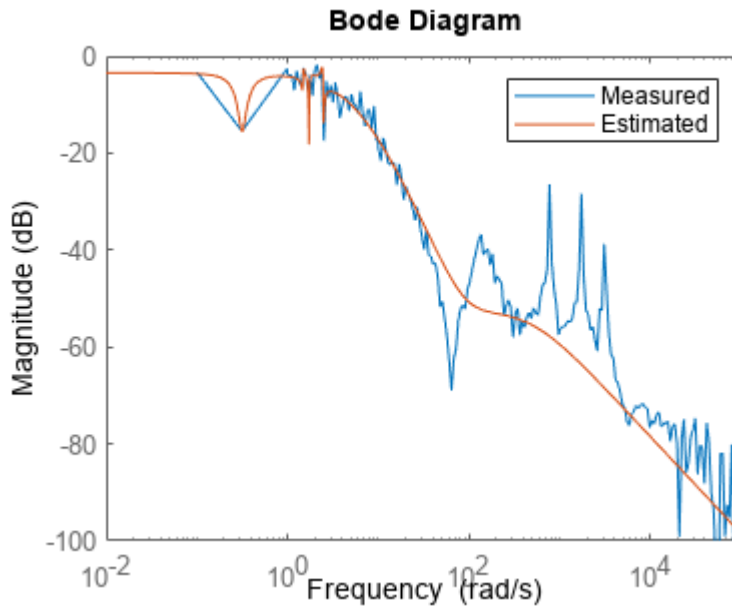
The estimated model contains spurious dynamics. The estimation algorithm misses the valley at 60 rad/s and the resonant peaks after that. Therefore, the model is not a good fit to the data.

The algorithm minimizes the squared error magnitude, $|W(w) (G(w) - f(w))|^2$, in the loss function. Plot this quantity as a function of frequency. This error plot provides a view of which data points contribute the most to the loss function, and so are the likely limiting factors during estimation. The plot can help you identify why there are spurious or uncaptured dynamics.

Because you have not specified a frequency-dependent weight, $W(w)$ is 1.

```
w = Gfrd.Frequency;
r1 = squeeze(freqresp(Gfit,w));
r2 = squeeze(freqresp(Gfrd,w));
fitError = r1-r2;
semilogx(w,abs(fitError).^2)
title('Weighted Estimation Error');
xlabel('Frequency (rad/s)');
ylabel('Magnitude (abs)')
```
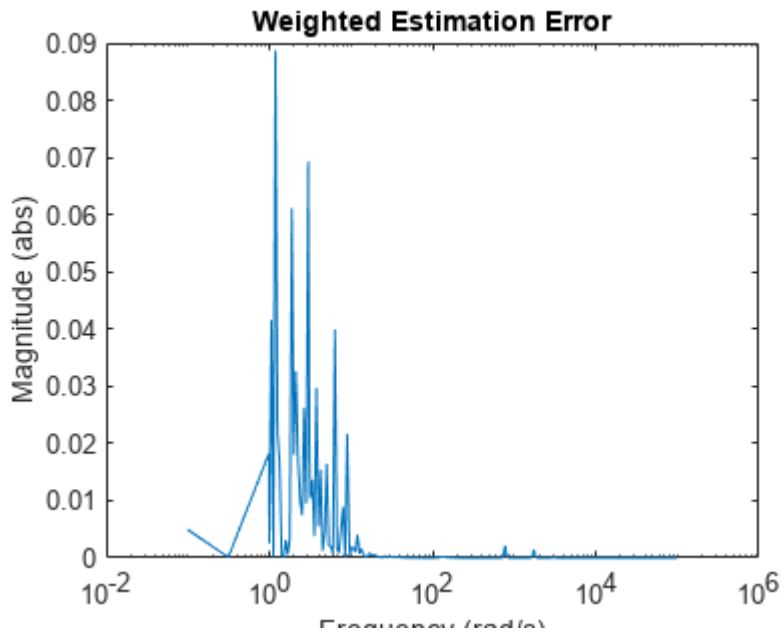
From the data, model, and error plots you can see that:

- The largest fitting errors are below 10 rad/s.
- The algorithm focusses on fitting the noisy high magnitude data points below 10 rad/s, which have a large contribution to the optimization loss function. As a result, the algorithm assigns spurious poles and zeros to this data region. To address this issue, you can preprocess the data to improve signal-to-noise ratio in this region. You can also use frequency-dependent weights to make the algorithm put less focus on this region.
- Below approximately 40 rad/s, most variations in data are due to noise. There are no significant system modes (valleys or peaks) in the data. To address this issue, you can use a moving-average filter over the data to smooth the measured response.
- The algorithm ignores the valley around 60 rad/s and the three lightly damped resonant peaks that follow it. These features contribute little to the loss function because the fitting error is small at these frequencies. To address this issue, you can specify frequency-dependent weights to make the algorithm pay more attention to these frequencies.

Preprocess Data

To improve the estimated model quality, preprocess the data. To do so, you truncate the low signal-to-noise portions of data below 1 rad/s and above 2e4 rad/s that are not interesting. Then you use a moving-average filter to smooth data in the low-frequency high-magnitude region below 40 rad/s. At these frequencies, the data has a low signal-to-noise ratio, but has dynamics that you are interested in capturing. Do not apply the filter at frequencies above 40 rad/s to avoid smoothing data where you see the valley and the three peaks that follow it.

Make a copy of the original `idfrd` data object.

```
GfrdProcessed = Gfrd;
```

Truncate the data below 1 rad/s and above 2e4 rad/s.

```
GfrdProcessed = fselect(GfrdProcessed,1,2e4);
```

Apply a three-point centered moving-average filter to smooth out the frequency-response data below 40 rad/s that contains spurious dynamics. The response data is stored in the `ResponseData` property of the object.

```
w = GfrdProcessed.Frequency;
f = squeeze(GfrdProcessed.ResponseData);
idx = w<40;
f(idx) = movmean(f(idx),3);
```

Here `f(idx)` is the frequency-response data at frequencies less than 40 rad/s.

Place the filtered data back into the original data object.

```
GfrdProcessed.ResponseData = f;
```

Plot the original and preprocessed data.

```
bodemag(Gfrd,GfrdProcessed);
ylim([-100 0]);
legend('Original data','Preprocessed data');
```
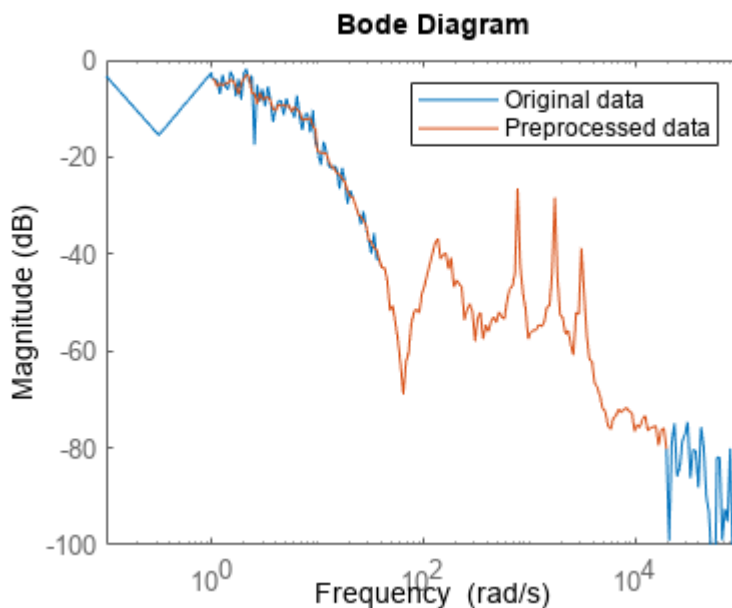

The plot shows that all desired dynamics are intact after preprocessing.

Specify Weighting Filter

Use a low weight for the low frequency region under 10 rad/s where spurious dynamics exist. This low weight and the smoothing applied earlier to this data reduce the chance of spurious peaks in the estimated model response in this region.

```
Weight = ones(size(f));
idx = w<10;
Weight(idx) = Weight(idx)/10;
```

Use a high weight for data in the frequency range 40-6e3 rad/s where you want to capture the dynamics but the response data magnitude is low.

```
idx = w>40 & w<6e3;
Weight(idx) = Weight(idx)*30;
```

Specify the weights in the `WeightingFilter` option of the estimation option set.

```
tfestOpt = tfestOptions('WeightingFilter',Weight);
```

Note that `Weight` is a custom weighting filter. You can also specify `WeightingFilter` as `'inv'` or `'invsqrt'` for frequency-response data. These options specify the weight as $1/|f(w)|$ and $1/\sqrt{|f(w)|}$, respectively. These options enable you to quickly test the effect of using a higher weight for low magnitude regions of data. `'invsqrt'` is typically a good initial choice. If these weights do not yield good estimation results, you can provide custom weights as shown in this example.

Estimate Model Using Preprocessed and Filtered Data

Estimate a transfer function model with 11 poles and 10 zeros using the preprocessed data and specified weighting filter.

```
GfitNew = tfest(GfrdProcessed,11,10,tfestOpt);
```

Plot the original data, the initial model response, and the new model response.

```
bodemag(Gfrd,Gfit,GfitNew);
ylim([-100 0]);
legend('Original Data','Original Model','New Model');
```
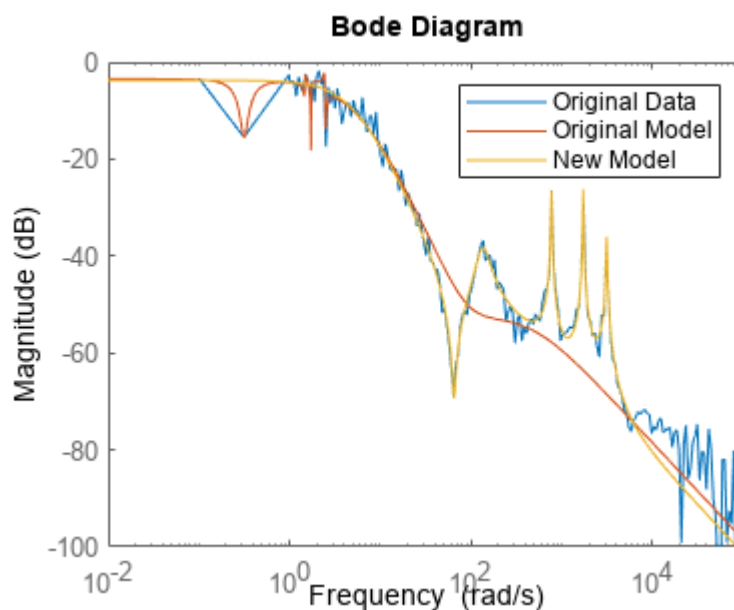

Plot the estimation error. Compute the estimation error by including the weighting filter `Weight` that you used for estimating `GfitNew`.

```
w = GfrdProcessed.Frequency;
r1 = squeeze(freqresp(GfitNew,w));
r2 = squeeze(freqresp(GfrdProcessed,w));
fitErrorNew = Weight.*(r1-r2);
semilogx(w,abs(fitErrorNew).^2)
title('Weighted Estimation Error');
```

```
xlabel('Frequency (rad/s)');
ylabel('Magnitude (abs)');
```
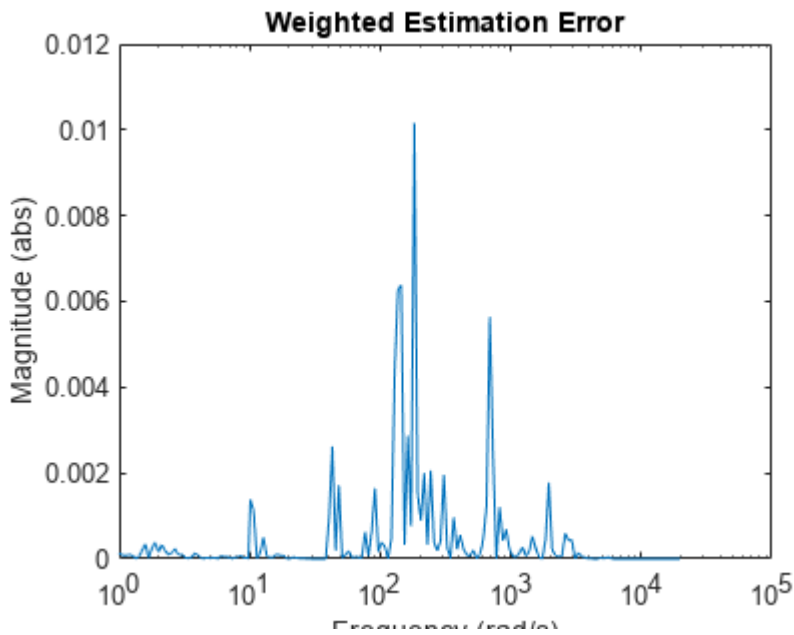

The new model successfully captures all system dynamics of interest.

You can use the weighted error plot for further troubleshooting if your initial choice of weights does not yield a satisfactory result.

See Also
`tfest | tfestOptions`

More About
- "Estimating Models Using Frequency-Domain Data"
- "Estimate Transfer Function Models at the Command Line" on page 8-10

Estimating Transfer Function Models for a Heat Exchanger

This example shows how to estimate a transfer function from measured signal data.

Heat Exchanger

In this example we estimate the transfer function for a heat exchanger. The heat exchanger consists of a coolant temperature, product temperature, and disturbance ambient temperature. We will estimate the coolant to product temperature transfer function.

Configuring the Measured Data

The measured data is stored in a MATLAB file and includes measurements of changes in coolant temperature around a nominal and changes in product temperature around a nominal.

```
load iddemo_heatexchanger_data
```

Collect the measured data using the `iddata` command and plot it.

```
data = iddata(pt,ct,Ts);
data.InputName  = '\Delta CTemp';
data.InputUnit  = 'C';
data.OutputName = '\Delta PTemp';
data.OutputUnit = 'C';
data.TimeUnit   = 'minutes';
plot(data)
```
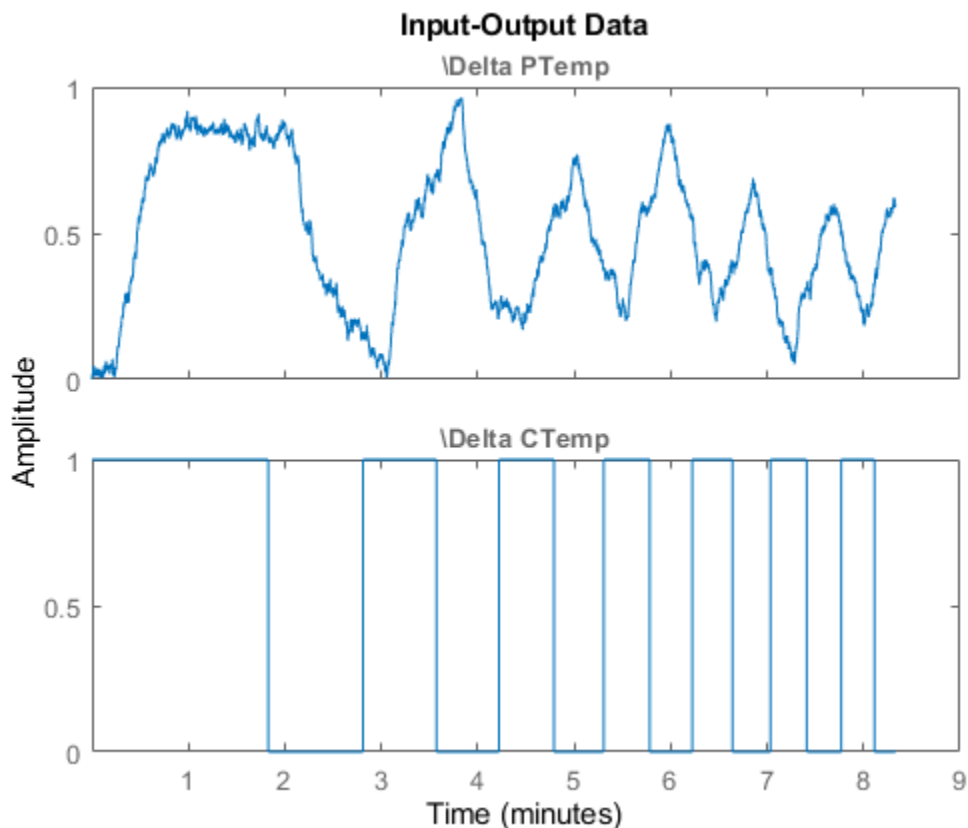
Transfer Function Estimation

From the physics of the problem we know that the heat exchanger can be described by a first order system with delay. Use the `tfest` command specifying one pole, no zeroes, and an unknown input/output delay to estimate a transfer function.

```
sysTF = tfest(data,1,0,nan)
```

```
sysTF =
  From input "\Delta CTemp" to output "\Delta PTemp":
                     1.467
  exp(-0.0483*s) * --------
                   s + 1.56

Continuous-time identified transfer function.

Parameterization:
   Number of poles: 1   Number of zeros: 0
   Number of free coefficients: 2
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using TFEST on time domain data "data".
Fit to estimation data: 36%
FPE: 0.02625, MSE: 0.02622
```

The `compare` and `resid` commands allow us to investigate how well the estimated model matches the measured data.

```
set(gcf,'DefaultAxesTitleFontSizeMultiplier',1,...
    'DefaultAxesTitleFontWeight','normal',...
    'Position',[100 100 780 520]);
resid(sysTF,data);
```
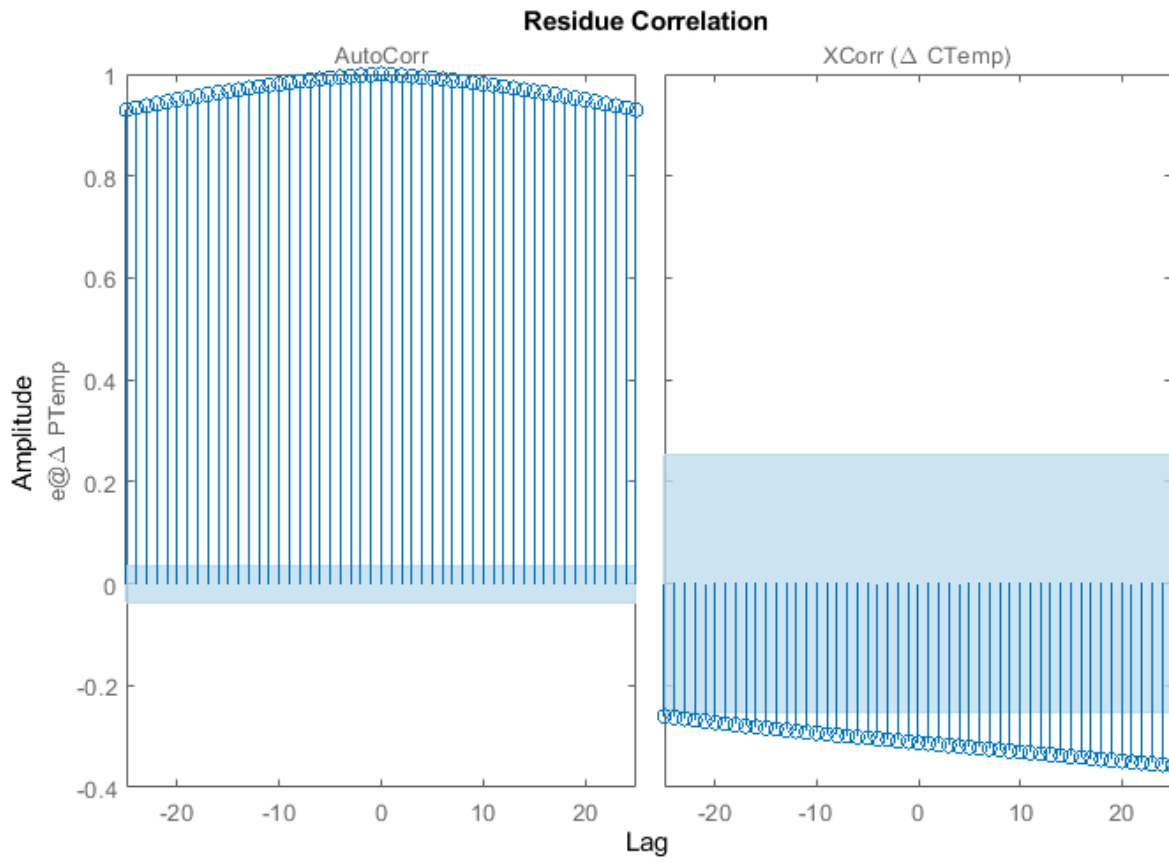
```
clf
compare(data,sysTF)
```
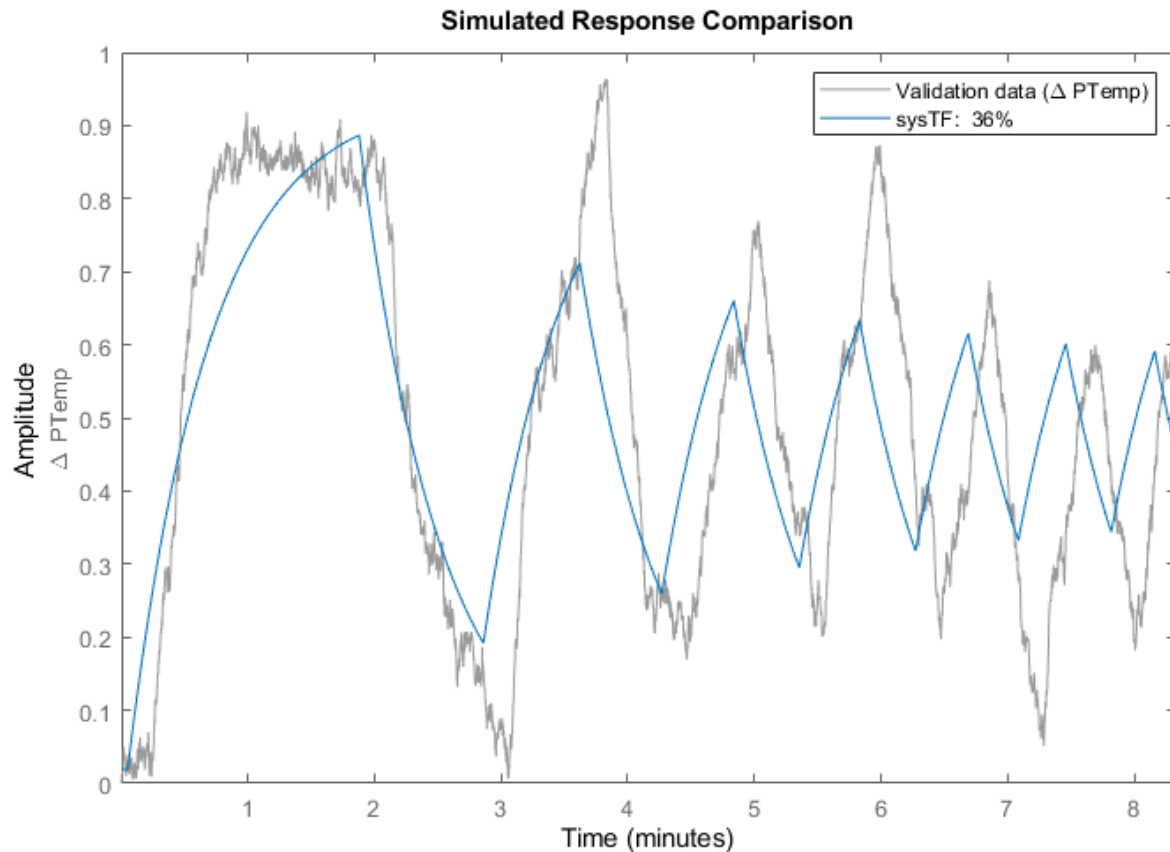
The figure shows that residuals are strongly correlated implying there is information in the measured data that has not been adequately captured by the estimated model.

Transfer Function Estimation from an Initial System

Previously we estimated a transfer function from data but apart for the system order did not include much apriori knowledge. From the physics of the problem we know that the system is stable and has positive gain. Inspecting the measured data we also know that the input/output delay is around 1/5 of a minute. We use this information to create an initial system and estimate a transfer function using this system as the initial guess.

```
sysInit = idtf(NaN,[1 NaN],'ioDelay',NaN);
sysInit.TimeUnit = 'minutes';
```

Restrict the transfer function numerator and denominator terms so that the system is stable with positive gain.

```
sysInit.Structure.num.Value   = 1;
sysInit.Structure.num.Minimum = 0;
sysInit.Structure.den.Value   = [1 1];
sysInit.Structure.den.Minimum = [0 0];
```

Restrict the input/output delay to the range [0 1] minute and use 1/5 minute as an initial value.

```
sysInit.Structure.ioDelay.Value   = 0.2;
sysInit.Structure.ioDelay.Minimum = 0;
sysInit.Structure.ioDelay.Maximum = 1;
```

Use the system as an initial guess for the estimation problem

```
sysTF_initialized = tfest(data,sysInit)
```

```
sysTF_initialized =
  From input "\Delta CTemp" to output "\Delta PTemp":
                    1.964
  exp(-0.147*s) * ---------
                  s + 2.115

Continuous-time identified transfer function.

Parameterization:
   Number of poles: 1    Number of zeros: 0
   Number of free coefficients: 2
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using TFEST on time domain data "data".
Fit to estimation data: 54.09%
FPE: 0.01351, MSE: 0.01349
```

```
resid(sysTF_initialized,data);
```

```
clf
compare(data,sysTF,sysTF_initialized)
```

Simulated Response Comparison
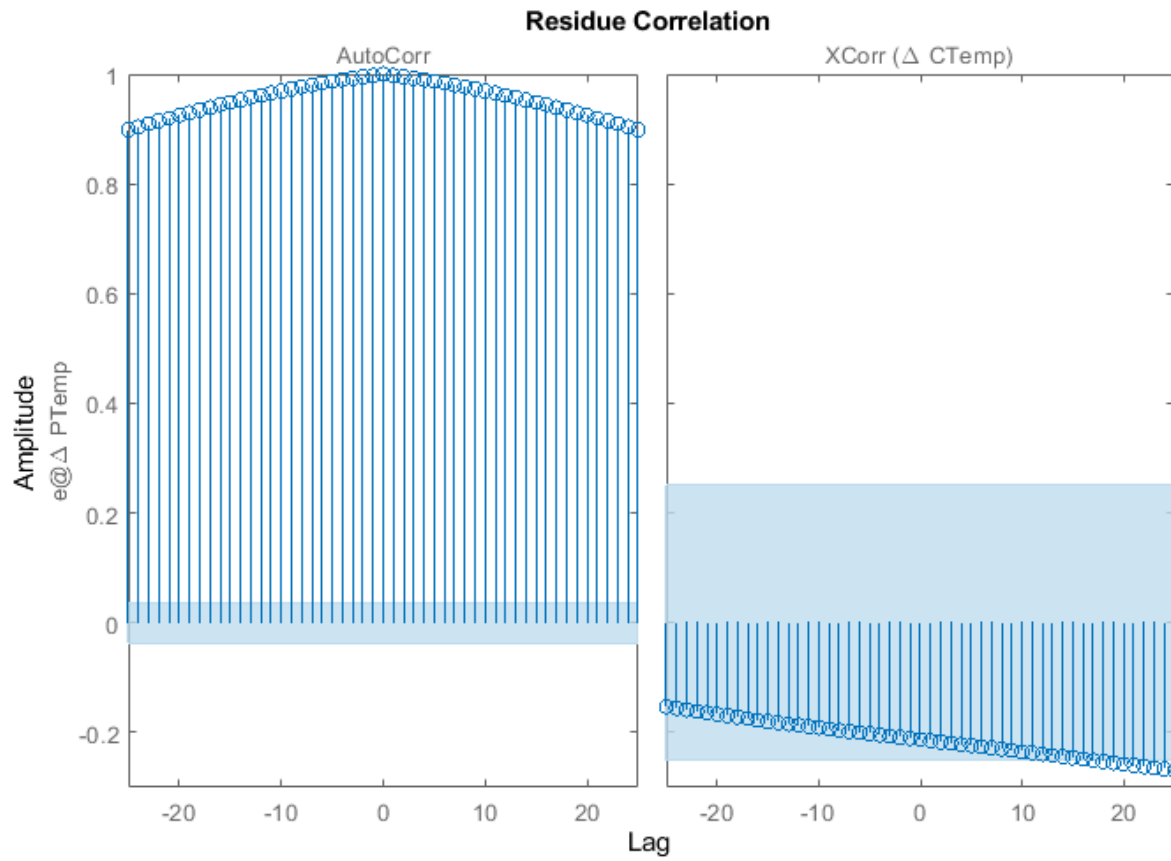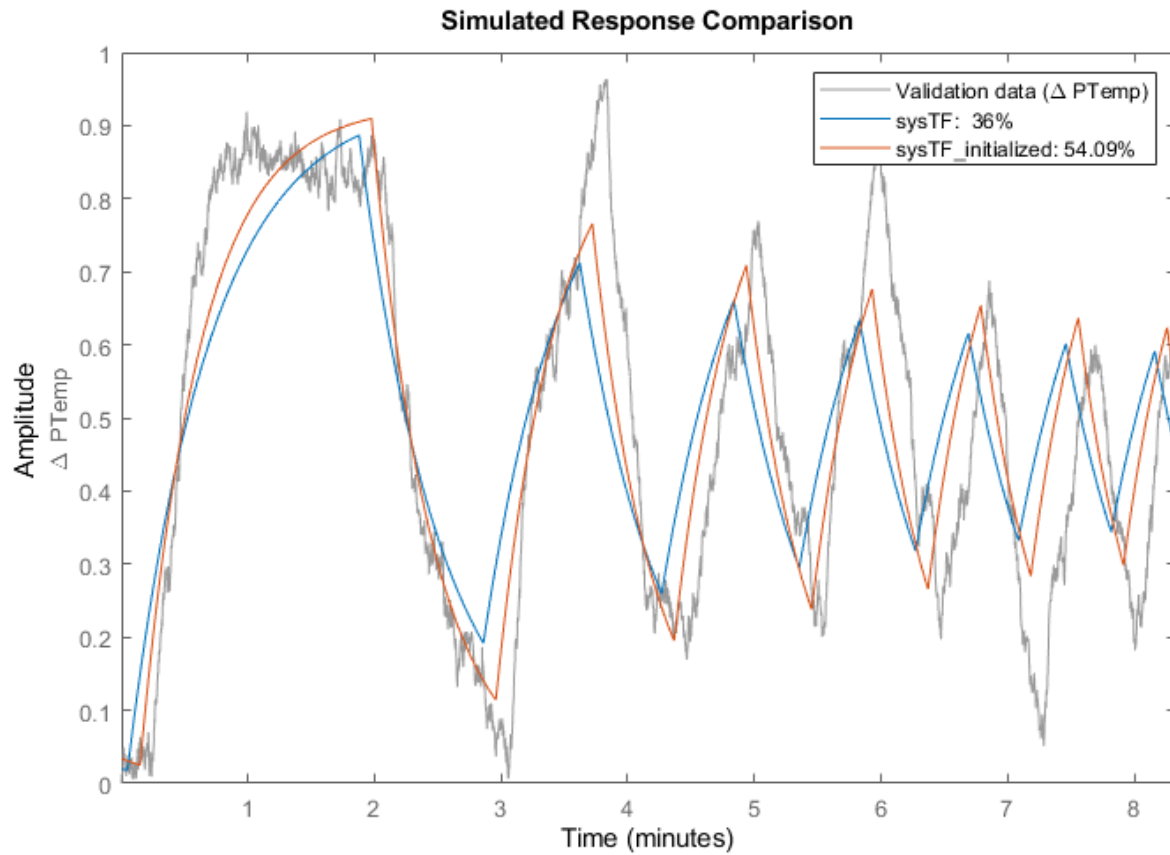
Process Model Estimation

In the above we treated the estimation problem as a transfer function estimation problem, but we know that there is some additional structure we can impose. Specifically the heat exchanger system is known to be a 1st order process with delay, or 'P1D' model of the form:

$$e^{-T_d s} \frac{K_p}{T_p s + 1}$$

Use the `procest` command to further impose this structure on the problem.

```
sysP1D  = procest(data,'P1D')
```

```
sysP1D =

Process model with transfer function:
             Kp
  G(s)  =  ---------- * exp(-Td*s)
            1+Tp1*s

         Kp = 0.90548
        Tp1 = 0.32153
         Td = 0.25435
```

```
Parameterization:
   {'P1D'}
   Number of free coefficients: 3
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "data".
Fit to estimation data: 70.4%
FPE: 0.005614, MSE: 0.005607
```

```
resid(sysP1D,data);
```



```
clf
compare(data,sysTF,sysTF_initialized,sysP1D)
```

Simulated Response Comparison
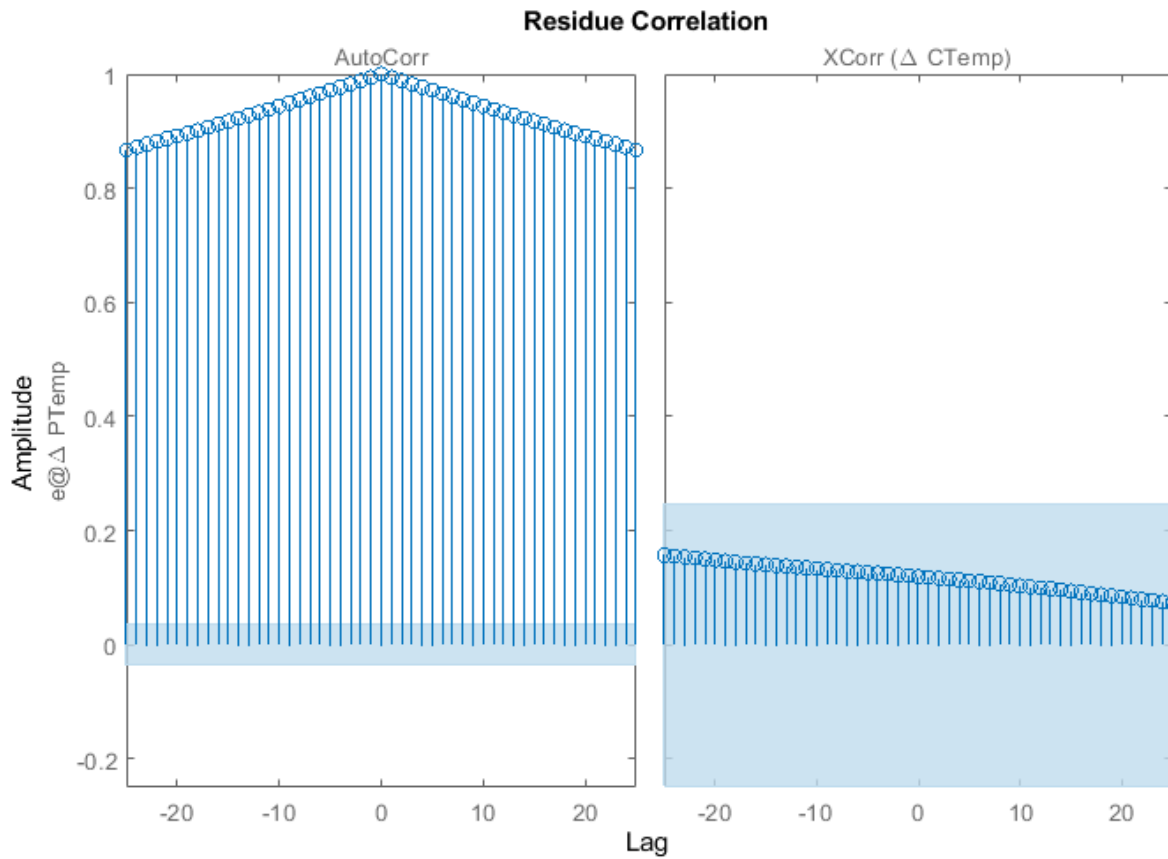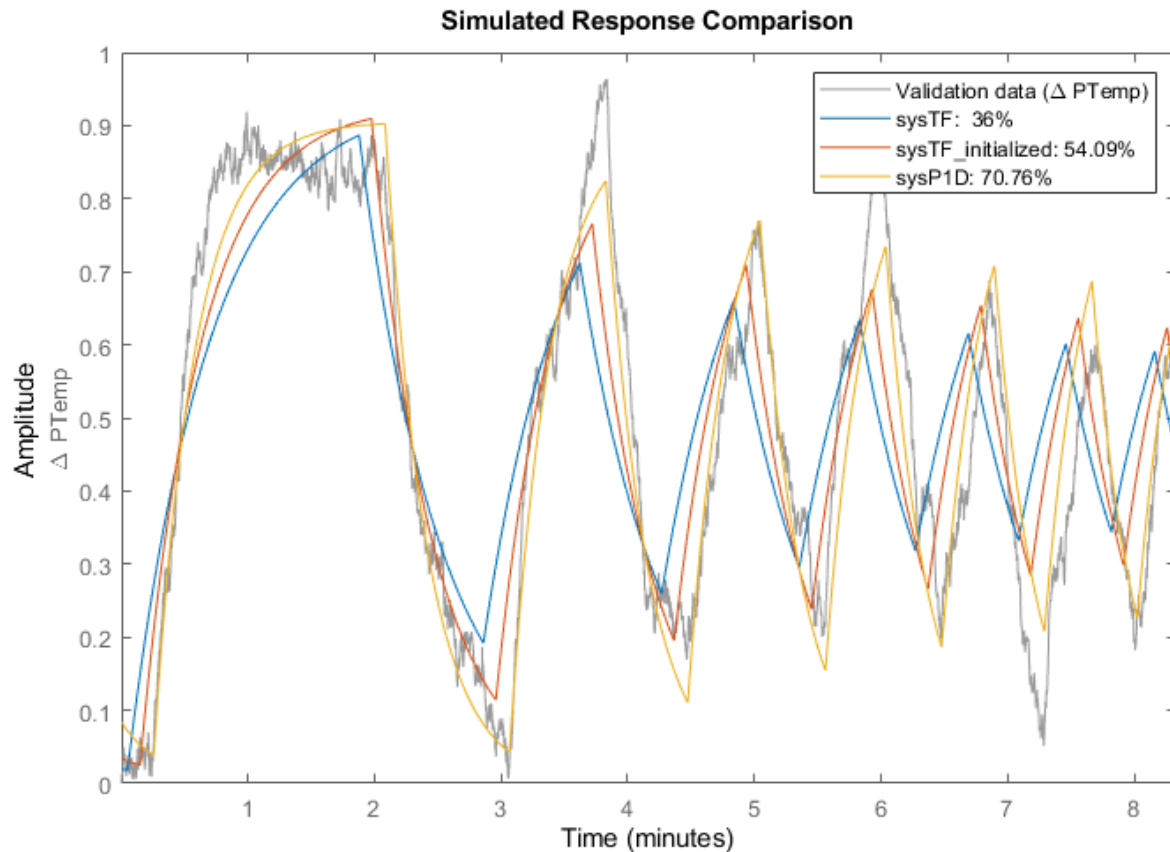
Process Model Estimation with Disturbance Model

The residual plots of all the estimations performed so far show that the residual correlation is still high, implying that the model is not rich enough to explain all the information in the measured data. The key missing piece is the disturbance ambient temperature which we have not yet included in the model.

Create a 'P1D' process model with restriction on delay and time constant values and use this as the initial guess for the estimation problem.

```
sysInit = idproc('P1D','TimeUnit','minutes');
```

Restrict the model to have positive gain, and delay in the range [0 1] minute.

```
sysInit.Structure.Kp.Value    = 1;
sysInit.Structure.Kp.Minimum  = 0;
sysInit.Structure.Tp1.Value   = 1;
sysInit.Structure.Tp1.Maximum = 10;
sysInit.Structure.Td.Value    = 0.2;
sysInit.Structure.Td.Minimum  = 0;
sysInit.Structure.Td.Maximum  = 1;
```

Specify the option to use a first-order model ('ARMA1') for the disturbance component. Use the template model `sysInit` along with the option set to estimate the model.

```matlab
opt = procestOptions('DisturbanceModel','ARMA1');
sysP1D_noise = procest(data,sysInit,opt)
```

```
sysP1D_noise =

Process model with transfer function:
              Kp
  G(s) = ---------- * exp(-Td*s)
          1+Tp1*s

         Kp = 0.91001
        Tp1 = 0.3356
         Td = 0.24833

An additive ARMA disturbance model has been estimated
      y = G u + (C/D)e

      C(s) = s + 591.6
      D(s) = s + 3.217

Parameterization:
    {'P1D'}
   Number of free coefficients: 5
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "data".
Fit to estimation data: 96.86% (prediction focus)
FPE: 6.307e-05, MSE: 6.294e-05
```

```matlab
resid(sysP1D_noise,data);
```
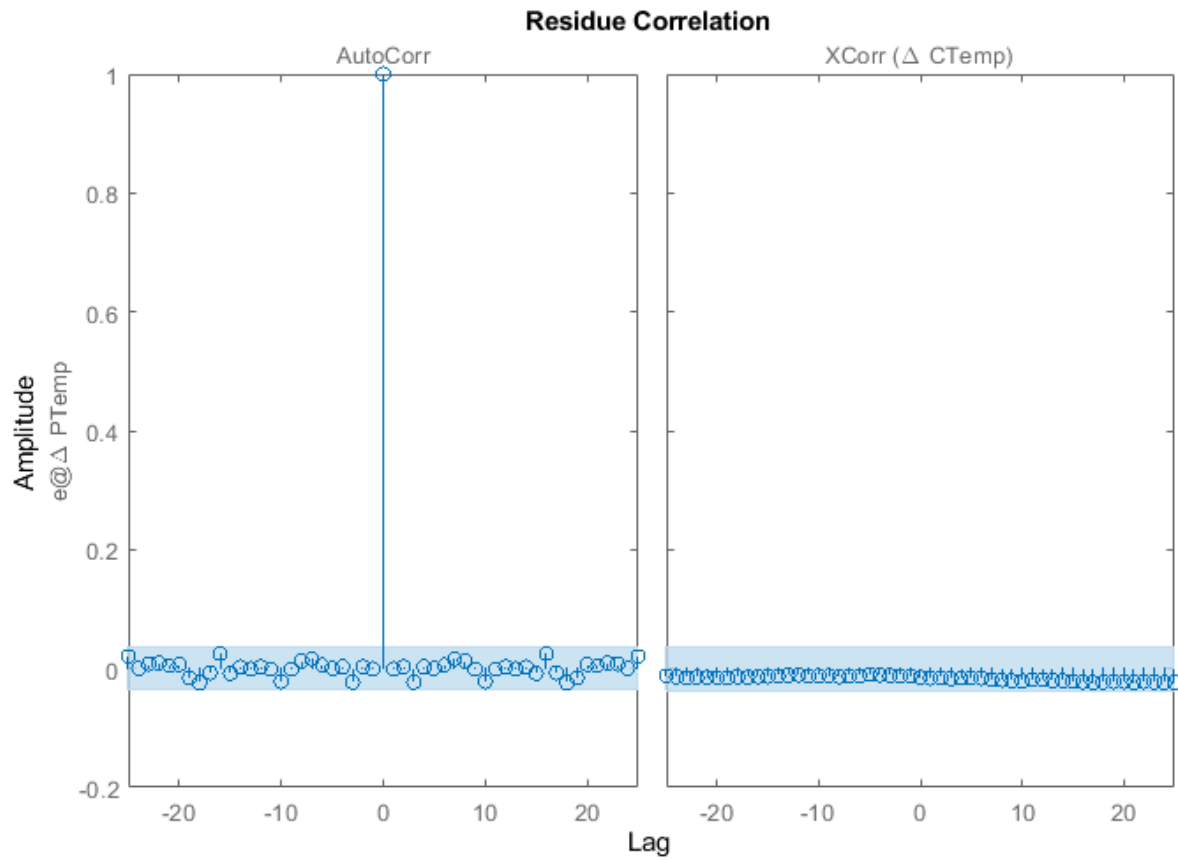
```
clf
compare(data,sysTF,sysTF_initialized,sysP1D,sysP1D_noise)
```
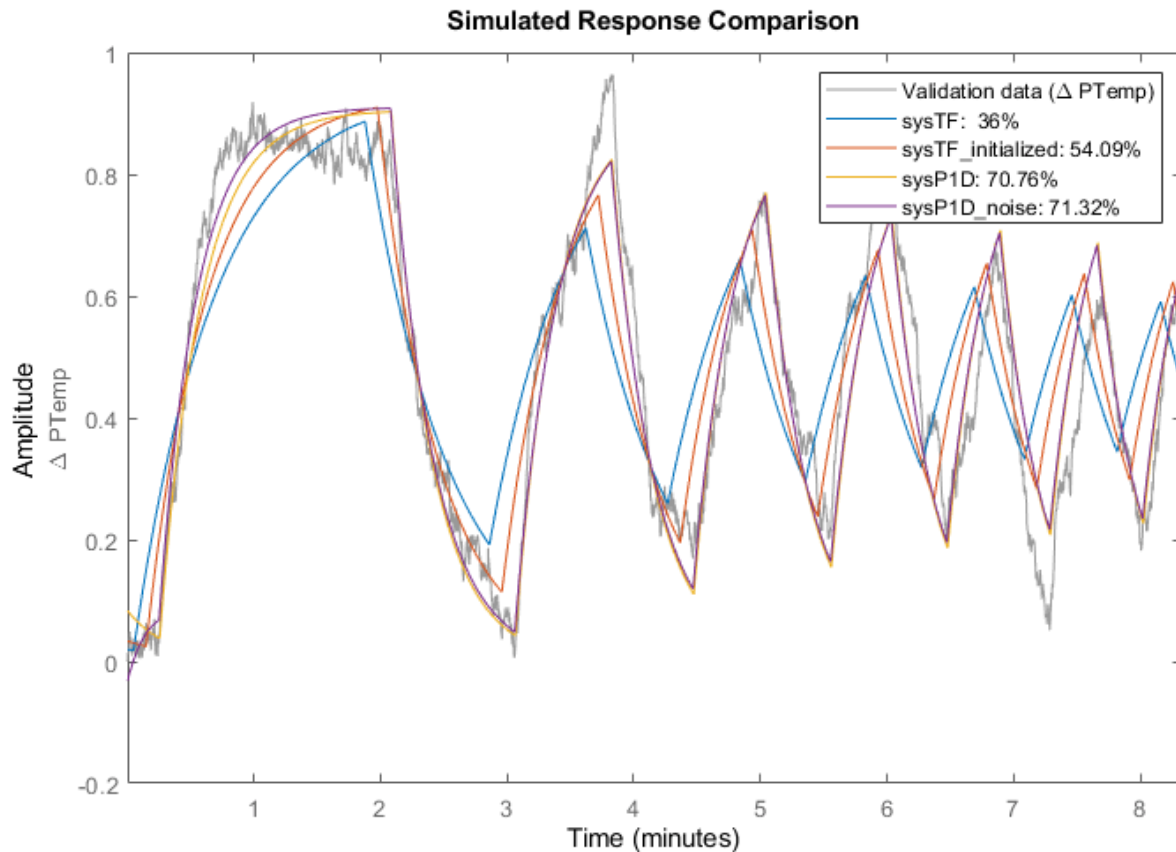
The residual plot clearly shows that the residuals are uncorrelated implying that we have a model that explains the measured data. The 'ARMA1' disturbance component we estimated is stored as numerator and denominator values in the "NoiseTF" property of the model.

```
sysP1D_noise.NoiseTF
```

```
ans =

  struct with fields:

    num: {[1 591.6038]}
    den: {[1 3.2172]}
```

Compare Different Models

Although we have identified a model that explains the measured data we note that the model fit to measured data is around 70%. The loss in fit value is a consequence of the strong effect of the ambient temperature disturbance which is illustrated as follows.

The measured data was obtained from a Simulink model with the following exact values (d is a Gaussian noise disturbance)

```
y = (1-pi/100)*exp(-15s)/(21.3s+1)*u + 1/(25s+1)*d
```

Create a 'P1D' model with these values and see how well that model fits the measured data.

```
sysReal = idproc('P1D','TimeUnit','minutes');
sysReal.Kp  = 1-pi/100;
sysReal.Td  = 15/60;
sysReal.Tp1 = 21.3/60;
sysReal.NoiseTF = struct('num',{[1 10000]},'den',{[1 0.04]});

compare(data,sysReal,sysTF,sysTF_initialized,sysP1D,sysP1D_noise);
```
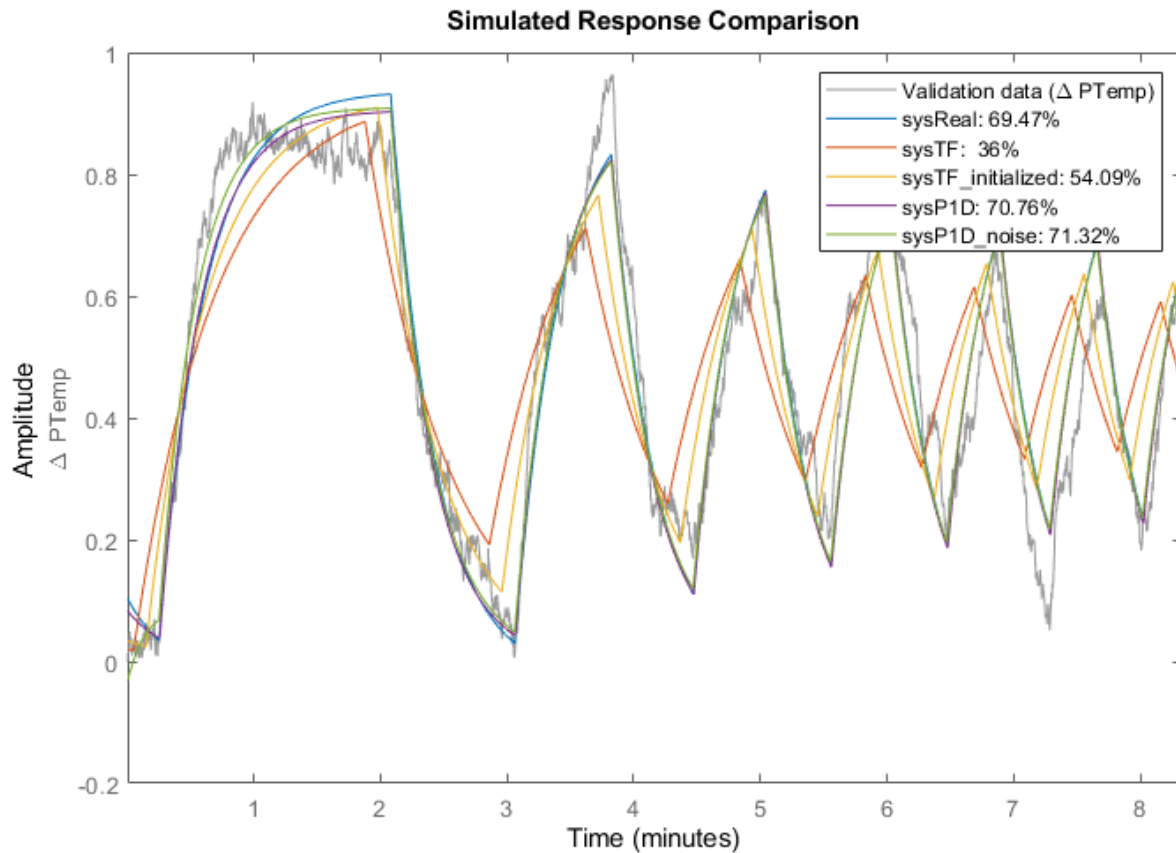

The comparison plot shows that the true system fit to measured data is also around 70% confirming that our estimated models are no worse than the real model in fitting measured data - a consequence of the strong effect of the ambient temperature disturbance.

Additional Information

For more information on identification of dynamic systems with System Identification Toolbox visit the System Identification Toolbox product information page.

Estimating Transfer Function Models for a Boost Converter

This example shows how to estimate a transfer function from frequency response data. You use Simulink® Control Design™ to collect frequency response data from a Simulink model and the `tfest` command to estimate a transfer function from the measured data. To run the example with previously saved frequency response data start from the **Estimating a Transfer Function** section.

Boost Converter

Open the Simulink model.

```
mdl = 'iddemo_boost_converter';
open_system(mdl);
```
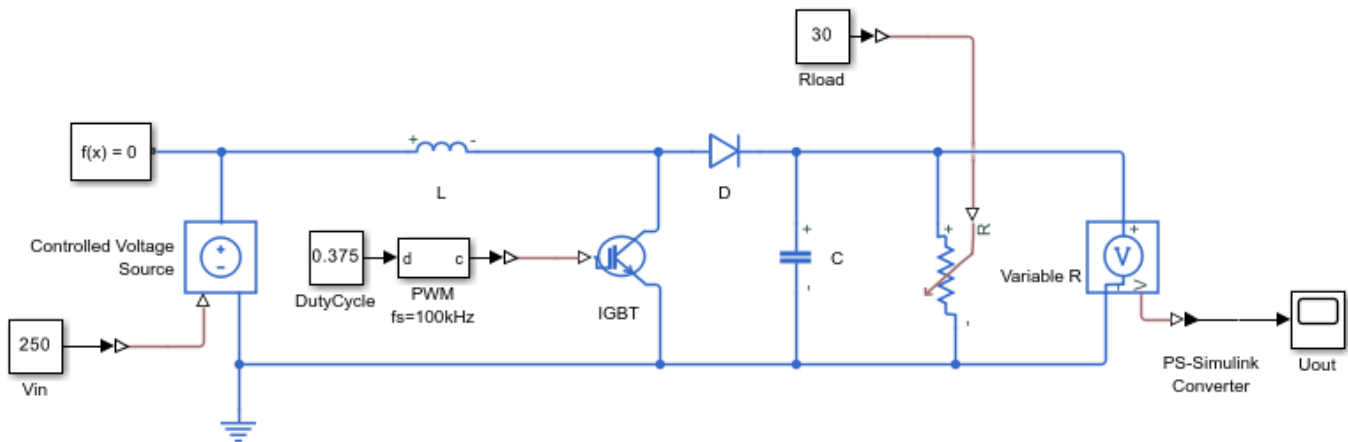

The model is of a Boost Converter circuit that converts a DC voltage to another DC voltage (typically a higher voltage) by controlled chopping or switching of the source voltage. In this model an IGBT driven by a PWM signal is used for switching.

For this example we are interested in the transfer function from the PWM duty cycle set-point to the load voltage, Uout.

Collect Frequency Response Data

We use the `frestimate` command to perturb the duty cycle set-point with sinusoids of different frequencies and log the resulting load voltage. From these we find how the system modifies the magnitude and phase of the injected sinusoids giving us discrete points on the frequency response.

Using `frestimate` requires two preliminary steps

- Specifying the frequency response input and output points

- Defining the sinusoids to inject at the input point

The frequency response input and output points are created using the `linio` command and for this example are the outputs of the `DutyCycle` and `Voltage Measurement` blocks.

```
ios = [...
    linio([mdl,'/DutyCycle'],1,'input'); ...
    linio([mdl,'/PS-Simulink Converter'],1,'output')];
```

We use the `frest.Sinestream` command to define the sinusoids to inject at the input point. We are interested in the frequency range 200 to 20k rad/s, and want to perturb the duty cycle by 0.03.

```
f = logspace(log10(200),log10(20000),10);
in = frest.Sinestream('Frequency',f,'Amplitude',0.03);
```

The simulation time required to simulate the model with this sine-stream signal is determined using the `getSimulationTime` command and as we know the simulation end time used by the model we can get an idea of how much longer the sine-stream simulation will take over simply running the model.

```
getSimulationTime(in)/0.02
```

```
ans =

   15.5933
```

We use the defined inputs and sine-stream with `frestimate` to compute discrete points on the frequency response.

```
[sysData,simlog] = frestimate(mdl,ios,in);
bopt                = bodeoptions;
bopt.Grid           = 'on';
bopt.PhaseMatching = 'on';
figure, bode(sysData,'*r',bopt)
```

Bode Diagram
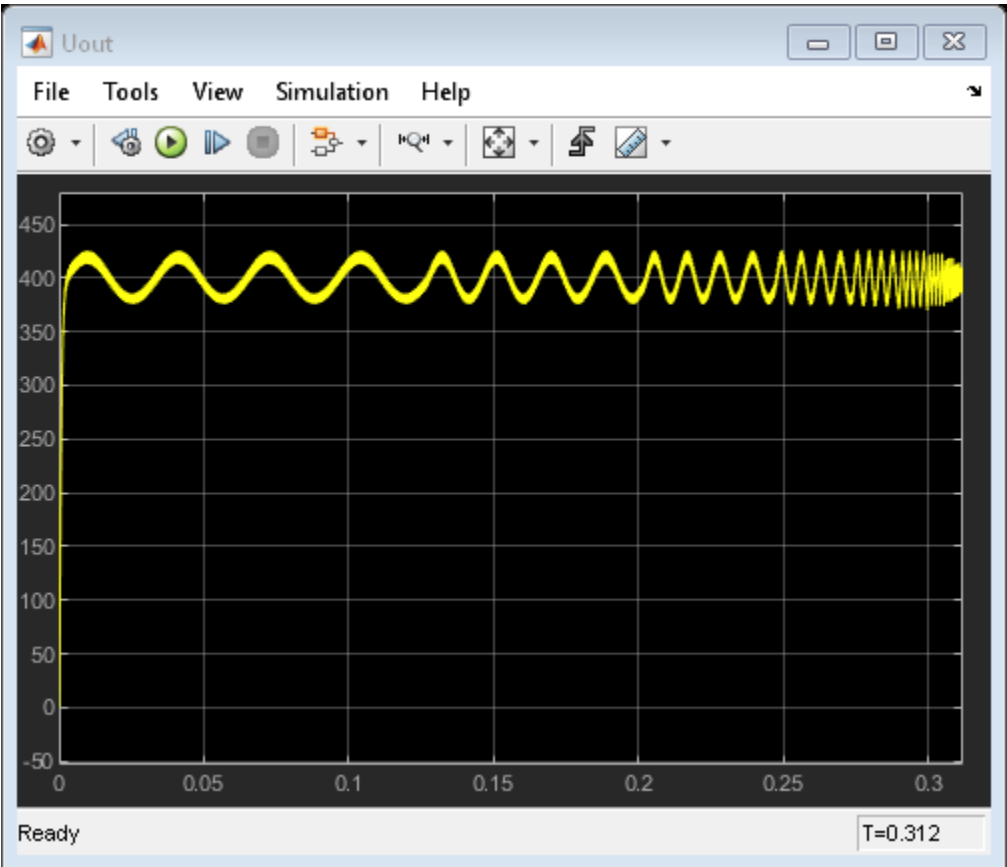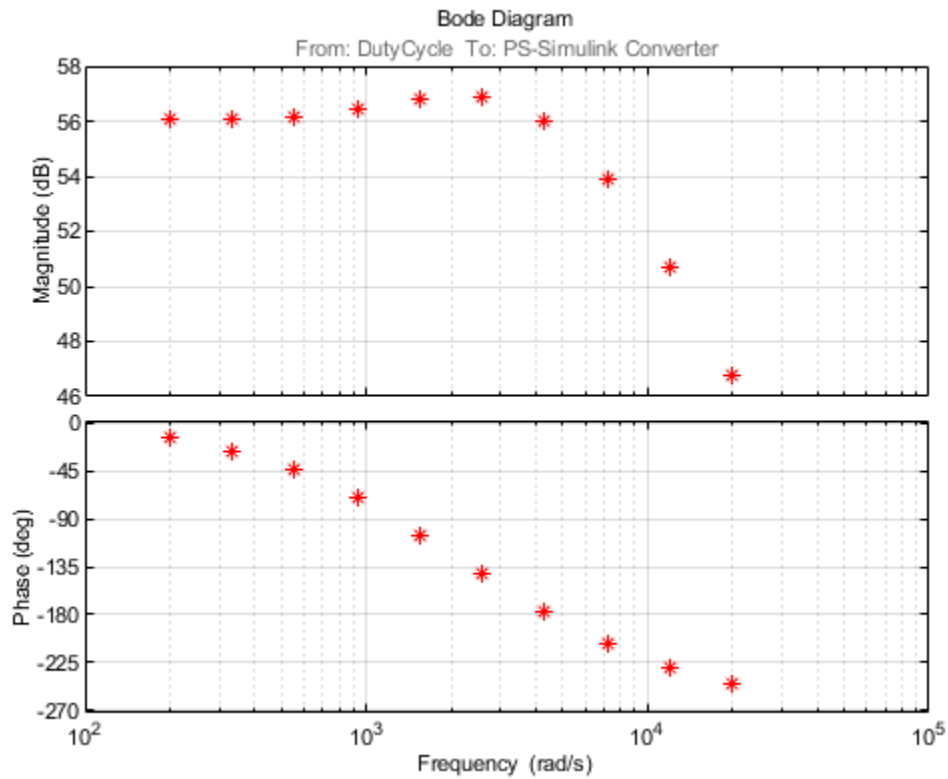
The Bode response shows a system with a 56db gain, some minor resonance around 2500 rad/s and a high frequency roll off of around 20 db/decade matching what we would expect for this circuit.

The `frest.simView` command allows us to inspect the frestimate process showing the injected signal, measured output, and frequency response in one graphical interface.

```
frest.simView(simlog,in,sysData);
```
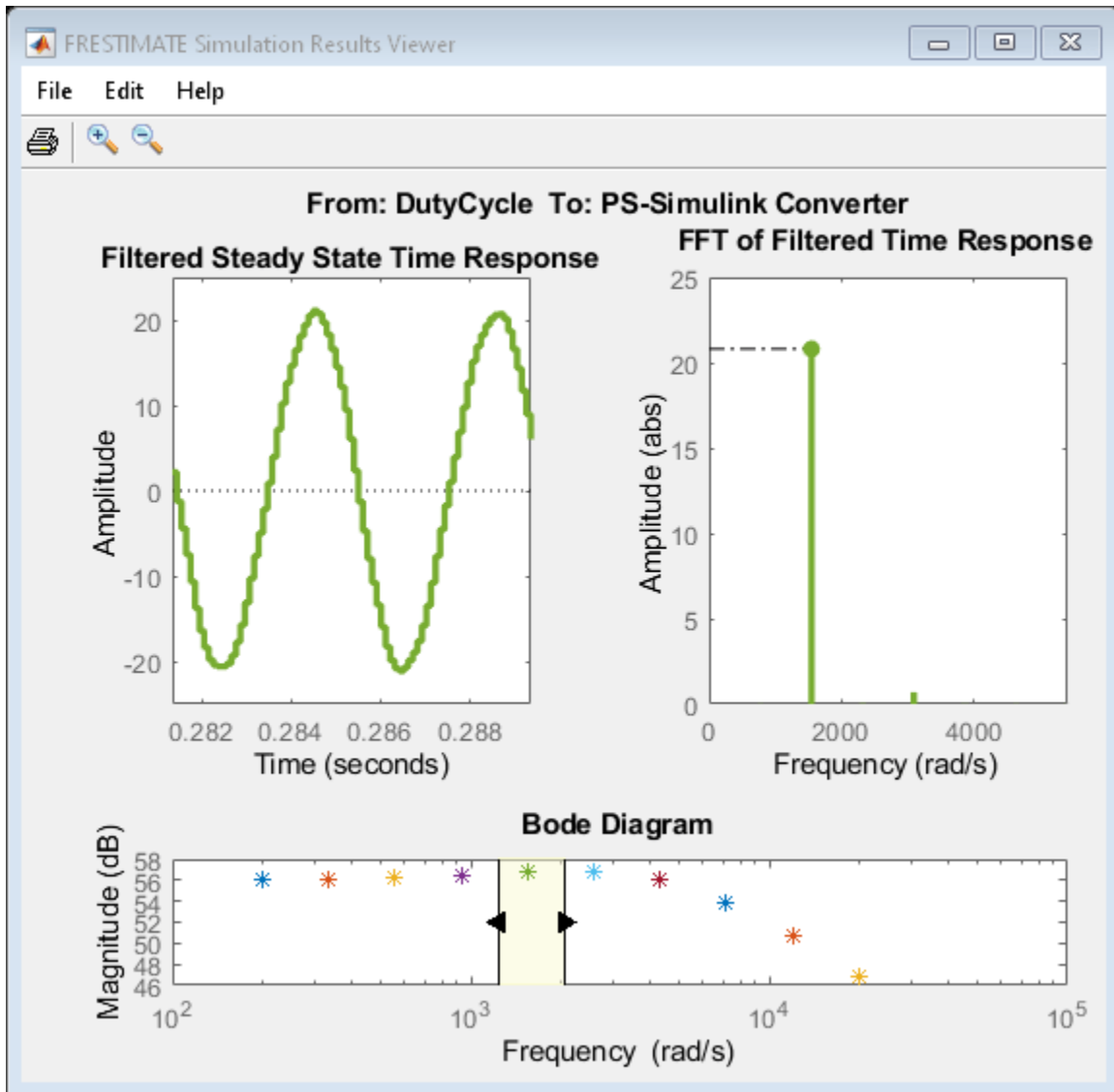
The figure shows the model response to injected sinusoids and the FFT of the model response. Notice that the injected sinusoids result in signals with a dominant frequency and limited harmonics indicating a linear model and successful frequency response data collection.

Estimating a Transfer Function

In the previous step we collected frequency response data. This data describes the system as discrete frequency points and we now fit a transfer function to the data.

We generated the frequency response data using Simulink Control Design, if Simulink Control Design is not installed use the following command to load the saved frequency response data

```
load iddemo_boostconverter_data
```

Inspecting the frequency response data we expect that the system can be described by a second order system.

```
sysA = tfest(sysData,2)
figure, bode(sysData,'r*',sysA,bopt)


sysA =
  From input "DutyCycle" to output "PS-Simulink Converter":
  -4.456e06 s + 6.175e09
  ----------------------
  s^2 + 6995 s + 9.834e06

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 response data "sysData".
Fit to estimation data: 98.04%
FPE: 281.4, MSE: 120.6
```
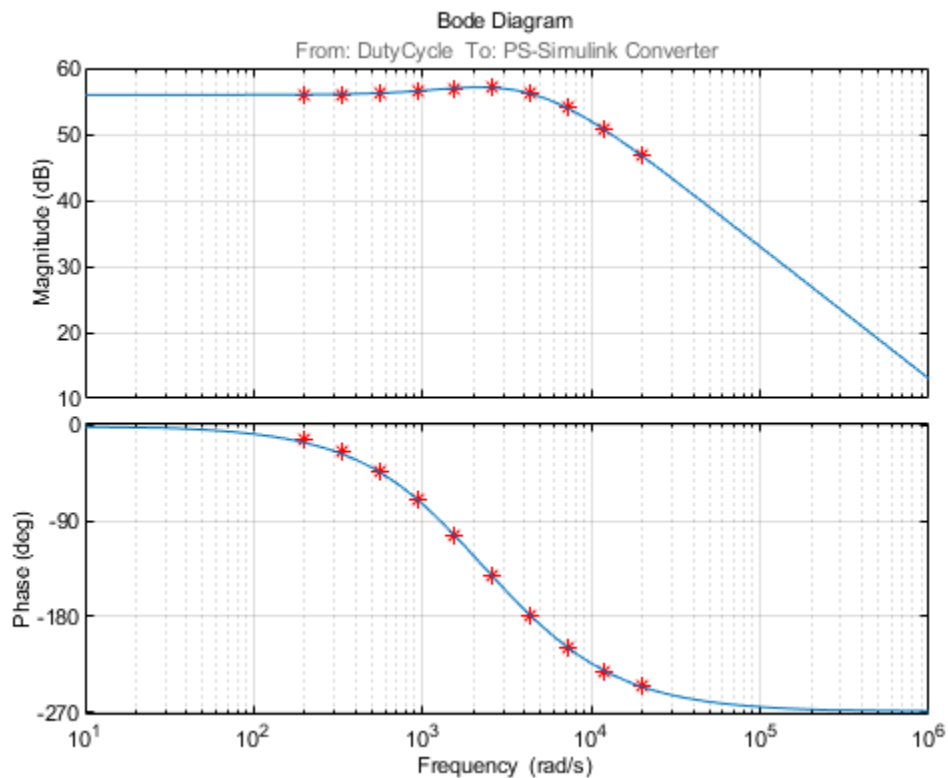

The estimated transfer function is accurate over the provided frequency range.

```
bdclose(mdl)
```

9

Identifying Frequency-Response Models

What is a Frequency-Response Model?

A *frequency-response model* is the frequency response of a linear system evaluated over a range of frequency values. The model is represented by an `idfrd` model object that stores the frequency response, sample time, and input-output channel information.

The frequency-response function describes the steady-state response of a system to sinusoidal inputs. For a linear system, a sinusoidal input of a specific frequency results in an output that is also a sinusoid with the same frequency, but with a different amplitude and phase. The frequency-response function describes the amplitude change and phase shift as a function of frequency.

You can estimate frequency-response models and visualize the responses on a Bode plot, which shows the amplitude change and the phase shift as a function of the sinusoid frequency.

For a discrete-time system sampled with a time interval T, the transfer function $G(z)$ relates the Z-transforms of the input $U(z)$ and output $Y(z)$:

$$Y(z) = G(z)U(z) + H(z)E(z)$$

The frequency-response is the value of the transfer function, $G(z)$, evaluated on the unit circle ($z = \exp^{iwT}$) for a vector of frequencies, w. $H(z)$ represents the noise transfer function, and $E(z)$ is the Z-transform of the additive disturbance $e(t)$ with variance λ. The values of G are stored in the `ResponseData` property of the `idfrd` object. The noise spectrum is stored in the `SpectrumData` property.

Where, the noise spectrum is defined as:

$$\Phi_v(\omega) = \lambda T |H(e^{i\omega T})|^2$$

A MIMO frequency-response model contains frequency-responses corresponding to each input-output pair in the system. For example, for a two-input, two-output model:

$$Y_1(z) = G_{11}(z)U_1(z) + G_{12}(z)U_2(z) + H_1(z)E_1(z)$$
$$Y_2(z) = G_{21}(z)U_1(z) + G_{22}(z)U_2(z) + H_2(z)E_2(z)$$

Where, G_{ij} is the transfer function between the ith output and the jth input. $H_1(z)$ and $H_2(z)$ represent the noise transfer functions for the two outputs. $E_1(z)$ and $E_2(z)$ are the Z-transforms of the additive disturbances, $e_1(t)$ and $e_2(t)$, at the two model outputs, respectively.

Similar expressions apply for continuous-time frequency response. The equations are represented in Laplace domain. For more details, see the `idfrd` reference page.

See Also

Related Examples
- "Estimate Frequency-Response Models in the App" on page 9-4
- "Estimate Frequency-Response Models at the Command Line" on page 9-6

More About
- "Data Supported by Frequency-Response Models" on page 9-3

Data Supported by Frequency-Response Models

You can estimate spectral analysis models from data with the following characteristics:

- Complex or real data.
- Time- or frequency-domain `iddata` or `idfrd` data object. To learn more about estimating time-series models, see "Time Series Analysis".
- Single- or multiple-output data.

See Also

More About

- "What is a Frequency-Response Model?" on page 9-2

Estimate Frequency-Response Models in the App

You must have already imported your data into the app and performed any necessary preprocessing operations. For more information, see "Data Preparation".

To estimate frequency-response models in the System Identification app:

1   In the System Identification app, select **Estimate > Spectral models** to open the Spectral Model dialog box.

2   In the **Method** list, select the spectral analysis method you want to use. For information about each method, see "Selecting the Method for Computing Spectral Models" on page 9-7.

3   Specify the frequencies at which to compute the spectral model in *one* of the following ways:

- In the **Frequencies** field, enter either a vector of values, a MATLAB expression that evaluates to a vector, or a variable name of a vector in the MATLAB workspace. For example, `logspace(-1,2,500)`.

- Use the combination of **Frequency Spacing** and **Frequencies** to construct the frequency vector of values:

    - In the **Frequency Spacing** list, select `Linear` or `Logarithmic` frequency spacing.

      Note For `etfe`, only the `Linear` option is available.

    - In the **Frequencies** field, enter the number of frequency points.

    For time-domain data, the frequency ranges from 0 to the Nyquist frequency. For frequency-domain data, the frequency ranges from the smallest to the largest frequency in the data set.

4   In the **Frequency Resolution** field, enter the frequency resolution, as described in "Controlling Frequency Resolution of Spectral Models" on page 9-8. To use the default value, enter `default` or, equivalently, the empty matrix `[]`.

5   In the **Model Name** field, enter the name of the correlation analysis model. The model name should be unique in the Model Board.

6   Click **Estimate** to add this model to the Model Board in the System Identification app.

7   In the Spectral Model dialog box, click **Close**.

8   To view the frequency-response plot, select the **Frequency resp** check box in the System Identification app. For more information about working with this plot, see "Frequency Response Plots for Model Validation" on page 17-57.

9   To view the estimated disturbance spectrum, select the **Noise spectrum** check box in the System Identification app. For more information about working with this plot, see "Noise Spectrum Plots" on page 17-64.

10   Validate the model after estimating it. For more information, see "Model Validation".

To export the model to the MATLAB workspace, drag it to the **To Workspace** rectangle in the System Identification app. You can retrieve the responses from the resulting `idfrd` model object using the `bode` or `nyquist` command.

See Also

Related Examples

- "Estimate Frequency-Response Models at the Command Line" on page 9-6

More About

- "What is a Frequency-Response Model?" on page 9-2
- "Data Supported by Frequency-Response Models" on page 9-3
- "Selecting the Method for Computing Spectral Models" on page 9-7
- "Controlling Frequency Resolution of Spectral Models" on page 9-8

Estimate Frequency-Response Models at the Command Line

You can use the `etfe`, `spa`, and `spafdr` commands to estimate spectral models. The following table provides a brief description of each command and usage examples.

The resulting models are stored as `idfrd` model objects. For detailed information about the commands and their arguments, see the corresponding reference page.

Commands for Frequency Response

Command	Description	Usage
`etfe`	Estimates an empirical transfer function using Fourier analysis.	To estimate a model m, use the following syntax: `m=etfe(data)`
`spa`	Estimates a frequency response with a fixed frequency resolution using spectral analysis.	To estimate a model m, use the following syntax: `m=spa(data)`
`spafdr`	Estimates a frequency response with a variable frequency resolution using spectral analysis.	To estimate a model m, use the following syntax: `m=spafdr(data,R,w)` where R is the resolution vector and w is the frequency vector.

Validate the model after estimating it. For more information, see "Model Validation".

See Also

Related Examples

- "Estimate Frequency-Response Models in the App" on page 9-4

More About

- "What is a Frequency-Response Model?" on page 9-2
- "Data Supported by Frequency-Response Models" on page 9-3
- "Selecting the Method for Computing Spectral Models" on page 9-7
- "Controlling Frequency Resolution of Spectral Models" on page 9-8

Selecting the Method for Computing Spectral Models

This section describes how to select the method for computing spectral models in the estimation procedures "Estimate Frequency-Response Models in the App" on page 9-4 and "Estimate Frequency-Response Models at the Command Line" on page 9-6.

You can choose from the following three spectral-analysis methods:

- `etfe` (**E**mpirical **T**ransfer **F**unction **E**stimate)

  For input-output data — This method computes the ratio of the Fourier transform of the output to the Fourier transform of the input.

  For time-series data — This method computes a periodogram as the normalized absolute squares of the Fourier transform of the time series.

  ETFE works well for highly resonant systems or narrowband systems. The drawback of this method is that it requires linearly spaced frequency values, does not estimate the disturbance spectrum, and does not provide confidence intervals. ETFE also works well for periodic inputs and computes exact estimates at multiples of the fundamental frequency of the input and their ratio.

- `spa` (**SP**ectral **A**nalysis)

  This method is the Blackman-Tukey spectral analysis method, where windowed versions of the covariance functions are Fourier transformed.

- `spafdr` (**SP**ectral **A**nalysis with **F**requency **D**ependent **R**esolution)

  This method is a variant of the Blackman-Tukey spectral analysis method with frequency-dependent resolution. First, the algorithm computes Fourier transforms of the inputs and outputs. Next, the products of the transformed inputs and outputs with the conjugate input transform are smoothed over local frequency regions. The widths of the local frequency regions can vary as a function of frequency. The ratio of these averages computes the frequency-response estimate.

See Also

Related Examples
- "Estimate Frequency-Response Models in the App" on page 9-4
- "Estimate Frequency-Response Models at the Command Line" on page 9-6

Controlling Frequency Resolution of Spectral Models

This section supports the estimation procedures "Estimate Frequency-Response Models in the App" on page 9-4 and "Estimate Frequency-Response Models at the Command Line" on page 9-6.

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 $\frac{2\pi k}{T}\left(\frac{k}{\text{Period}}\right)$ where $k = 1, 2, ..., \text{Period}$.

For periodic data, the frequency resolution is ignored.

See Also
`etfe` | `spa` | `spafdr`

Related Examples
- "Estimate Frequency-Response Models in the App" on page 9-4
- "Estimate Frequency-Response Models at the Command Line" on page 9-6

Spectrum Normalization

The *spectrum* of a signal is the square of the Fourier transform of the signal. The spectral estimate using the commands spa, spafdr, and etfe is normalized by the sample time T:

$$\Phi_y(\omega) = T \sum_{k=-M}^{M} R_y(kT) e^{-iwT} W_M(k)$$

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:

$$\widehat{R}_y(kT) = \frac{1}{N} \sum_{l=1}^{N} y(lT - kT) y(lT)$$

Because there is no scaling in a discrete Fourier transform of a vector, the purpose of T is to relate the discrete transform of a vector to the physically meaningful transform of the measured signal. This normalization sets the units of $\Phi_y(\omega)$ as power per radians per unit time, and makes the frequency units radians per unit time.

The scaling factor of T is necessary to preserve the energy density of the spectrum after interpolation or decimation.

By Parseval's theorem, the average energy of the signal must equal the average energy in the estimated spectrum, as follows:

$$Ey^2(t) = \frac{1}{2\pi} \int_{-\pi/T}^{\pi/T} \Phi_y(\omega) d\omega$$

$$S1 \equiv Ey^2(t)$$

$$S2 \equiv \frac{1}{2\pi} \int_{-\pi/T}^{\pi/T} \Phi_y(\omega) d\omega$$

To compare the left side of the equation (S1) to the right side (S2), enter the following commands. In this code, phiy contains $\Phi_y(\omega)$ between $\omega = 0$ and $\omega = \pi/T$ with the frequency step given as follows:

$$\left(\frac{\pi}{T \cdot \text{length(phiy)}} \right)$$

```
load iddata1
```

Create a time-series iddata object.

```
y = z1(:,1,[]);
```

Define sample interval from the data.

```
T = y.Ts;
```

Estimate the frequency response.

```
sp = spa(y);
```

Remove spurious dimensions

```
phiy = squeeze(sp.spec);
```

Compute average energy of the signal.

```
S1 = sum(y.y.^2)/size(y,1)
```

S1 = 19.4646

Compute average energy from the estimated energy spectrum, where S2 is scaled by T.

```
S2 = sum(phiy)/length(phiy)/T
```

S2 = 19.2076

Thus, the average energy of the signal approximately equals the average energy in the estimated spectrum.

See Also
etfe | spa | spafdr

10

Identifying Impulse-Response Models

What Is Time-Domain Correlation Analysis?

Time-domain correlation analysis refers to non-parametric estimation of the impulse response of dynamic systems as a finite impulse response (FIR) model from the data. The estimated model is stored as transfer function model object (`idtf`). For information about transfer function models, see "What are Transfer Function Models?" on page 8-2.

Correlation analysis assumes a linear system and does not require a specific model structure.

Impulse response is the output signal that results when the input is an impulse and has the following definition for a discrete model:

$$u(t) = 0 \quad t > 0$$
$$u(t) = 1 \quad t = 0$$

The response to an input *u(t)* is equal to the convolution of the impulse response, as follows:

$$y(t) = \int_0^t h(t - z) \cdot u(z)dz$$

See Also

Related Examples
- "Estimate Impulse-Response Models Using System Identification App" on page 10-4
- "Estimate Impulse-Response Models at the Command Line" on page 10-6

More About
- "Data Supported by Correlation Analysis" on page 10-3

Data Supported by Correlation Analysis

You can estimate impulse-response models from data with the following characteristics:

- Real or complex data.
- Single- or multiple-output data.
- Time- or frequency-domain data with nonzero sample time.

  Time-domain data must be regularly sampled. You cannot use time-series data for correlation analysis.

See Also

More About

- "What Is Time-Domain Correlation Analysis?" on page 10-2

Estimate Impulse-Response Models Using System Identification App

Before you can perform this task, you must have:

- Imported data into the System Identification app. See "Import Time-Domain Data into the App" on page 2-13. For supported data formats, see "Data Supported by Correlation Analysis" on page 10-3.
- Performed any required data preprocessing operations. To improve the accuracy of your model, you should detrend your data. See "Ways to Prepare Data for System Identification" on page 2-5.

To estimate in the System Identification app using time-domain correlation analysis:

1   In the System Identification app, select **Estimate > Correlation models** to open the Correlation Model dialog box.

2   In the **Time span (s)** field, specify a scalar value as the time interval over which the impulse or step response is calculated. For a scalar time span *T*, the resulting response is plotted from *-T/4* to *T*.

> **Tip** You can also enter a 2-D vector in the format `[min_value max_value]`.

3   In the **Order of whitening filter** field, specify the filter order.

The prewhitening filter is determined by modeling the input as an autoregressive process of order *N*. The algorithm applies a filter of the form *A(q)u(t)=u_F(t)*. That is, the input *u(t)* is subjected to an FIR filter *A* to produce the filtered signal *u_F(t)*. *Prewhitening* the input by applying a whitening filter before estimation might improve the quality of the estimated impulse response *g*.

The order of the prewhitening filter, *N*, is the order of the *A* filter. *N* equals the number of lags. The default value of *N* is `10`, which you can also specify as `[]`.

4   In the **Model Name** field, enter the name of the correlation analysis model. The name of the model should be unique in the Model Board.

5   Click **Estimate** to add this model to the Model Board in the System Identification app.

6   In the Correlation Model dialog box, click **Close**.

Next Steps

- Export the model to the MATLAB workspace for further analysis by dragging it to the **To Workspace** rectangle in the System Identification app.
- View the transient response plot by selecting the **Transient resp** check box in the System Identification app. For more information about working with this plot and selecting to view impulse- versus step-response, see "Impulse and Step Response Plots" on page 17-51.

See Also

Related Examples

- "Identify Delay Using Transient-Response Plots" on page 10-9

- "Estimate Impulse-Response Models at the Command Line" on page 10-6

More About

- "What Is Time-Domain Correlation Analysis?" on page 10-2
- "Data Supported by Correlation Analysis" on page 10-3
- "Correlation Analysis Algorithm" on page 10-11

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 2-34. For supported data formats, see "Data Supported by Correlation Analysis" on page 10-3.
- Performed any required data preprocessing operations. If you use time-domain data, you can detrend it before estimation. See "Ways to Prepare Data for System Identification" on page 2-5.

Use `impulseest` to compute impulse response models. `impulseest` estimates a high-order, noncausal FIR model using correlation analysis. The resulting models are stored as `idtf` model objects and contain impulse-response coefficients in the model numerator.

To estimate the model m and plot the impulse or step response, use the following syntax:

```
m=impulseest(data,N);
impulse(m,Time);
step(m,Time);
```

where `data` is a single- or multiple-output `iddata` or `idfrd` object. `N` is a scalar value specifying the order of the FIR system corresponding to the time range 0:Ts:(N-1)*Ts, where *Ts* is the data sample time.

You can also specify estimation options, such as regularizing kernel, pre-whitening filter order and data offsets, using `impulseestOptions` and pass them as an input to `impulseest`. For example:

```
opt = impulseestOptions('RegularizationKernel','TC'));
m = impulseest(data,N,opt);
```

To view the confidence region for the estimated response, use `impulseplot` and `stepplot` to create the plot. Then use `showConfidence`.

For example:

```
h = stepplot(m,Time);
showConfidence(h,3) % 3 std confidence region
```

Note `cra` is an alternative method for computing impulse response from time-domain data only.

Next Steps

- Perform model analysis. See "Validating Models After Estimation" on page 17-2.

See Also

Related Examples

- "Identify Delay Using Transient-Response Plots" on page 10-9
- "Compute Response Values" on page 10-8
- "Estimate Impulse-Response Models Using System Identification App" on page 10-4

More About

Compute Response Values

You can use `impulse` and `step` commands with output arguments to get the numerical impulse- and step-response vectors as a function of time, respectively.

To get the numerical response values:

1   Compute the FIR model by using `impulseest`, as described in "Estimate Impulse-Response Models at the Command Line" on page 10-6.

2   Apply the following syntax on the resulting model:

```
% To compute impulse-response data
[y,t,~,ysd] = impulse(model)
% To compute step-response data
[y,t,~,ysd] = step(model)
```

where `y` is the response data, `t` is the time vector, and `ysd` is the standard deviations of the response.

See Also

Related Examples

- "Estimate Impulse-Response Models at the Command Line" on page 10-6

Identify Delay Using Transient-Response Plots

You can use transient-response plots to estimate the input delay, or *dead time*, of linear systems. Input delay represents the time it takes for the output to respond to the input.

In the System Identification app: To view the transient response plot, select the **Transient resp** check box in the System Identification app. For example, the following step response plot shows a time delay of about 0.25 s before the system responds to the input.
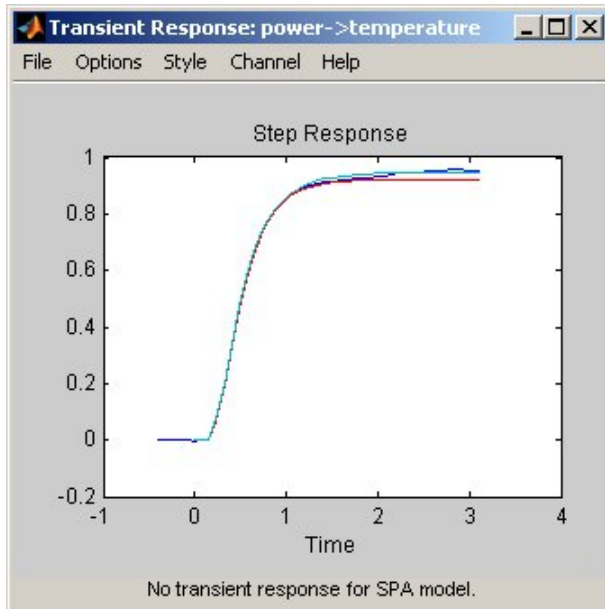
Step Response Plot

At the command line: You can use `impulseplot` to plot the impulse response. The time delay is equal to the first positive peak in the transient response magnitude that is greater than the confidence region for positive time values.

For example, the following commands create an impulse-response plot with a 1-standard-deviation confidence region:

```
load dry2
ze = dry2(1:500);
opt = impulseestOptions('RegularizationKernel','TC');
sys = impulseest(ze,40,opt);
h = impulseplot(sys);
showConfidence(h,1);
```
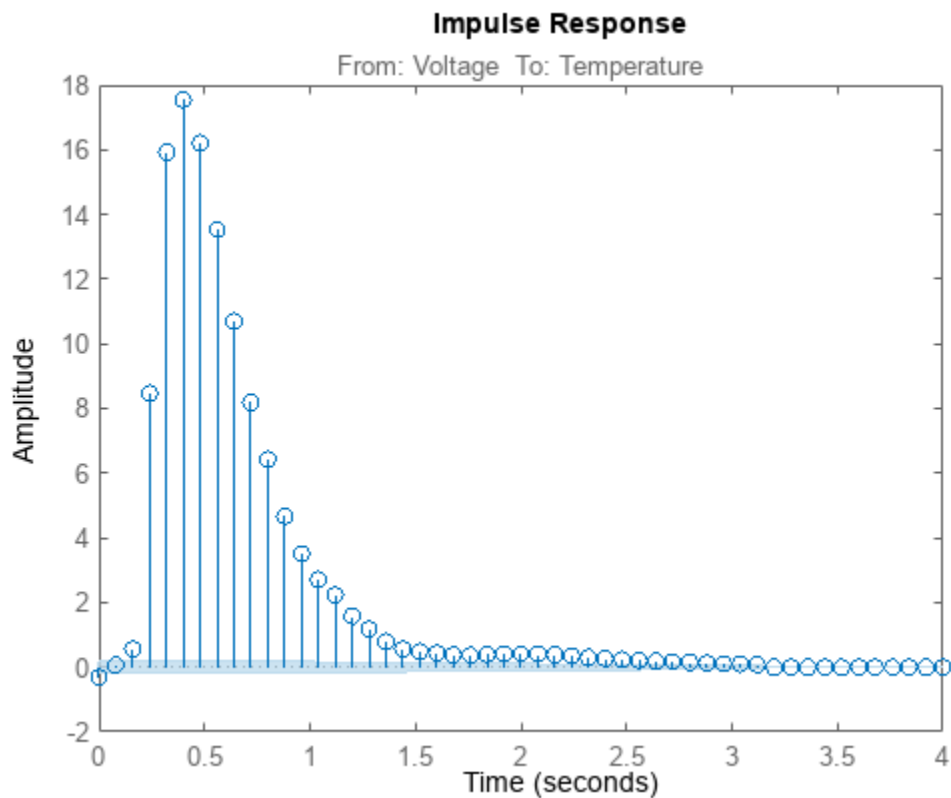
The resulting figure shows that the first positive peak of the response magnitude, which is greater than the confidence region for positive time values, occurs at 0.24 s.

Instead of using `showConfidence`, you can plot the confidence interval interactively, by right-clicking on the plot and selecting **Characteristics** > **Confidence Region**.

See Also

Related Examples

- "Estimate Impulse-Response Models Using System Identification App" on page 10-4
- "Estimate Impulse-Response Models at the Command Line" on page 10-6

Correlation Analysis Algorithm

Correlation analysis refers to methods that estimate the impulse response of a linear model, without specific assumptions about model orders.

The impulse response, *g*, is the system output when the input is an impulse signal. The output response to a general input, *u*(*t*), is 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*.

`impulseest` provides several important options for the estimation:

- Regularization — Regularize the least-squares estimate. With regularization, the algorithm forms an estimate of the prior decay and mutual correlation among *g*(*k*), and then merges this prior estimate with the current information about *g* from the observed data. This approach results in an estimate that has less variance but also some bias. You can choose one of several kernels to encode the prior estimate.

  This option is essential because the model order n can often be quite large. In cases without regularization, n can be automatically decreased to secure a reasonable variance.

  Specify the regularizing kernel using the `RegularizationKernel` name-value argument of `impulseestOptions`.

- Prewhitening — Prewhiten the input by applying an input-whitening filter of order `PW` to the data. Use prewhitening when you are performing unregularized estimation. Using a prewhitening filter minimizes the effect of the neglected tail—k > n—of the impulse response. To achieve prewhitening, the algorithm:

  1 Defines a filter `A` of order `PW` that whitens the input signal `u`:

     `1/A = A(u)e`, where `A` is a polynomial and `e` is white noise.

  2 Filters the inputs and outputs with `A`:

     `uf = Au, yf = Ay`

  3 Uses the filtered signals `uf` and `yf` for estimation.

  Specify prewhitening using the `PW` name-value pair argument of `impulseestOptions`.

- Autoregressive Parameters — Complement the basic underlying FIR model 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 option both gives better results for small n values and allows unbiased estimates when data are generated in closed loop. `impulseest` sets NA to 5 when t > 0 and sets NA to 0 (no autoregressive component) when t < 0.

- Noncausal effects — Include response to negative lags. Use this option if the estimation data includes 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 algorithm handles the existence and character of such feedback h, and estimates h in the same way as g by simply trading places between y and u in the estimation call. Using `impulseest` with an indication of negative delays, `mi = impulseest(data,nk,nb)`, where nk < 0, returns a model `mi` with an impulse response

$$[h(-nk), h(-nk-1), ..., h(0), g(1), g(2), ..., g(nb+nk)]$$

that has an alignment that corresponds to the lags $[nk, nk+1, .., 0, 1, 2, ..., nb+nk]$. The algorithm achieves this alignment because the input delay (`InputDelay`) of model `mi` is `nk`.

For a multi-input multi-output system, the impulse response $g(k)$ is an ny-by-nu matrix, where ny is the number of outputs and nu is the number of inputs. The i–j element of the matrix $g(k)$ describes the behavior of the ith output after an impulse in the jth input.

See Also

Related Examples

- "Estimate Impulse-Response Models Using System Identification App" on page 10-4
- "Estimate Impulse-Response Models at the Command Line" on page 10-6

Nonlinear Black-Box Model Identification

About Identified Nonlinear Models

What Are Nonlinear Models?

Dynamic models in System Identification Toolbox software are mathematical relationships between the inputs $u(t)$ and outputs $y(t)$ of a system. The model is *dynamic* because the output value at the current time depends on the input-output values at previous time instants. Therefore, dynamic models have memory of the past. You can use the input-output relationships to compute the current output from previous inputs and outputs. Dynamic models have states, where a state vector contains the information of the past.

A general form of discrete-time model is:

$y(t) = f(u(t - 1), y(t - 1), u(t - 2), y(t - 2), . . .)$

Such a model is nonlinear if the function f is a nonlinear function. f may represent arbitrary nonlinearities, such as switches and saturations.

The toolbox uses objects to represent various linear and nonlinear model structures. The nonlinear model objects are collectively known as *identified nonlinear models*. These models represent nonlinear systems with coefficients that are identified using measured input-output data. See "Nonlinear Model Structures" on page 11-6 for more information.

When to Fit Nonlinear Models

In practice, all systems are nonlinear and the output is a nonlinear function of the input variables. However, a linear model is often sufficient to accurately describe the system dynamics. In most cases, you should first try to fit linear models.

However, for some scenarios, you might need the additional flexibility of nonlinear models.

Linear Model Is Not Good Enough

You might need nonlinear models when a linear model provides a poor fit to the measured output signals and cannot be improved by changing the model structure or order. Nonlinear models have more flexibility in capturing complex phenomena than the linear models of similar orders.

Physical System Is Weakly Nonlinear

From physical insight or data analysis, you might know that a system is weakly nonlinear. In such cases, you can estimate a linear model and then use this model as an initial model for nonlinear estimation. Nonlinear estimation can improve the fit by using nonlinear components of the model structure to capture the dynamics not explained by the linear model. For more information, see "Initialize Nonlinear ARX Estimation Using Linear Model" on page 11-20 and "Initialize Hammerstein-Wiener Estimation Using Linear Model" on page 12-7.

Physical System Is Inherently Nonlinear

You might have physical insight that your system is nonlinear. Certain phenomena are inherently nonlinear in nature, including dry friction in mechanical systems, actuator power saturation, and sensor nonlinearities in electromechanical systems. You can try modeling such systems using the Hammerstein-Wiener model structure, which lets you interconnect linear models with static nonlinearities. For more information, see "Identifying Hammerstein-Wiener Models" on page 12-5.

Nonlinear models might be necessary to represent systems that operate over a range of operating points. In some cases, you might fit several linear models, where each model is accurate at specific operating conditions. You can also try using the nonlinear ARX model structure with tree partitions to model such systems. For more information, see "Identifying Nonlinear ARX Models" on page 11-17.

If you know the nonlinear equations describing a system, you can represent this system as a nonlinear grey-box model and estimate the coefficients from experimental data. In this case, the coefficients are the parameters of the model. For more information, see "Grey-Box Model Estimation".

Before fitting a nonlinear model, try transforming your input and output variables such that the relationship between the transformed variables becomes linear. For example, you might be dealing with a system that has current and voltage as inputs to an immersion heater, and the temperature of the heated liquid as an output. In this case, the output depends on the inputs via the power of the heater, which is equal to the product of current and voltage. Instead of fitting a nonlinear model to two-input and one-output data, you can create a new input variable by taking the product of current and voltage. You can then fit a linear model to the single-input/single-output data.

Linear and Nonlinear Dynamics Are Captured Separately

You might have multiple data sets that capture the linear and nonlinear dynamics separately. For example, one data set with low amplitude input (excites the linear dynamics only) and another data set with high amplitude input (excites the nonlinear dynamics). In such cases, first estimate a linear model using the first data set. Next, use the model as an initial model to estimate a nonlinear model using the second data set. For more information, see "Initialize Nonlinear ARX Estimation Using Linear Model" on page 11-20 and "Initialize Hammerstein-Wiener Estimation Using Linear Model" on page 12-7.

Nonlinear Model Estimation

Black Box Estimation

In a black-box or "cold start" estimation, you only have to specify the order to configure the structure of the model.

```
sys = estimator(data,orders)
```

where `estimator` is the name of an estimation command to use for the desired model type.

For example, you use `nlarx` to estimate nonlinear ARX models, and `nlhw` for Hammerstein-Wiener models.

The first argument, `data`, is time-domain data represented as an `iddata` object. The second argument, `orders`, represents one or more numbers whose definition depends upon the model type.

- For nonlinear ARX models, `orders` refers to the model orders and delays for defining the regressor configuration. Alternatively, you can explicitly specify linear, polynomial, and customer regressors.
- For Hammerstein-Wiener models, `orders` refers to the model order and delays of the linear subsystem transfer function.

When working in the System Identification app, you specify the orders or regressors in the appropriate edit fields of corresponding model estimation dialog boxes.

Refining Existing Models

You can refine the parameters of a previously estimated nonlinear model using the following command:

```
sys = estimator(data,sys0)
```

This command updates the parameters of an existing model `sys0` to fit the data and returns the results in output model `sys`. For nonlinear systems, *estimator* can be `nlarx`, `nlhw`, or `nlgreyest`.

Initializing Estimations with Known Information About Linear Component

Nonlinear ARX (`idnlarx`) and Hammerstein-Wiener (`idnlhw`) models contain a linear component in their structure. If you have knowledge of the linear dynamics, such as through identification of a linear model using low-amplitude data, you can incorporate it during the estimation of nonlinear models. In particular, you can replace the `orders` input argument with a previously estimated linear model using the following command:

```
sys = estimator(data,LinModel)
```

This command uses the linear model `LinModel` to determine the order of the nonlinear model `sys` as well as initialize the coefficients of its linear component.

Estimation Options

There are many options associated with an estimation algorithm that configures the estimation objective function, initial conditions, and numerical search algorithm, among other things of the model. For every estimation command, *estimator*, there is a corresponding option command named *estimator*`Options`. For example, use `nlarxOptions` to generate the option set for `nlarx`. The options command returns an option set that you then pass as an input argument to the corresponding estimation command.

For example, to estimate a nonlinear ARX model with `simulation` as the focus and `lsqnonlin` as the search method, use `nlarxOptions`.

```
load iddata1 z1
Options = nlarxOptions('Focus','simulation','SearchMethod','lsqnonlin');
sys= nlarx(z1,[2 2 1],Options);
```

Information about the options used to create an estimated model is stored in `sys.Report.OptionsUsed`. For more information, see "Estimation Report" on page 1-21.

See Also

Related Examples

More About

- "Available Nonlinear Models" on page 11-10
- "About Identified Linear Models" on page 1-10

Nonlinear Model Structures

About System Identification Toolbox Model Objects

Objects are instances of model classes. Each *class* is a blueprint that defines the following information about your model:

- How the object stores data
- Which operations you can perform on the object

This toolbox includes nine classes for representing models. For example, `idss` represents linear state-space models and `idnlarx` represents nonlinear ARX models. For a complete list of available model objects, see "Available Linear Models" on page 1-19 and "Available Nonlinear Models" on page 11-10.

Model *properties* define how a model object stores information. Model objects store information about a model, such as the mathematical form of a model, names of input and output channels, units, names and values of estimated parameters, parameter uncertainties, and estimation report. For example, an `idss` model has an `InputName` property for storing one or more input channel names.

The allowed operations on an object are called *methods*. In System Identification Toolbox software, some methods have the same name but apply to multiple model objects. For example, `step` creates a step response plot for all dynamic system objects. However, other methods are unique to a specific model object. For example, `canon` is unique to state-space `idss` models and `linearize` to nonlinear black-box models.

Every class has a special method, called the *constructor*, for creating objects of that class. Using a constructor creates an instance of the corresponding class or *instantiates the object*. The constructor name is the same as the class name. For example, `idss` and `idnlarx` are both the name of the class and the name of the constructor for instantiating the linear state-space models and nonlinear ARX models, respectively.

When to Construct a Model Structure Independently of Estimation

You use model constructors to create a model object at the command line by specifying all required model properties explicitly.

You must construct the model object independently of estimation when you want to:

- Simulate or analyze the effect of model parameters on its response, independent of estimation.
- Specify an initial guess for specific model parameter values before estimation. You can specify bounds on parameter values, or set up the auxiliary model information in advance, or both. Auxiliary model information includes specifying input/output names, units, notes, user data, and so on.

In most cases, you can use the estimation commands to both construct and estimate the model—without having to construct the model object independently. For example, the estimation command `tfest` creates a transfer function model using data and the number of poles and zeros of the model. Similarly, `nlarx` creates a nonlinear ARX model using data and model orders and delays that define the regressor configuration. For information about how to both construct and estimate models with a single command, see "Model Estimation Commands" on page 1-31.

In case of grey-box models, you must always construct the model object first and then estimate the parameters of the ordinary differential or difference equation.

Commands for Constructing Nonlinear Model Structures

The following table summarizes the model constructors available in the System Identification Toolbox product for representing various types of nonlinear models.

After model estimation, you can recognize the corresponding model objects in the MATLAB Workspace browser by their class names. The name of the constructor matches the name of the object it creates.

For information about how to both construct and estimate models with a single command, see "Model Estimation Commands" on page 1-31.

Summary of Model Constructors

Model Constructor	Resulting Model Class
`idnlgrey`	Nonlinear ordinary differential or difference equation (grey-box models). You write a function or MEX-file to represent the governing equations.
`idnlarx`	Nonlinear ARX models, which define the predicted output as a nonlinear function of past inputs and outputs.
`idnlhw`	Nonlinear Hammerstein-Wiener models, which include a linear dynamic system with nonlinear static transformations of inputs and outputs.
`idNeuralStateSpace`	Neural state-space models, which use neural networks to approximate the functions representing a nonlinear state space realization of your system.

For more information about when to use these commands, see "When to Construct a Model Structure Independently of Estimation" on page 11-6.

Model Properties

A model object stores information in the *properties* of the corresponding model class.

The nonlinear models `idnlarx`, `idnlhw`, and `idnlgrey` are based on the `idnlmodel` superclass and inherit all `idnlmodel` properties.

In general, all model objects have properties that belong to the following categories:

- Names of input and output channels, such as `InputName` and `OutputName`
- Sample time of the model, such as `Ts`
- Time units
- Model order and mathematical structure (for example, ODE or nonlinearities)
- Properties that store estimation results (`Report`)
- User comments, such as `Notes` and `Userdata`

For information about getting help on object properties, see the model reference pages.

The following table summarizes the commands for viewing and changing model property values. Property names are not case sensitive. You do not need to type the entire property name if the first few letters uniquely identify the property.

Task	Command	Example

View all model properties and their values	Use `get`.	Load sample data, compute a nonlinear ARX model, and list the model properties.
		
		`load iddata1`
		`sys = nlarx(z1,[4 4 1]);`
		`get(sys)`

Access a specific model property	Use dot notation.	View the output function in the previous model. `sys.OutputFcn`
	For properties, such as `Report`, that are configured like structures, use dot notation of the form `model.PropertyName.FieldName`. `FieldName` is the name of any field of the property.	View the options used in the nonlinear ARX model estimation. `sys.Report.OptionsUsed`
Change model property values	Use dot notation.	Change the nonlinearity mapping function that the output function uses. `sys.OutputFcn = 'idSigmoidNetwork';`
Access model parameter values and uncertainty information	Use `getpvec` and `getcov` (for `idnlgrey` models only).	Model parameters and associated uncertainty data. `getpvec(sys)`
Set model parameter values and uncertainty information	Use `setpar` and `setcov` (for `idnlgrey` models only).	Set the parameter vector. `sys = setpar(sys,'Value',parlist)`
Get number of parameters	Use `nparams`.	Get the number of parameters. `nparams(sys)`

See Also

Related Examples

- "Identifying Nonlinear ARX Models" on page 11-17
- "Identifying Hammerstein-Wiener Models" on page 12-5
- "Represent Nonlinear Dynamics Using MATLAB File for Grey-Box Estimation" on page 13-78

More About

- "About Identified Nonlinear Models" on page 11-2

Available Nonlinear Models

Overview

The System Identification Toolbox software provides three types of nonlinear model structures:

- "Nonlinear ARX Models" on page 11-10
- "Hammerstein-Wiener Models" on page 11-10
- "Nonlinear Grey-Box Models" on page 11-11
- "Neural State-Space Models" on page 11-11

The toolbox refers to Nonlinear ARX and Hammerstein-Wiener collectively as "nonlinear black box" models. You can configure these models in a variety of ways to represent various behavior using nonlinear functions such as wavelet networks, tree partitions, piece-wise linear functions, polynomials, saturation and dead zones.

The nonlinear grey-box models let you estimate coefficients of nonlinear differential equations.

Neural state space models let you estimate, using neural networks, the functions tat define the nonlinear state space realization of your system.

Nonlinear ARX Models

Nonlinear ARX models extend the linear ARX models on page 6-2 to the nonlinear case and have this structure:

$y(t) = f(y(t - 1), ..., y(t - na), u(t - nk), ..., u(t -nk -nb + 1))$

where the function f depends on a finite number of previous inputs u and outputs y. na is the number of past output terms and nb is the number of past input terms used to predict the current output. nk is the delay from the input to the output, specified as the number of samples.

Use this model to represent nonlinear extensions of linear models. This structure allows you to model complex nonlinear behavior using output functions that combine linear and nonlinear components, such as wavelet and sigmoid networks. Typically, you use nonlinear ARX models as black-box structures. The output function of the nonlinear ARX model is a flexible mapping function with parameters that need not have physical significance.

System Identification Toolbox software uses `idnlarx` objects to represent nonlinear ARX models. For more information about estimation, see "Nonlinear ARX Models".

Hammerstein-Wiener Models

Hammerstein-Wiener models describe dynamic systems using one or two static nonlinear blocks in series with a linear block. The linear block is a discrete transfer function and represents the dynamic component of the model.

You can use the Hammerstein-Wiener structure to capture physical nonlinear effects in sensors and actuators that affect the input and output of a linear system, such as dead zones and saturation. Alternatively, use Hammerstein-Wiener structures as black box structures that do not represent physical insight into system processes.

System Identification Toolbox software uses `idnlhw` objects to represent Hammerstein-Wiener models. For more information about estimation, see "Hammerstein-Wiener Models".

Nonlinear Grey-Box Models

Nonlinear state-space models have this representation:

$$\dot{x}(t) = F(x(t), u(t))$$
$$y(t) = H(x(t), u(t))$$

where F and H can have any parameterization. A nonlinear ordinary differential equation of high order can be represented as a set of first order equations. You use the `idnlgrey` object to specify the structures of such models based on physical insight about your system. The parameters of such models typically have physical interpretations. Use this model to represent nonlinear ODEs with unknown parameters.

For more information about estimating nonlinear state-space models, see "Grey-Box Model Estimation".

Neural State-Space Models

Neural state-space models are a type of nonlinear state-space models, and therefore have this representation:

$$\dot{x}(t) = F(x(t), u(t))$$
$$y(t) = H(x(t), u(t))$$

where F and H are implemented using neural networks.

To identify a system from experiment data, first, create a neural state-space object with a given number of states, inputs and outputs using `idNeuralStateSpace`. Next, create neural networks to approximate the state and output functions using `createMLPNetwork`. Then, to train the networks, use `nlssest`, supplying time-domain experiment data, the neural state-space object, and training options as input arguments.

After the training is completed, you can validate the resulting state-space system by simulating it against real data, and then use it to generate deployment code or design a control system. For example, to facilitate "Nonlinear MPC" (Model Predictive Control Toolbox) design, generate the state transition and Jacobian functions using `generateMATLABFunction`.

See Also

Related Examples

- "Identifying Nonlinear ARX Models" on page 11-17
- "Identifying Hammerstein-Wiener Models" on page 12-5
- "Represent Nonlinear Dynamics Using MATLAB File for Grey-Box Estimation" on page 13-78

More About

Preparing Data for Nonlinear Identification

Estimating nonlinear ARX and Hammerstein-Wiener models requires uniformly sampled time-domain data. Your data can have one or more input and output channels.

For time-series data, you can only fit nonlinear ARX models and nonlinear state-space models on page 13-25.

Tip Whenever possible, use different data sets for model estimation and validation.

Before estimating models, import your data into the MATLAB workspace and do *one* of the following:

- **In the System Identification app.** Import data into the app, as described in "Represent Data".
- **At the command line.** Represent your data as an `iddata` object, as described in the corresponding reference page.

You can analyze data quality and preprocess data by interpolating missing values, filtering to emphasize a specific frequency range, or resampling using a different sample time (see "Ways to Prepare Data for System Identification" on page 2-5).

Data detrending can be useful in certain cases, such as before modeling the relationship between the change in input and the change in output about an operating point. However, most applications do not require you to remove offsets and linear trends from the data before nonlinear modeling.

See Also

Related Examples

- "Identifying Nonlinear ARX Models" on page 11-17
- "Identifying Hammerstein-Wiener Models" on page 12-5
- "Represent Nonlinear Dynamics Using MATLAB File for Grey-Box Estimation" on page 13-78

More About

- "About Identified Nonlinear Models" on page 11-2
- "Available Nonlinear Models" on page 11-10

What are Nonlinear ARX Models?

Nonlinear ARX models extend the linear ARX models on page 6-2 to the nonlinear case. The structure of these models enables you to model complex nonlinear behavior using flexible nonlinear functions, such as wavelet and sigmoid networks. For information about when to fit nonlinear models, see "About Identified Nonlinear Models" on page 11-2.

Nonlinear ARX Model Extends the Linear ARX Structure

A linear SISO ARX model on page 6-2 has the following structure:

$$y(t) + a_1 y(t-1) + a_2 y(t-2) + \ldots + a_{na} y(t-na) = b_1 u(t) + b_2 u(t-1) + \ldots + b_{nb} u(t-nb+1) + e(t)$$

Where, u,y, and e are the input, output, and noise. This structure implies that the current output $y(t)$ is predicted as a weighted sum of past output values and current and past input values. na is the number of past output terms, and nb is the number of past input terms used to predict the current output. The input delay nk is set to zero to simplify the notation. Rewriting the equation as a product gives:

$$y_p(t) = [-a_1, \ -a_2, \ldots, \ -a_{na}, b_1, b_2, \ldots, b_{nb}]$$
$$* [y(t-1), y(t-2), \ldots, y(t-na), u(t), u(t-1), \ldots, u(t-nb-1)]^T$$

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 coefficients vector $[-a_1, \ldots, b_{nb}]$ represents the weighting applied to these regressors. The linear ARX model thus predicts the current output y_p as a weighted sum of its regressors.

The structure of a nonlinear ARX model allows the following additional flexibility:

*   Instead of the weighted sum of the regressors that represents a linear mapping, the nonlinear ARX model has a more flexible nonlinear mapping function, F.

    $$y_p(t) = F(y(t-1), y(t-2), y(t-3), \ldots, u(t), u(t-1), u(t-2), \ldots)$$

    Inputs to F are model regressors. When you specify the nonlinear ARX model structure, you can choose one of several available nonlinear functions. For example, F can represent a weighted sum of wavelets that operate on the distance of the regressors from their means. For more information, see "Available Mapping Functions for Nonlinear ARX Models" on page 11-22.

*   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 $y(t-1)^2$, $u(t-1)*y(t-2)$, abs($u(t-1)$), and max($u(t-1)*y(t-3)$,-10).

Structure of Nonlinear ARX Models

A nonlinear ARX model consists of model regressors and an output function. The output function contains one or more mapping objects, one for each model output. Each mapping object can include a linear and a nonlinear function that act on the model regressors to give the model output and a fixed offset for that output. This block diagram represents the structure of a single-output nonlinear ARX model in a simulation scenario.
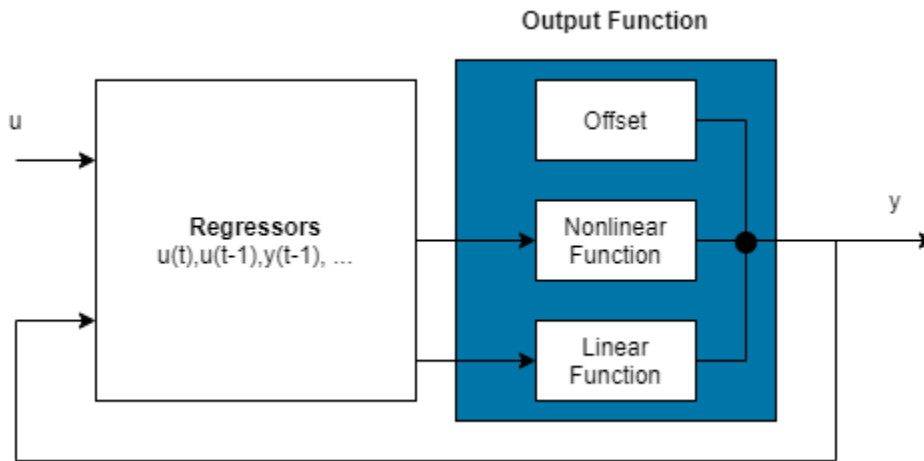
The software computes the nonlinear ARX model output y in two stages:

1   It computes regressor values from the current and past input values and the past output data.

In the simplest case, regressors are delayed inputs and outputs, such as $u(t–1)$ and $y(t–3)$. These kind of regressors are called linear regressors. You specify linear regressors using the `linearRegressor` object. You can also specify linear regressors by using linear ARX model orders as an input argument. For more information, see "Nonlinear ARX Model Orders and Delay" on page 11-16. However, this second approach constrains your regressor set to linear regressors with consecutive delays. To create polynomial regressors, use the `polynomialRegressor` object. To create periodic regressors that contain the sine and cosine functions of delayed input and output variables , use the `periodicRegressor` object. You can also specify custom regressors, which are nonlinear functions of delayed inputs and outputs. For example, $u(t–1)y(t–3)$ is a custom regressor that multiplies instances of input and output together. Specify custom regressors using the `customRegressor` object.

You can assign any of the regressors as inputs to the linear function block of the output function, the nonlinear function block, or both.

2   It maps the regressors to the model output using an output function block. The output function block can include multiple mapping objectslinear, nonlinear, and offset blocks in parallel. For example, consider the following equation:

$$F(x) = L^T(x - r) + g(Q(x - r)) + d$$

Here, x is a vector of the regressors, and r is the mean of x. $F(x) = L^T(x - r) + y_0$ is the output of the linear function block. $g(Q(x - r)) + y_0$ represents the output of the nonlinear function block. Q is a projection matrix that makes the calculations well-conditioned. d is a scalar offset that is added to the combined outputs of the linear and nonlinear blocks. The exact form of $F(x)$ depends on your choice of output function. You can select from the available mapping objects on page 11-22, such as tree-partition networks, wavelet networks, and multilayer neural networks. You can also exclude either the linear or the nonlinear function block from the output function.

When estimating a nonlinear ARX model, the software computes the model parameter values, such as L, r, d, Q, and other parameters specifying g.

The resulting nonlinear ARX models are `idnlarx` objects that store all model data, including model regressors and parameters of the output function. For more information about these objects, see "Nonlinear Model Structures" on page 11-6.

Typically, you use nonlinear ARX models as black-box structures. The nonlinear function of the nonlinear ARX model is a flexible nonlinearity estimator with parameters that need not have physical significance. You can estimate nonlinear ARX in the **System Identification** app or at the command line using the `nlarx` command. You can use uniformly sampled time-domain input-output data or time-series data (no inputs) for estimating nonlinear ARX models. Your data can have one or more input and output channels. You cannot use frequency-domain data for estimation.

Nonlinear ARX Model Orders and Delay

You can use the orders and delays of a nonlinear ARX model to define the linear regressors of the model. The orders and delay are defined as follows:

- *na* — Number of past output terms used to predict the current output.
- *nb* — Number of past input terms used to predict the current output.
- *nk* — Delay from input to the output in terms of the number of samples.

The meaning of *na*, *nb*, and *nk* is similar to that for linear ARX model parameters. Orders are specified as scalars for SISO data, and as *ny*-by-*nu* matrices for MIMO data, where *ny* and *nu* are the number of outputs and inputs. If you are not sure what values to use for the orders and delays, you can estimate them as described in "Preliminary Step – Estimating Model Orders and Input Delays" on page 6-8. 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 linear regressors using the model orders and delays. For example, suppose that you specify `na = 2`, `nb = 3`, and `nk = 5` for a SISO model with input *u* and output *y*. The toolbox computes linear regressors *y*(*t*-2), *y*(*t*-1),*u*(*t*-5), *u*(*t*-6), and *u*(*t*-7).

Rather than use ARX model order to specify regressor delays, you can also specify regressors directly to capture more complex behavior. When you specify linear regressors directly, you can include nonconsecutive delay terms. You can also specify polynomial regressors and custom regressors. For more information, see "Estimate Nonlinear ARX Models in the App" on page 11-24 and "Estimate Nonlinear ARX Models at the Command Line" on page 11-27.

See Also
`idnlarx` | `nlarx`

More About

- "About Identified Nonlinear Models" on page 11-2
- "Identifying Nonlinear ARX Models" on page 11-17
- "Available Mapping Functions for Nonlinear ARX Models" on page 11-22
- "Using Nonlinear ARX Models" on page 11-42
- "How the Software Computes Nonlinear ARX Model Output" on page 11-43

Identifying Nonlinear ARX Models

Nonlinear ARX models extend the linear ARX model to the nonlinear case. For information about the structure of nonlinear ARX models, see "What are Nonlinear ARX Models?" on page 11-14

You can estimate nonlinear ARX models in the **System Identification** app or at the command line using the `nlarx` command. To estimate a nonlinear ARX model, you first prepare the estimation data. You then configure the model structure and estimation algorithm, and then perform the estimation. After estimation, you can validate the estimated model as described in "Validate Nonlinear ARX Models" on page 11-38.

Prepare Data for Identification

You can use uniformly sampled time-domain input-output data or time-series data (no inputs) for estimating nonlinear ARX models. Your data can have zero or more input channels and one or more output channels. You cannot use frequency-domain data for estimation.

To prepare the data for model estimation, import your data into the MATLAB workspace, and do *one* of the following:

- **In the System Identification app** — Import data into the app, as described in "Represent Data".
- **At the command line** — Represent your data as an `iddata` object.

After importing the data, you can analyze data quality and preprocess data by interpolating missing values, filtering to emphasize a specific frequency range, or resampling using a different sample time. For more information, see "Ways to Prepare Data for System Identification" on page 2-5. For most applications, you do not need to remove offsets and linear trends from the data before nonlinear modeling. However, data detrending can be useful in some cases, such as before modeling the relationship between the change in input and output about an operating point.

After preparing your estimation data, you can configure your model structure, loss function, and estimation algorithm, and then estimate the model using the estimation data.

Configure Nonlinear ARX Model Structure

A nonlinear ARX model consists of a set of regressors that can be any combination of linear, polynomial, and custom regressors, and an output function that typically contains a nonlinear and a linear component, as well as a static offset. The block diagram represents the structure of a nonlinear ARX model on page 11-14 in a simulation scenario.

Output Function
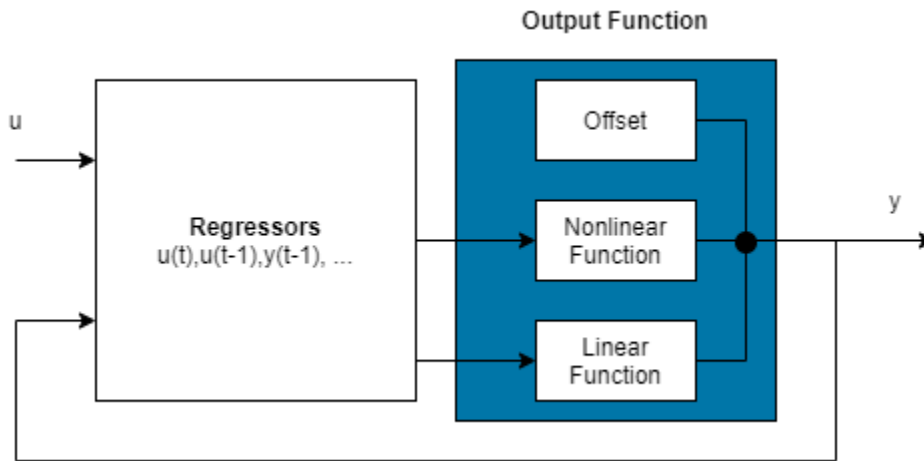
To configure the structure of a nonlinear ARX model:

1 Configure the model regressors.

Choose the linear, polynomial, and customer regressors based on your knowledge of the physical system you are trying to model.

a Specify linear regressors in *one* of the following ways:

- Specify model orders and delay to create the set of linear regressors. For more information, see "Nonlinear ARX Model Orders and Delay" on page 11-16.
- Directly specify the linear regressors by constructing and using `linearRegressor` objects.
- Initialize the model using a linear ARX model. You can perform this operation at the command line only. The initialization configures the nonlinear ARX model to use the linear regressors of the linear model. For more information, see "Initialize Nonlinear ARX Estimation Using Linear Model" on page 11-20.

b Specify polynomial regressors and custom regressors. Polynomial regressors are polynomials that are composed of delayed input and output variables. Custom regressors are arbitrary functions of past inputs and outputs, such as products, powers, and other MATLAB expressions of input and output variables. Specify polynomial and custom regressors in addition to, or instead of, linear regressors for greater flexibility in modeling your data. For more information, see `polynomialRegressor` and `customRegressor`.

c Assign regressors as inputs to the linear and nonlinear components of the output function block. Any regressor can be assigned to either or both of these components. Limiting the number of regressors that are input to the nonlinear component can help reduce model complexity and keep the estimation well conditioned.

The choice of regressors to use for each component can require multiple trials. You can examine a nonlinear ARX plot on page 11-39 to help you gain insight into which regressors have the strongest effect on the model output. Understanding the relative importance of the regressors on the output can then help you decide how to assign the regressors.

2 Configure the output function block.

Specify and configure the output function $F(x)$.

$$F(x) = L^T(x - r) + g(Q(x - r)) + d$$

Here, x is a vector of the regressors, and r is the mean of the regressors x. $F(x) = L^T(x - r) + y_0$ is the output of the linear function block. $g(Q(x - r)) + y_0$ represents the output of the nonlinear function block. Q is a projection matrix that makes the calculations well conditioned. d is a scalar offset that is added to the combined outputs of the linear and nonlinear blocks. The exact form of $F(x)$ depends on your choice of output function. You can select from available nonlinearity estimators on page 11-22, such as tree-partition networks, wavelet networks, and multilayer neural networks. You can also exclude either the linear or the nonlinear function block from the output function.

You can also perform one of the following tasks:

a   Exclude the nonlinear function from the output function such that $F(x) =$
$F(x) = L^T(x - r) + y_0$.

b   Exclude the linear function from the output function such that $F(x) = g(Q(x - r)) + y_0$.

Note You cannot exclude the linear function from tree partitions and neural networks.

For information about how to configure the model structure at the command line and in the app, see "Estimate Nonlinear ARX Models at the Command Line" on page 11-27 and "Estimate Nonlinear ARX Models in the App" on page 11-24.

Specify Estimation Options for Nonlinear ARX Models

To configure the model estimation, specify the loss function to be minimized, and choose the estimation algorithm and other estimation options to perform the minimization.

Configure Loss Function

The loss function or cost function is a function of the error between the model output and the measured output. For more information about loss functions, see "Loss Function and Model Quality Metrics" on page 1-46.

At the command line, use the `nlarx` option set, `nlarxOptions` to configure your loss function. You can specify the following options:

- `Focus` — Specifies whether the simulation or prediction error is minimized during parameter estimation. By default, the software minimizes one-step prediction errors, which correspond to a `Focus` value of `'prediction'`. If you want a model that is optimized for reproducing simulation behavior, specify `Focus` as `'simulation'`. Minimization of simulation error requires differentiable nonlinear functions and takes more time than one-step-ahead prediction error minimization. Thus, you cannot use `idTreePartition` and `idFeedforwardNetwork` nonlinearities when minimizing the simulation error because these nonlinearity estimators are not differentiable.
- `OutputWeight` — Specifies a weighting of the error in multi-output estimations.
- `Regularization` — Modifies the loss function to add a penalty on the variance of the estimated parameters. For more information, see "Regularized Estimates of Model Parameters" on page 1-34.

Specify Estimation Algorithm

To estimate a nonlinear ARX model, the software uses iterative search algorithms to minimize the error between the simulated or predicted model output and the measured output. At the command

line, use `nlarxOptions` to specify the search algorithm and other estimation options. Some of the options you can specify are:

- `SearchMethod` — Search method for minimization of prediction or simulation errors, such as Gauss-Newton and Levenberg-Marquardt line search, and Trust-Region-Reflective Newton approach.
- `SearchOptions` — Option set for the search algorithm, with fields that depend on the value of `SearchMethod`, such as:
  - `MaxIterations` — Maximum number of iterations.
  - `Tolerance` — Condition for terminating iterative search when the expected improvement of the parameter values is less than a specified value.

To see a complete list of available estimation options, see `nlarxOptions`. For details about how to specify these estimation options in the app, see "Estimate Nonlinear ARX Models in the App" on page 11-24.

After preprocessing the estimation data and configuring the model structure and estimation options, you can estimate the model in the **System Identification** app, or using `nlarx` at the command line. The resulting model is an `idnlarx` object that stores all model data, including model regressors and parameters of the nonlinearity estimator. For more information about these model objects, see "Nonlinear Model Structures" on page 11-6. You can validate the estimated model as described in "Validate Nonlinear ARX Models" on page 11-38.

Initialize Nonlinear ARX Estimation Using Linear Model

At the command line, you can use an ARX structure polynomial model (`idpoly` with only *A* and *B* as active polynomials) for nonlinear ARX estimation. To learn more about when to use linear models, see "When to Fit Nonlinear Models" on page 11-2.

Typically, you create a linear ARX model using the `arx` command. You can provide the linear model when constructing or estimating a nonlinear ARX model. For example, use the following syntax to estimate a nonlinear ARX model using estimation data and a linear ARX model `LinARXModel`.

```
m = nlarx(data,LinARXModel)
```

Here `m` is an `idnlarx` object, and `data` is a time-domain `iddata` object. The software uses the linear model for initializing the nonlinear ARX estimation by:

- Assigning the linear ARX model orders and delays as initial values of the nonlinear ARX model orders (`na` and `nb` properties of the `idnlarx` object) and delays (`nk` property). The software uses these orders and delays to compute linear regressors in the nonlinear ARX model structure on page 11-14.
- Using the *A* and *B* polynomials of the linear model to compute the linear function of the nonlinearity estimators (`LinearCoef` parameter of the nonlinearity estimator object), except if the nonlinearity estimator is a neural network.

During estimation, the estimation algorithm uses these values to adjust the nonlinear model to the data.

Note When you use the same data for estimation, a nonlinear ARX model initialized using a linear ARX model produces a better fit to measured output than the linear ARX model itself.

By default, the nonlinearity estimator is the wavelet network (`idWaveletNetwork` object). You can also specify different input and output nonlinearity estimators. For example, you can specify a sigmoid network nonlinearity estimator.

```
m = nlarx(data,LinARXModel,'idSigmoidNetwork')
```

For an example, see "Estimate Nonlinear ARX Models Initialized Using Linear ARX Models" on page 11-35.

See Also

Functions
idnlarx | nlarx

Apps
System Identification

More About

Available Mapping Functions for Nonlinear ARX Models

A nonlinear ARX model consists of model regressors and an output function. The output function includes one or more mapping objects, one object for each output of the model. Each mapping object maps a set of input signals into a single output signal, and typically includes a nonlinear function, a linear function, and an offset. For more information about nonlinear ARX model structure, see "What are Nonlinear ARX Models?" on page 11-14.

The System Identification Toolbox software provides several mapping objects for nonlinear ARX models. When you estimate nonlinear ARX models in the app or at the command line, the software creates and configures the mapping objects that you specify. You can also create and configure mapping objects independently at the command line and then specify these customized objects when you perform estimation.

Some mapping functions represent the nonlinear function as a summed series of nonlinear units, such as wavelet networks or sigmoid functions. Others use models that draw on machine learning algorithms. One mapping object contains no nonlinear function at all, just a linear function and an offset. The following table lists the available mapping objects. For a detailed description of an object, see the corresponding reference page.

Nonlinearity	Mapping Object	Description	Comments
Wavelet network	`idWaveletNetwork`	Sum of dilated and translated wavelets in a wavelet network.	Default
One layer sigmoid network	`idSigmoidNetwork`	Sum of dilated and translated sigmoid unit functions.	
Tree partition	`idTreePartition`	Piecewise linear function over partitions of the regressor space defined by a binary tree.	Useful for modeling data collected over a range of operating conditions.
Linear	`idLinear`	Contains only a linear component and an offset. This estimator produces a model that is similar to the linear ARX model, but offers the additional flexibility of specifying custom regressors.	Use to create linear model structures with nonlinearities embedded in the regressors.
Custom network	`idCustomNetwork`	Similar to sigmoid network but with a user-defined unit function.	User Defined. For advanced use.
Multilayered neural network	`idFeedforwardNetwork`	Any feedforward network that is created using Deep Learning Toolbox™ software, such as `feedforwardnet` or `cascadeforwardnet`.	Requires Deep Learning Toolbox.
Gaussian process (GP) regression model	`idGaussianProcess`	Kernel-based zero-mean Gaussian random process regression model.	Useful when measurement data is limited. Requires Statistics and Machine Learning Toolbox™.

Nonlinearity	Mapping Object	Description	Comments
Tree ensemble regression model	`idTreeEnsemble`	Decision tree ensemble regression model, which is an ensemble of binary decision trees.	Typically good predictive performance because using a tree combination reduces overfitting. Can invoke parallel processing. Requires Statistics and Machine Learning Toolbox.
Support vector machine (SVM) regression model	`idSupportVectorMachine`	Kernel-based SVM regression model with constraints.	Robust to outliers. Requires Statistics and Machine Learning Toolbox.

See Also

More About

- "What are Nonlinear ARX Models?" on page 11-14
- "Identifying Nonlinear ARX Models" on page 11-17

Estimate Nonlinear ARX Models in the App

You can estimate nonlinear ARX models in the **System Identification** app after you perform the following tasks:

- Import data into the System Identification app (see "Preparing Data for Nonlinear Identification" on page 11-13).
- (Optional) Choose a nonlinearity estimator in "Available Mapping Functions for Nonlinear ARX Models" on page 11-22.

To estimate a nonlinear ARX model using the imported estimation data and chosen nonlinearity estimators:

1 In the **System Identification** app, select **Estimate > Nonlinear ARX Models** to open the **Estimate Nonlinear ARX Models** dialog box.

2 (Optional) Edit **Model name** by deleting the default model name and entering a new name. The name of the model should be unique to all nonlinear ARX models in the System Identification app.

3 (Optional) If you want to refine the parameters of a previously estimated model or configure the model structure to match that of an existing model:

    a Click **Initialize**. A Initial Model Specification dialog box opens.

    b In the Initial Model drop-down list, select a nonlinear ARX model.

       The model must be in the Model Board of the System Identification app and the input/output dimensions of this initial model must match that of the estimation data, selected as **Working Data** in the app.

    c Click **OK**.

       The model structure as well as the parameter values are updated to match that of the selected model.

Clicking **Estimate** causes the estimation to use the parameters of the initial model as the starting point.

> **Note** When you select an initial model, you can optionally update the estimation algorithm settings to match those used for the initial model by selecting the **Inherit the model's algorithm properties** option.

4 Keep the default settings in the Nonlinear Models dialog box that specify the model structure and the algorithm, or modify these settings:

> **Note** For more information about available options, click **Help** in the Nonlinear Models dialog box to open the app help.

What to Configure	Options in Nonlinear Models GUI	Comment
Model order	In the **Regressors** tab, edit the **No. of Terms** corresponding to each input and output channel.	Model order *na* is the output number of terms and *nb* is the input number of terms.
Input delay	In the **Regressors** tab, edit the **Delay** corresponding to an input channel.	If you do not know the input delay value, click **Infer Input Delay**. This action opens the Infer Input Delay dialog box to suggest possible delay values.
Regressors	In the **Regressors** tab, click **Edit Regressors**.	This action opens the Model Regressors dialog box. Use this dialog box to create custom regressors or to include specific regressors in the nonlinear block.
Nonlinearity estimator	In the **Model Properties** tab.	To use all standard and custom regressors in the linear block only, you can exclude the nonlinear block by setting **Nonlinearity** to None.
Estimation algorithm	In the **Estimate** tab, click **Algorithm Options**.	

5 To obtain regularized estimates of model parameters, in the **Estimate** tab, click **Estimation Options**. Specify the regularization constants in the **Regularization_Tradeoff_Constant** and **Regularization_Weighting** fields. To learn more, see "Regularized Estimates of Model Parameters" on page 1-34.

6 Click **Estimate** to add this model to the System Identification app.

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 app. For more information about validating models, see "Validate Nonlinear ARX Models" on page 11-38.

If you get a poor fit, try changing the model structure or algorithm configuration in step 5.

You can export the model to the MATLAB workspace by dragging it to **To Workspace** in the **System Identification** app.

See Also

Apps
System Identification

More About

- "What are Nonlinear ARX Models?" on page 11-14
- "Available Mapping Functions for Nonlinear ARX Models" on page 11-22
- "Identifying Nonlinear ARX Models" on page 11-17
- "Validate Nonlinear ARX Models" on page 11-38
- "Using Nonlinear ARX Models" on page 11-42

Estimate Nonlinear ARX Models at the Command Line

You can estimate nonlinear ARX models after you perform the following tasks:

- Prepare your data, as described in "Preparing Data for Nonlinear Identification" on page 11-13.
- (Optional) Estimate model orders and delays the same way you would for linear ARX models. See "Preliminary Step – Estimating Model Orders and Input Delays" on page 6-8.
- (Optional) Choose a mapping function for the output function in "Available Mapping Functions for Nonlinear ARX Models" on page 11-22.
- (Optional) Estimate or construct a linear ARX model for initialization of nonlinear ARX model. See "Initialize Nonlinear ARX Estimation Using Linear Model" on page 11-20.

Estimate Model Using nlarx

Use `nlarx` to both construct and estimate a nonlinear ARX model. After each estimation, validate the model on page 11-38 by comparing it to other models and simulating or predicting the model response.

Basic Estimation

Start with the simplest estimation using `m = nlarx(data,[na nb nk])`. For example:

```
load iddata1;
% na = nb = 2 and nk = 1
m = nlarx(z1,[2 2 1])

m =

Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  Linear regressors in variables y1, u1

Output function: Wavelet network with 1 units
Sample time: 0.1 seconds

Status:
Estimated using NLARX on time domain data "z1".
Fit to estimation data: 68.83% (prediction focus)
FPE: 1.975, MSE: 1.885
More information in model's "Report" property.
```

View the regressors.

```
getreg(m)

ans = 4x1 cell
    {'y1(t-1)'}
    {'y1(t-2)'}
    {'u1(t-1)'}
    {'u1(t-2)'}
```

11-27

The second input argument `[na nb nk]` specifies the model orders and delays. By default, the output function is the wavelet network `idWaveletNetwork`, which accepts 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.

Create an `nlarxOptions` option set and configure the `Focus` property to minimize simulation error.

```
opt = nlarxOptions('Focus','simulation');
M = nlarx(z1,[2 2 1],'idSigmoidNetwork',opt);
```

Configure Model Regressors

Linear Regressors

Linear regressors represent linear functions that are based on delayed input and output variables and which provide the inputs into the model output function. When you use orders to specify a model, the number of input regressors is equal to `na` and the number of output regressors is equal to `nb`. The orders syntax limits you to consecutive lags. You can also create linear regressors using `linearRegressor`. When you use `linearRegressor`, you can specify arbitrary lags.

Polynomial Regressors

Explore including polynomial regressors using `polynomialRegressor` in addition to the linear regressors in the nonlinear ARX model structure. Polynomial regressors are polynomial functions of delayed inputs and outputs. (see "Nonlinear ARX Model Orders and Delay" on page 11-16).

For example, generate polynomial regressors up to order 2.

```
P = polynomialRegressor({'y1','u1'},{[1],[2]},2);
```

Append the polynomial regressors to the linear regressors in `m.Regressors.`

```
m.Regressors = [m.Regressors;P];
getreg(m)
```

```
ans = 8x1 cell
    {'y1(t-1)'   }
    {'y1(t-2)'   }
    {'u1(t-1)'   }
    {'u1(t-2)'   }
    {'y1(t-3)'   }
    {'y1(t-5)'   }
    {'y1(t-1)^2'}
    {'u1(t-2)^2'}
```

`m` now includes polynomial regressors.

View the size of the `m.Regressors` array.

```
size(m.Regressors)
```

```
ans = 1×2
```

```
        3       1
```

The array now contains three regressor objects.

Custom Regressors

Use `customRegressor` to construct regressors as arbitrary functions of model input and output variables.

.For example, create two custom regressors that implement `'sin(y1(t-1)'` and `'y1(t-2).*u1(t-3)'`.

```
C1 = customRegressor({'y1'},{1},@(x)sin(x));
C2 = customRegressor({'y1','u1'},{2,3},@(x,y)x.*y);
m.Regressors = [m.Regressors;C1;C2];
getreg(m) % displays all regressors

ans = 10x1 cell
    {'y1(t-1)'          }
    {'y1(t-2)'          }
    {'u1(t-1)'          }
    {'u1(t-2)'          }
    {'y1(t-3)'          }
    {'y1(t-5)'          }
    {'y1(t-1)^2'        }
    {'u1(t-2)^2'        }
    {'sin(y1(t-1))'     }
    {'y1(t-2).*u1(t-3)'}
```

View the properties of custom regressors. For example, get the function handle of the first custom regressor in the array. This regressor is the fourth regressor set in the `Regressors` array.

```
C1_fcn = m.Regressors(4).VariablesToRegressorFcn

C1_fcn = function_handle with value:
    @(x)sin(x)
```

View the regressor description.

```
display(m.Regressors(4))

Custom regressor: sin(y1(t-1))
    VariablesToRegressorFcn: @(x)sin(x)
                  Variables: {'y1'}
                       Lags: {[1]}
                 Vectorized: 1
               TimeVariable: 't'

  Regressors described by this set
```

Combine Regressors

Once you have created linear, polynomial, and custom regressor objects, you can combine them in any way you want to suit your estimation needs.

Specify Regressor Inputs to Linear and Nonlinear Components

Model regressors can enter as inputs to either or both linear and nonlinear function components of the mapping functions making up the output function. To reduce model complexity and keep the estimation well-conditioned, consider assigning a reduced set of regressors to the nonlinear component. You can also assign a subset of regressors to the linear component. The regressor usage table that manages the assignments provides complete flexibility. You can assign any combination of regressors to each component. For example, specify a nonlinear ARX model to be linear in past outputs and nonlinear in past inputs.

```
m = nlarx(z1,[2 2 1]);
disp(m.RegressorUsage)

                 y1:LinearFcn     y1:NonlinearFcn
                _____     _____

      y1(t-1)       true              true
      y1(t-2)       true              true
      u1(t-1)       true              true
      u1(t-2)       true              true

m.RegressorUsage{3:4,1} = false;
m.RegressorUsage{1:2,2} = false;
disp(m.RegressorUsage)

                 y1:LinearFcn     y1:NonlinearFcn
                _____     _____

      y1(t-1)       true              false
      y1(t-2)       true              false
      u1(t-1)       false             true
      u1(t-2)       false             true
```

Configure Output Function

The following table summarizes available mapping objects for the model output function.

Mapping Description	Value (Default Mapping Object Configuration)	Mapping Object
Wavelet network (default)	`'idWaveletNetwork'` or `'wave'`	`idWaveletNetwork`
One layer sigmoid network	`'idSigmoidNetwork'` or `'sigm'`	`idSigmoidNetwork`
Tree partition	`'idTreePartition'` or `'tree'`	`idTreePartition`
F is linear in x	`'idLinear'` or `[ ]` or `''`	`idLinear`

Additional available mapping objects 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 Deep Learning Toolbox software. See the `idFeedforwardNetwork` reference page.

Specify a custom network by defining a function called `gaussunit.m`, as described in the `idCustomNetwork` reference page. Define the custom network object `CNetw` and estimate the model:

```
CNetw = idCustomNetwork(@gaussunit);
m = nlarx(data,[na nb nk],CNetw)
```

Exclude Linear Function in Output Function

If your model output function includes `idWaveletNetwork`, `idSigmoidNetwork`, or `idCustomNetwork` mapping objects, you can exclude the linear function using the `LinearFcn.Use` property of the mapping object. The mapping object becomes $F(x)=g(Q(x-r))+y_0$, where $g(.)$ is the nonlinear function and y_0 is the offset.
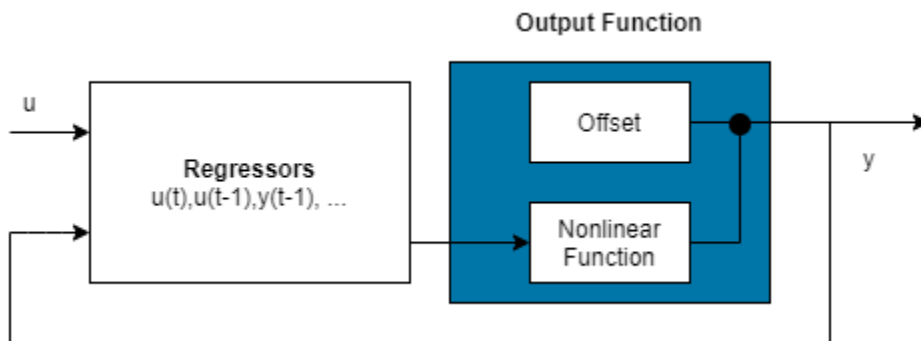
Output Function

Note You cannot exclude the linear function from tree partition and neural network mapping objects.

Exclude Nonlinear Function in Output Function

Configure the nonlinear ARX structure to include only the linear function in the mapping object by setting the mapping object to `idLinear`. In this case, $F(x) = L^T(x-r) + y_0$ 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.

Output Function
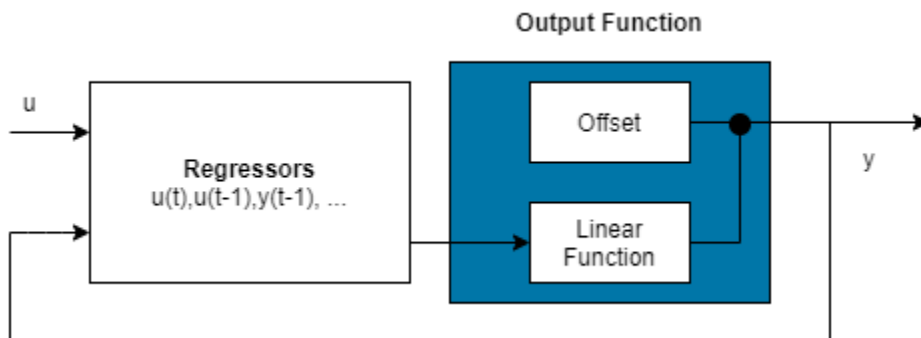
A popular nonlinear ARX configuration in many applications uses polynomial regressors to model system nonlinearities. In such cases, the system is considered to be a linear combination of products of (delayed) input and output variables. Use the `polynomialRegressor` command to easily generate combinations of regressor products and powers.

For example, suppose that you know the output $y(t)$ of a system to be a linear combination of ($y(t - 1))^2$, $(u(t - 1))^2$ and $y(t - 1)u(t - 1)$). To model such a system, use:

```
P = polynomialRegressor({'y1','u1'},{1,1},2)

P =
Order 2 regressors in variables y1, u1
               Order: 2
            Variables: {'y1'  'u1'}
                 Lags: {[1]  [1]}
           UseAbsolute: [0 0]
      AllowVariableMix: 0
           AllowLagMix: 0
          TimeVariable: 't'

  Regressors described by this set

P.AllowVariableMix = true;
M = nlarx(z1,P,idLinear);
getreg(M)

ans = 3x1 cell
    {'y1(t-1)^2'       }
    {'u1(t-1)^2'       }
    {'y1(t-1)*u1(t-1)'}
```

For more complex combinations of polynomial delays and mixed-variable regressors, you can also use `customRegressor`.

Iteratively Refine Model

If your model structure includes mapping objects that support iterative search (see "Specify Estimation Options for Nonlinear ARX Models" on page 11-19), you can use `nlarx` to refine model parameters:

```
m1 = nlarx(z1,[2 2 1],idSigmoidNetwork);
m2 = nlarx(z1,m1); % can repeatedly run this command
```

You can also use `pem` to refine the original model:

```
m2 = pem(z1,m1);
```

Check the search termination criterion `m.Report.Termination.WhyStop`. If `WhyStop` indicates that the estimation reached the maximum number of iterations, try repeating the estimation and possibly specifying a larger value for the `nlarxOptions.SearchOptions.MaxIterations` estimation option:

```
opt = nlarxOptions;
opt.SearchOptions.MaxIterations = 30;
m2 = nlarx(z1,m1,opt); % runs 30 more iterations
                       % starting from m1
```

When the `m.Report.Termination.WhyStop` value is `Near (local) minimum, (norm( g) < tol` or `No improvement along the search direction with line search`, validate your model to see if this model adequately fits the data. If not, the solution might be stuck in a local

minimum of the cost-function surface. Try adjusting the SearchOptions.`Tolerance` value or the `SearchMethod` option in the nlarxOptions option set, and repeat the estimation.

You can also try perturbing the parameters of the last model using `init` (called *randomization*) and refining the model using `nlarx`:

```
M1 = nlarx(z1, [2 2 1], idSigmoidNetwork); % original model
M1p = init(M1);      % randomly perturbs parameters about nominal values
M2 = nlarx(z1, M1p);   % estimates parameters of perturbed model
```

You can display the progress of the iterative search in the MATLAB Command Window using the `nlarxOptions.Display` estimation option:

```
opt = nlarxOptions('Display','on');
M2= nlarx(z1,M1p,opt);
```

Troubleshoot Estimation

If you do not get a satisfactory model after many trials with various model structures and algorithm settings, it is possible that the data is poor. For example, your data might be missing important input or output variables and does not sufficiently cover all the operating points of the system.

Nonlinear black-box system identification usually requires more data than linear model identification to gain enough information about the system.

Use nlarx to Estimate Nonlinear ARX Models

This example shows how to use `nlarx` to estimate a nonlinear ARX model for measured input/output data.

Prepare the data for estimation.

```
load twotankdata
z = iddata(y, u, 0.2);
ze = z(1:1000); zv = z(1001:3000);
```

Estimate several models using different model orders, delays, and nonlinearity settings.

```
m1 = nlarx(ze,[2 2 1]);
m2 = nlarx(ze,[2 2 3]);
m3 = nlarx(ze,[2 2 3],idWaveletNetwork(8));
```

An alternative way to perform the estimation is to configure the model structure first, and then to estimate this model.

```
m4 = idnlarx([2 2 3],idSigmoidNetwork(14));
m4.RegressorUsage.("y1:NonlinearFcn")(3:4) = false;
m4 = nlarx(ze,m4);
```

Compare the resulting models by plotting the model outputs with the measured output.

```
compare(zv, m1,m2,m3,m4)
```

See Also

Functions
`idnlarx` | `nlarx`

More About

Estimate Nonlinear ARX Models Initialized Using Linear ARX Models

This example shows how to estimate nonlinear ARX models by using linear ARX models.

Load the estimation data.

```
load throttledata.mat
```

This command loads the data object `ThrottleData` into the workspace. The object contains input and output samples collected from an engine throttle system, sampled at a rate of 100 Hz.

A DC motor controls the opening angle of the butterfly valve in the throttle system. A step signal (in volts) drives the DC motor. The output is the angular position (in degrees) of the valve.

Plot the data to view and analyze the data characteristics.

```
plot(ThrottleData)
```


In the normal operating range of 15-90 degrees, the input and output variables have a linear relationship. You use a linear model of low order to model this relationship.

In the throttle system, a hard stop limits the valve position to 90 degrees, and a spring brings the valve to 15 degrees when the DC motor is turned off. These physical components introduce nonlinearities that a linear model cannot capture.

Estimate an ARX model to model the linear behavior of this single-input single-output system in the normal operating range.

Detrend the data because linear models cannot capture offsets.

```
Tr = getTrend(ThrottleData);
Tr.OutputOffset = 15;
DetrendedData = detrend(ThrottleData,Tr);
```

Estimate a linear ARX model with na=2, nb=1, nk=1.

```
opt = arxOptions('Focus','simulation');
LinearModel = arx(DetrendedData,[2 1 1],opt);
```

Compare the simulated model response with the estimation data.
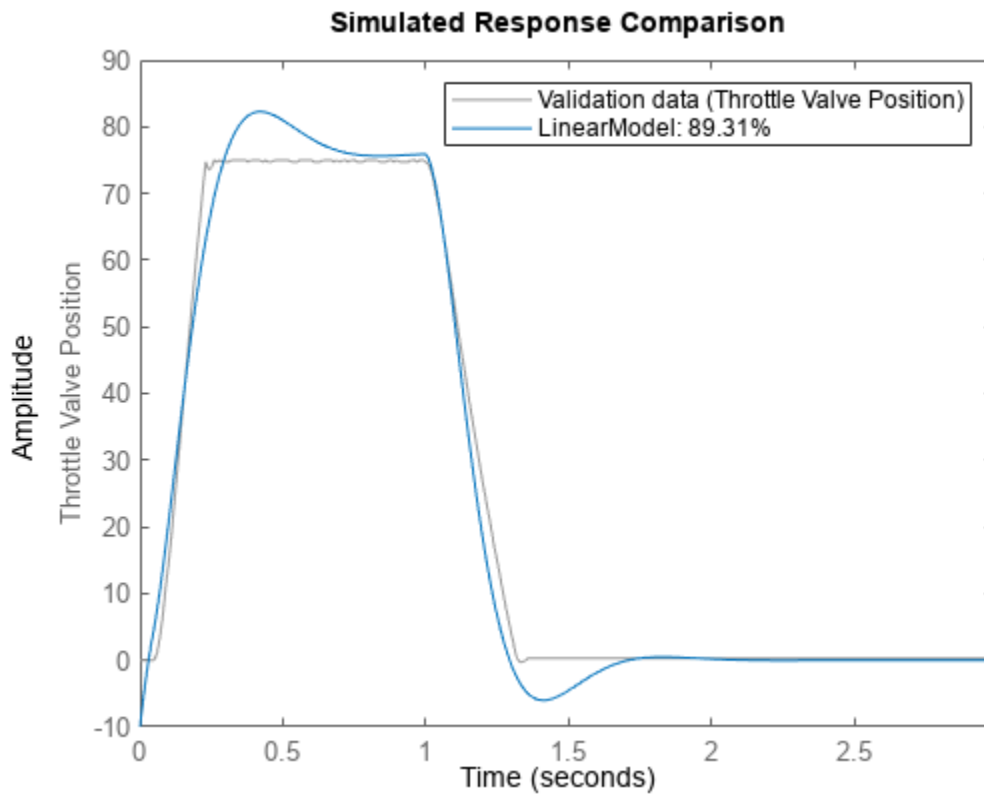
```
compare(DetrendedData, LinearModel)
```


The linear model captures the rising and settling behavior in the linear operating range but does not account for output saturation at 90 degrees.

Estimate a nonlinear ARX model to model the output saturation.

```
optNL = nlarxOptions('Focus','simulation');
NonlinearModel = nlarx(ThrottleData,LinearModel,'idSigmoidNetwork',optNL);
```

The software uses the orders and delay of the linear model for the orders of the nonlinear model. In addition, the software computes the linear function of the `idSigmoidNetwork` nonlinearity estimator.

Compare the nonlinear model with the estimation data.

```
compare(ThrottleData, NonlinearModel)
```
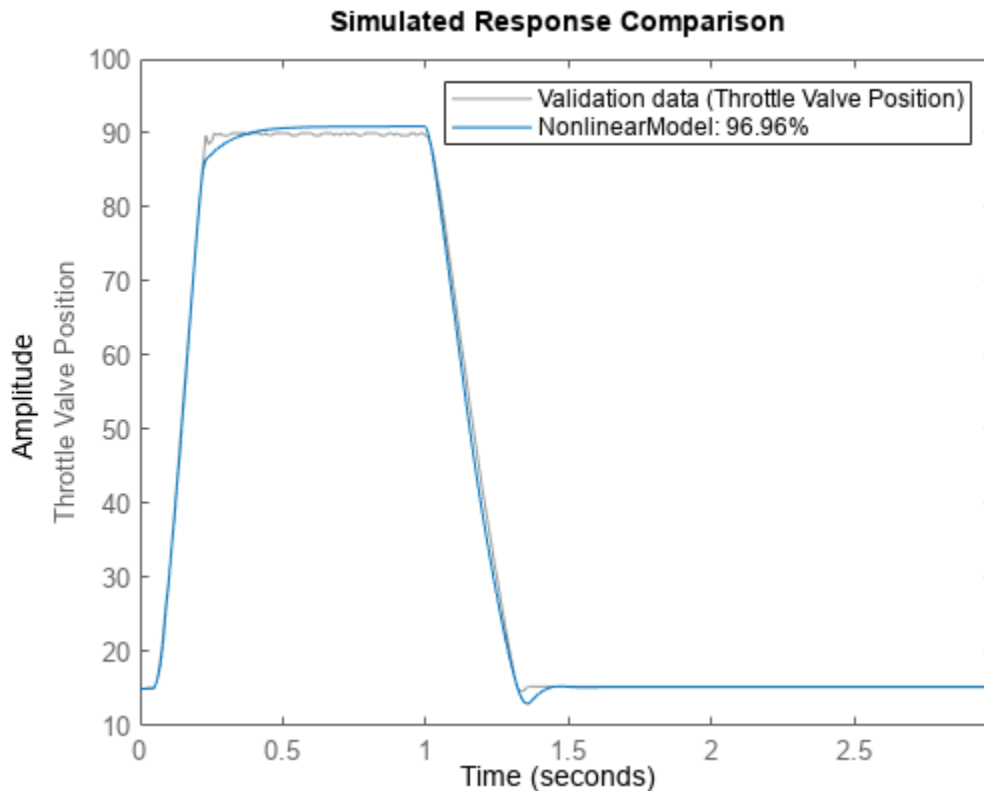

The model captures the nonlinear effects (output saturation) and improves the overall fit to data.

See Also

More About

- "Identifying Nonlinear ARX Models" on page 11-17
- "Estimate Nonlinear ARX Models in the App" on page 11-24
- "Estimate Nonlinear ARX Models at the Command Line" on page 11-27

Validate Nonlinear ARX Models

After estimating a nonlinear ARX model on page 11-17 for your system, you can validate whether it reproduces the system behavior within acceptable bounds. You can validate your model in different ways. It is recommended that you use separate data sets for estimating and validating your model. If the validation indicates low confidence in the estimation, then see "Troubleshooting Model Estimation" on page 17-96 for next steps. For general information about validating models, see "Model Validation".

Compare Model Output to Measured Output

Plot simulated or predicted model output and measured output data for comparison, and compute best fit values. At the command line, use `compare` command. You can also use `sim` and `predict` to simulate and predict model response. For information about plotting simulated and predicted output in the app, see "Simulation and Prediction in the App" on page 17-15.

Check Iterative Search Termination Conditions

The estimation report that is generated after model estimation lists the reason the software terminated the estimation. For example, suppose that the report indicates that the estimation reached the maximum number of iterations. You can try repeating the estimation by specifying a larger value for the maximum number of iterations. For information about how to configure the maximum number of iterations and other estimation options, see "Specify Estimation Options for Nonlinear ARX Models" on page 11-19.

To view the estimation report in the app, after model estimation is complete, view the **Estimation Report** area of the **Estimate** tab. At the command line, use `M.Report.Termination` to display the estimation termination conditions, where `M` is the estimated Nonlinear ARX model. For example, check the `M.Report.Termination.WhyStop` field that describes why the estimation was stopped.

For more information about the estimation report, see "Estimation Report" on page 1-21.

Check the Final Prediction Error and Loss Function Values

You can compare the performance of several estimated models by comparing the final prediction error and loss function values that are shown in the estimation report.

To view these values for an estimated model `M` at the command line, use the `M.Report.Fit.FPE` (final prediction error) and `M.Report.Fit.LossFcn` (value of loss function at estimation termination) properties. Smaller values typically indicate better performance. However, `M.Report.Fit.FPE` values can be unreliable when the model contains many parameters relative to the estimation data size. Use these indicators with other validation techniques to draw reliable conclusions.

Perform Residual Analysis

Residuals are differences between the model output and the measured output. Thus, residuals represent the portion of the output not explained by the model. You can analyze the residuals using techniques such as the whiteness test and the independence test. For more information about these tests, see "What Is Residual Analysis?" on page 17-43.

At the command line, use `resid` to compute, plot, and analyze the residuals. To plot residuals in the app, see "How to Plot Residuals in the App" on page 17-46.

Examine Nonlinear ARX Plots

A nonlinear ARX plot displays the evaluated model nonlinearity for a chosen model output as a function of one or two model regressors. For a model `M`, the model nonlinearity (`M.Nonlinearity`) is a nonlinearity estimator function, such as `idWaveletNetwork`, `idSigmoidNetwork`, or `idTreePartition`, that uses model regressors as its inputs.
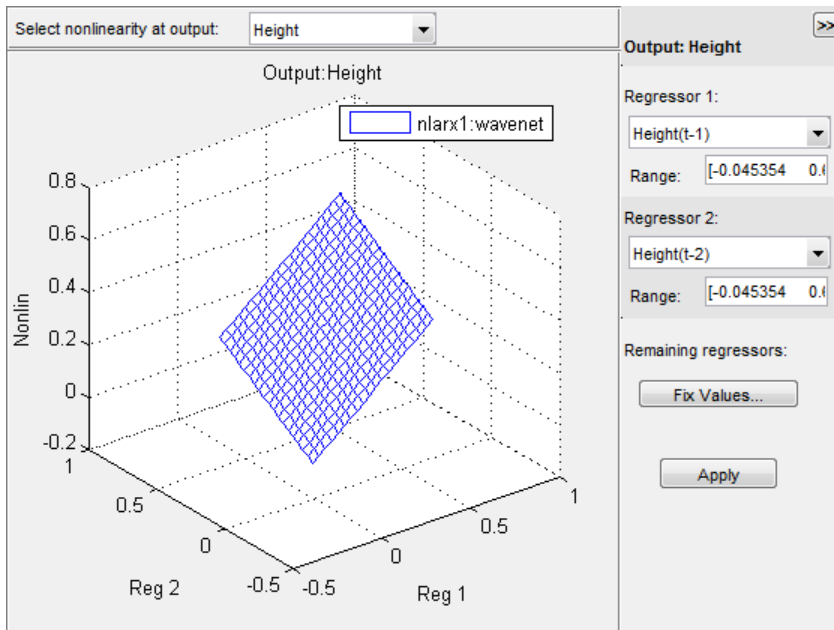
To understand what is plotted, suppose that `{r1,r2,…,rN}` are the `N` regressors used by a nonlinear ARX model `M` with nonlinearity `nl` corresponding to a model output. You can use `getreg(M)` to view these regressors. The expression `Nonlin = evaluate(nl,[v1,v2,...,vN])` returns the model output for given values of these regressors, that is, `r1 = v1`, `r2 = v2`, ..., `rN = vN`. For plotting the nonlinearities, you select one or two of the `N` regressors, for example, `rsub = {r1,r4}`. The software varies the values of these regressors in a specified range, while fixing the value of the remaining regressors, and generates the plot of `Nonlin` vs. `rsub`. By default, the software sets the values of the remaining fixed regressors to their estimated means, but you can change these values. The regressor means are stored in the `Nonlinearity.Parameters.RegressorMean` property of the model.

Examining a nonlinear ARX plot can help you gain insight into which regressors have the strongest effect on the model output. Understanding the relative importance of the regressors on the output can help you decide which regressors to include in the nonlinear function for that output. If the shape of the plot looks like a plane for all the chosen regressor values, then the model is probably linear in those regressors. In this case, you can remove the corresponding regressors from nonlinear block, and repeat the estimation.

Furthermore, you can create several nonlinear models for the same data using different nonlinearity estimators, such a `idWaveletNetwork` network and `idTreePartition`, and then compare the nonlinear surfaces of these models. Agreement between plots for various models increases the confidence that these nonlinear models capture the true dynamics of the system.

Creating a Nonlinear ARX Plot

To create a nonlinear ARX plot in the **System Identification** app, select the **Nonlinear ARX** check box in the **Model Views** area. To include or exclude a model on the plot, click the corresponding model icon in the app. For general information about creating and working with plots in the app, see "Working with Plots" on page 21-8.

At the command line, after you have estimated a nonlinear ARX model M, use `plot` to view the shape of the nonlinearity.

`plot(M)`

You can use additional `plot` arguments to specify the following information:

* Include multiple nonlinear ARX models on the plot.
* Configure the regressor values for computing the nonlinearity values.

Configuring a Nonlinear ARX Plot

To configure the nonlinear ARX plot:

1 Select the output channel in the **Select nonlinearity at output** drop-down list. The nonlinearity values that correspond to the selected output channel are displayed.

2 Select **Regressor 1** from the list of available regressors. In the **Range** field, enter the range of values to include on the plot for this regressor. The regressor values are plotted on the **Reg1** axis of the plot.

If the regressor selection options are not visible, click to expand the Nonlinear ARX Model Plot window.

3 Specify **Regressor 2** as one of the following options:

* To display three axes on the plot, select **Regressor 2**. In the **Range** field, enter the range of values to include on the plot for this regressor. The regressor values are plotted on the **Reg2** axis of the plot.
* To display only two axes, select `<none>` in the **Regressor 2** list.

4 Fix the values of the regressors that are not displayed by clicking **Fix Values**. In the Fix Regressor Values dialog box, double-click the **Value** cell to edit the constant value of the corresponding regressor. The default values are determined during model estimation. Click **OK**.

If you generate the nonlinear ARX plot in the app, you can perform the following additional tasks:

Action	Procedure
Change the grid spacing of the regressors along each axis.	In the plot window, select **Options** > **Set number of samples**, and enter the number of samples to use for each regressor. Click **Apply** and then **Close**. For example, if the number of samples is 20, each regressor variable contains 20 points in its specified range. For a 3-D plots, this results in evaluating the nonlinearity at 20 x 20 = 400 points.
Change axis limits.	Select **Options** > **Set axis limits** to open the Axis Limits dialog box, and edit the limits. Click **Apply**.
Hide or show the plot legend.	Select **Style** > **Legend**. Select this option again to show the legend.
Rotate in three dimensions. **Note** Available only when you have selected two regressors as independent variables.	Select **Style** > **Rotate 3D** and drag the axes on the plot to a new orientation. To disable three-dimensional rotation, select **Style** > **Rotate 3D** again.

See Also
compare | sim | resid | predict

More About
- "Identifying Nonlinear ARX Models" on page 11-17
- "Estimate Nonlinear ARX Models in the App" on page 11-24
- "Estimate Nonlinear ARX Models at the Command Line" on page 11-27
- "Using Nonlinear ARX Models" on page 11-42

Using Nonlinear ARX Models

After identifying a nonlinear ARX model, you can use the model for the following tasks:

- **Simulation and prediction** — At the command line, use `sim` and `predict` to simulate and predict the model output. To compare models to measured output and to each other, use `compare`. For information about plotting simulated and predicted output in the app, see "Simulation and Prediction in the App" on page 17-15. You can also specify the initial conditions for simulation and prediction. The toolbox provides several options to facilitate how you specify initial states. For example, you can use `findstates` and `data2state` to compute state values based on the requirement to maximize fit to measured output or based on operating conditions. See the `idnlarx` reference page for a definition of the nonlinear ARX model states. To learn more about how `sim` and `predict` compute the model output, see "How the Software Computes Nonlinear ARX Model Output" on page 11-43.

  You can also forecast the response of a dynamic system by using the `forecast` command. The command predicts future outputs of the system using past output measurements. For more information, see "Forecasting Response of Nonlinear ARX Models" on page 14-41.

- **Linearization** — Compute linear approximation of nonlinear ARX models using `linearize` or `linapp`.

  The `linearize` command provides a first-order Taylor series approximation of the system about an operating point. `linapp` computes a linear approximation of a nonlinear model for a given input data. For more information, see the "Linear Approximation of Nonlinear Black-Box Models" on page 11-50. You can compute the operating point for linearization using `findop`.

  After computing a linear approximation of a nonlinear model, you can perform linear analysis and control design on your model using Control System Toolbox commands. For more information, see "Using Identified Models for Control Design Applications" on page 19-2 and "Create and Plot Identified Models Using Control System Toolbox Software" on page 19-5.

- **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 `idCustomNetwork` or `idFeedforwardNetwork` nonlinearity estimator, or custom regressors. For more information, see "Simulate Identified Model in Simulink" on page 20-5.

See Also

More About

- "What are Nonlinear ARX Models?" on page 11-14
- "Identifying Nonlinear ARX Models" on page 11-17
- "Linear Approximation of Nonlinear Black-Box Models" on page 11-50

How the Software Computes Nonlinear ARX Model Output

This topic describes how the software evaluates the output of nonlinearity estimators and uses this output to compute the response of a nonlinear ARX model.

Evaluating Nonlinearities

Evaluating the predicted output of a nonlinearity for a specific regressor value *x* requires that you first extract the nonlinearity *F* and regressors from the model:

```
F = m.Nonlinearity;
x = getreg(m,'all',data) % computes regressors
```

Evaluate *F*(*x*):

```
y = evaluate(F,x)
```

where x is a row vector of regressor values.

You can also evaluate predicted output values at multiple time instants by evaluating *F* for several regressor vectors simultaneously:

```
y = evaluate(F,[x1;x2;x3])
```

Simulation and Prediction of Sigmoid Network

This example shows how the software computes the simulated and predicted output of a nonlinear ARX model as a result of evaluating the output of its nonlinearity estimator for given regressor values.

Estimating and Exploring a Nonlinear ARX Model

Estimate nonlinear ARX model with sigmoid network nonlinearity.

```
load twotankdata
estData = iddata(y,u,0.2,'Tstart',0);
M = nlarx(estData,[1 1 0],'idsig');
```

Inspect the model properties and estimation result.

```
present(M)


M =

Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  Linear regressors in variables y1, u1
  List of all regressors

Output function: Sigmoid network with 10 units
Sample time: 0.2 seconds
```

```
Status:
Estimated using NLARX on time domain data "estData".
Fit to estimation data: 96.31% (prediction focus)
FPE: 4.804e-05, MSE: 4.666e-05

Model Properties
```

This command provides information about input and output variables, regressors, and nonlinearity estimator.

Inspect the nonlinearity estimator.

```
NL = M.Nonlinearity; % equivalent to M.nl
class(NL)    % nonlinearity class

ans =
'idSigmoidNetwork'
```

```
display(NL) % equivalent to NL

NL =
Sigmoid Network
Inputs: y1(t-1), u1(t)
Output: y1(t)

 Nonlinear Function: Sigmoid network with 10 units
 Linear Function: initialized to [-0.161 -0.105]
 Output Offset: initialized to 0.00119

       Inputs: {'y1(t-1)'  'u1(t)'}
      Outputs: {'y1(t)'}
  NonlinearFcn: 'Sigmoid units and their parameters'
     LinearFcn: 'Linear function parameters'
        Offset: 'Offset parameters'
```

Inspect the sigmoid network parameter values.

```
NL.Parameters;
```

Prediction of Output

The model output is:

y1(t)= f(y1(t-1),u1(t))

where f is the sigmoid network function. The model regressors *y1(t-1)* and *u1(t)* are inputs to the nonlinearity estimator. Time t is a discrete variable representing kT, where k = 0, 1, ... , and T is the sampling interval. In this example, T=0.2 second.

The output prediction equation is:

yp(t)=f(y1_meas(t-1),u1_meas(t))

where *yp(t)* is the predicted value of the response at time t. *y1_meas(t-1)* and *u1_meas(t)* are the measured output and input values at times *t-1* and t, respectively.

Computing the predicted response includes:

- Computing regressor values from input-output data.
- Evaluating the nonlinearity for given regressor values.

To compute the predicted value of the response using initial conditions and current input:

Estimate model from data and get nonlinearity parameters.

```
load twotankdata
estData = iddata(y,u,0.2,'Tstart',0);
M = nlarx(estData,[1 1 0],'idsig');
NL = M.Nonlinearity;
```

Specify zero initial states.

```
x0 = 0;
```

The model has one state because there is only one delayed term $y1(t-1)$. The number of states is equal to `sum(getDelayInfo(M))`.

Compute the predicted output at time $t=0$.

```
RegValue = [0,estData.u(1)]; % input to nonlinear function f
yp_0 = evaluate(NL,RegValue);
```

`RegValue` is the vector of regressors at `t=0` . The predicted output is $yp(t=0)=f(y1\_meas(t=-1),u1\_meas(t=0))$. In terms of MATLAB variables, this output is `f(0,estData.u(1))`, where

- $y1\_meas(t=0)$ is the measured output value at `t=0`, which is to `estData.y(1)`.
- $u1\_meas(t=1)$ is the second input data sample `estData.u(2)`.

Perform one-step-ahead prediction at all time values for which data is available.

```
RegMat = getreg(M,[],estData,x0);
yp = evaluate(NL,RegMat.Variables);
```

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_direct = predict(M,estData,1,'InitialState',x0);
% compare
t = estData.SamplingInstants;
plot(t,yp, t,yp_direct.OutputData,'.')
```
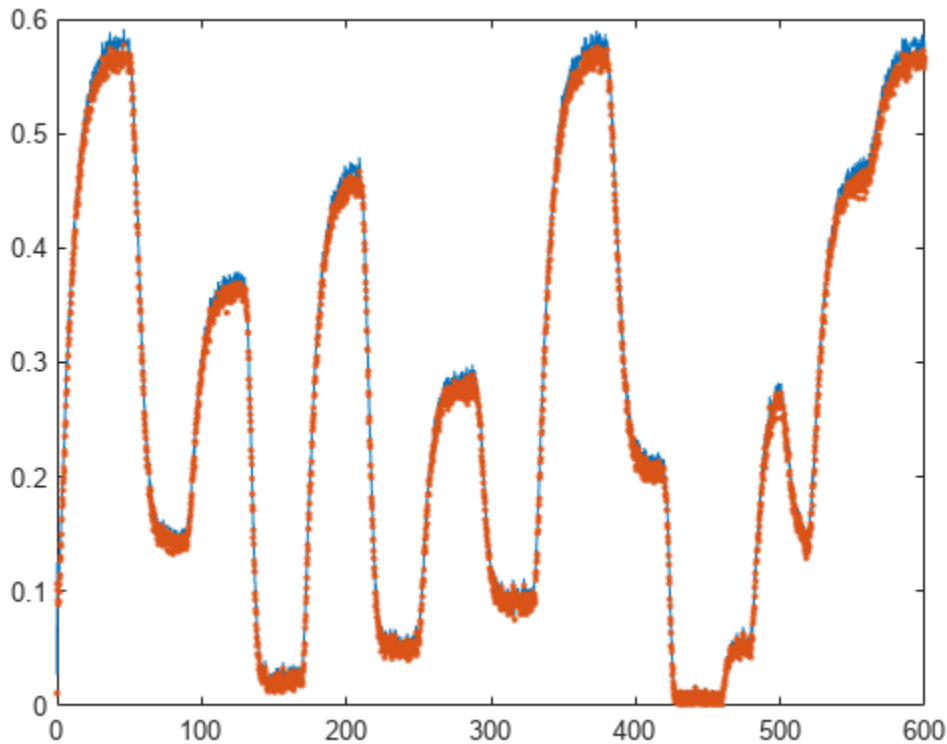
Simulation of Output

The model output is:

y1(t)=f(y1(t-1),u1(t))

where *f* is the sigmoid network function. The model regressors *y1(t-1)* and *u1(t)* are inputs to the nonlinearity estimator. Time *t* is a discrete variable representing *kT* , where *k*= 0, 1,.., and *T* is the sampling interval. In this example, *T*=0.2 second.

The simulated output is:

ys(t) = f(ys(t-1),u1_meas(t))

where *ys(t)* is the simulated value of the response at time *t*. The simulation equation is the same as the prediction equation, except that the past output value `ys(t-1)` results from the simulation at the previous time step, rather than the measured output value.

Computing the simulated response includes:

*   Computing regressor values from input-output data using simulated output values.
*   Evaluating the nonlinearity for given regressor values.

To compute the simulated value of the response using initial conditions and current input:

Estimate model from data and get nonlinearity parameters.

```
load twotankdata
estData = iddata(y,u,0.2,'Tstart',0);
M = nlarx(estData,[1 1 0],'idSigmoidNetwork');
NL = M.Nonlinearity;
```

Specify zero initial states.

```
x0 = 0;
```

The model has one state because there is only one delayed term *y1(t-1)*. The number of states is equal to `sum(getDelayInfo(M))`.

Compute the simulated output at time *t* =0, *ys(t=0)*.

```
RegValue = [0,estData.u(1)];
ys_0 = evaluate(NL,RegValue);
```

RegValue is the vector of regressors at *t*=0. *ys(t=0)=f(y1(t=-1),u1_meas(t=0))*. In terms of MATLAB variables, this output is `f(0,estData.u(1))`, where

- *y1(t=-1)* is the initial state `x0 (=0)`.
- *u1_meas(t=0)* is the value of the input at *t* =0, which is the first input data sample `estData.u(1)`.

Compute the simulated output at time *t*=1, *ys(t=1)*.

```
RegValue = [ys_0,estData.u(2)];
ys_1 = evaluate(NL,RegValue);
```

The simulated output *ys(t=1)=f(ys(t=0),u1_meas(t=1))*. In terms of MATLAB variables, this output is `f(ys_0,estData.u(2))`, where

- *ys(t=0)* is the simulated value of the output at t=0.
- *u1_meas(t=1)* is the second input data sample estData.u(2).

Compute the simulated output at time *t*=2.

```
RegValue = [ys_1,estData.u(3)];
ys_2 = evaluate(NL,RegValue);
```

Unlike for output prediction, you cannot use `getreg` to compute regressor values for all time values. You must compute regressors values at each time sample separately because the output samples required for forming the regressor vector are available iteratively, one sample at a time.

These steps are equivalent to the simulated response computed in a single step using `sim(idnlarx)`:

```
ys = sim(M,estData,x0);
```

Nonlinearity Evaluation

This example performs a low-level computation of the nonlinearity response for the `idSigmoidNetwork` function:

$$F(x) = (x - r)PL + a_1 f((x - r)Qb_1 + c_1) + \dots$$
$$+ a_n f((x - r)Qb_n + c_n) + d$$

where *f* is the sigmoid function, given by the following equation:

$$f(z) = \frac{1}{e^{-z} + 1}$$

In `F(x)`, the input to the sigmoid function is `x-r`. `x` is the regressor value and `r` is regressor mean, computed from the estimation data. a_n , n_n, and c_n are the network parameters stored in the model property `M.nl.par`, where `M` is an `idnlarx` object.

Compute the output value at time t=1, when the regressor values are `x=[estData.y(1),estData.u(2)]`:

Estimate model from sample data.

```
load twotankdata
estData = iddata(y,u,0.2,'Tstart',0);
M = nlarx(estData,[1 1 0],idSigmoidNetwork);
NL = M.OutputFcn;
```

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.Input.Mean;
P = NL.LinearFcn.InputProjection;
L = NL.LinearFcn.Value';
d = NL.Offset.Value;
Q = NL.NonlinearFcn.Parameters.InputProjection;
aVec = NL.NonlinearFcn.Parameters.OutputCoefficient;   % [a_1; a_2; ...]
cVec = NL.NonlinearFcn.Parameters.Translation;   % [c_1; c_2; ...]
bMat = NL.NonlinearFcn.Parameters.Dilation;      % [b_1; b_2; ...]
```

Compute the linear portion of the response (plus offset).

```
yLinear = (x-r)*P*L+d;
```

Compute the nonlinear portion of the response.

```
f = @(z)1/(exp(-z)+1); % anonymous function for sigmoid unit
yNonlinear = 0;
for k = 1:length(aVec)
    fInput = (x-r)*Q* bMat(:,k)+cVec(k);
    yNonlinear = yNonlinear+aVec(k)*f(fInput);
end
```

Compute total response `y = F(x) = yLinear + yNonlinear`.

```
y = yLinear + yNonlinear;
```

`y` is equal to `evaluate(NL,x)`.

See Also
evaluate

More About
- "What are Nonlinear ARX Models?" on page 11-14

- "Identifying Nonlinear ARX Models" on page 11-17

Linear Approximation of Nonlinear Black-Box Models

Why Compute a Linear Approximation of a Nonlinear Model?

Control design and linear analysis techniques using Control System Toolbox software require linear models. You can use your estimated nonlinear model in these applications after you linear the model. After you linearize your model, you can use the model for control design and linear analysis.

Choosing Your Linear Approximation Approach

System Identification Toolbox software provides two approaches for computing a linear approximation of nonlinear ARX on page 11-17 and Hammerstein-Wiener on page 12-5 models.

To compute a linear approximation of a nonlinear model for a given input signal, use the `linapp` command. The resulting model is only valid for the same input that you use to compute the linear approximation. For more information, see "Linear Approximation of Nonlinear Black-Box Models for a Given Input" on page 11-50.

If you want a tangent approximation of the nonlinear dynamics that is accurate near the system operating point, use the `linearize` command. The resulting model is a first-order Taylor series approximation for the system about the operating point, which is defined by a constant input and model state values. For more information, see "Tangent Linearization of Nonlinear Black-Box Models" on page 11-51.

Linear Approximation of Nonlinear Black-Box Models for a Given Input

`linapp` computes the best linear approximation, in a mean-square-error sense, of a nonlinear ARX or Hammerstein-Wiener model for a given input or a randomly generated input. The resulting linear model might only be valid for the same input signal as you the one you used to generate the linear approximation.

`linapp` estimates the best linear model that is structurally similar to the original nonlinear model and provides the best fit between a given input and the corresponding simulated response of the nonlinear model.

To compute a linear approximation of a nonlinear black-box model for a given input, you must have these variables:

- Nonlinear ARX model (`idnlarx` object) or Hammerstein-Wiener model (`idnlhw` object)
- Input signal for which you want to obtain a linear approximation, specified as a real matrix or an `iddata` object

`linapp` uses the specified input signal to compute a linear approximation:

- For nonlinear ARX models, `linapp` estimates a linear ARX model using the same model orders `na`, `nb`, and `nk` as the original model.
- For Hammerstein-Wiener models, `linapp` estimates a linear Output-Error (OE) model using the same model orders `nb`, `nf`, and `nk`.

To compute a linear approximation of a nonlinear black-box model for a randomly generated input, you must specify the minimum and maximum input values for generating white-noise input with a magnitude in this rectangular range, `umin` and `umax`.

For more information, see the `linapp` reference page.

Tangent Linearization of Nonlinear Black-Box Models

`linearize` computes a first-order Taylor series approximation for nonlinear system dynamics about an *operating point*, which is defined by a constant input and model state values. The resulting linear model is accurate in the local neighborhood of this operating point.

To compute a tangent linear approximation of a nonlinear black-box model, you must have these variables:

- Nonlinear ARX model (`idnlarx` object) or Hammerstein-Wiener model (`idnlhw` object)
- Operating point

To specify the operating point of your system, you must specify the constant input and the states. For more information about state definitions for each type of parametric model, see these reference pages:

- `idnlarx` — Nonlinear ARX model
- `idnlhw` — Nonlinear Hammerstein-Wiener model

If you do not know the operating point values for your system, see "Computing Operating Points for Nonlinear Black-Box Models" on page 11-51.

For more information, see the `idnlarx/linearize` or `idnlhw/linearize` reference page.

Computing Operating Points for Nonlinear Black-Box Models

An *operating point* is defined by a constant input and model state values.

If you do not know the operating conditions of your system for linearization, you can use `findop` to compute the operating point from specifications.

Computing Operating Point from Steady-State Specifications

Use `findop` to compute an operating point from steady-state specifications:

- Values of input and output signals.
  If either the steady-state input or output value is unknown, you can specify it as `NaN` to estimate this value. This is especially useful when modeling MIMO systems, where only a subset of the input and output steady-state values are known.
- More complex steady-state specifications.

  Construct an object that stores specifications for computing the operating point, including input and output bounds, known values, and initial guesses. For more information, see `idnlarx/operspec` or `idnlhw/operspec`.

For more information, see the `idnlarx/findop` or `idnlhw/findop` reference page.

Computing Operating Points at a Simulation Snapshot

Compute an operating point at a specific time during model simulation (snapshot) by specifying the snapshot time and the input value. To use this method for computing the equilibrium operating point,

choose an input that leads to a steady-state output value. Use that input and the time value at which the output reaches steady state (*snapshot* time) to compute the operating point.

It is optional to specify the initial conditions for simulation when using this method because initial conditions often do not affect the steady-state values. By default, the initial conditions are zero.

However, for nonlinear ARX models, the steady-state output value might depend on initial conditions. For these models, you should investigate the effect of initial conditions on model response and use the values that produce the desired output. You can use `data2state` to map the input-output signal values from before the simulation starts to the model's initial states. Because the initial states are a function of the past history of the model's input and output values, `data2state` generates the initial states by transforming the data.

See Also

`idnlarx/linearize` | `idnlhw/linearize`

More About

- "Using Nonlinear ARX Models" on page 11-42
- "Using Hammerstein-Wiener Models" on page 12-9

Nonlinear Modeling of a Magneto-Rheological Fluid Damper

This example shows nonlinear black-box modeling of the dynamic behavior of a magneto-rheological fluid damper. It shows how to create Nonlinear ARX and Hammerstein-Wiener models of the damper using measurements of its velocity and the damping force.

The data used in this example was provided by Dr. Akira Sano (Keio University, Japan) and Dr. Jiandong Wang (Peking University, China) who performed the experiments in a laboratory of Keio University. See the following reference for a more detailed description of the experimental system and some related studies.

J.Wang, A. Sano, T. Chen, and B. Huang. Identification of Hammerstein systems without explicit parameterization of nonlinearity. International Journal of Control, In press, 2008. DOI: 10.1080/00207170802382376.

Experimental Setup

Magneto-Rheological (MR) fluid dampers are semi-active control devices used for reducing vibrations of various dynamic structures. MR fluids, whose viscosities depend on the input voltage/current of the device, provide controllable damping forces.

To study the behavior of such devices, a MR damper was fixed at one end to the ground and connected at the other end to a shaker table generating vibrations. The voltage of the damper was set to 1.25 v. The damping force f(t) was sampled every 0.005 s. The displacement was sampled every 0.001 s, which was then used to estimate the velocity v(t) at the sampling period of 0.005 s.

Input-Output Data

A data set containing the input and output measurements was stored in a MAT file called mrdamper.mat. The input v(t) is the velocity [cm/s] of the damper, and the output f(t) is the damping force [N]. The MAT file contains 3499 samples of data corresponding to a sampling rate of 0.005 s. This data will be used for all the estimation and validation tasks carried out in this example.

```
% Let us begin by loading and inspecting the data.
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'mrdamper.mat'));
who


Your variables are:

F   Ts   V
```

Package loaded variables F (output force), V (input velocity) and Ts (sample time) into an IDDATA object.

```
z = iddata(F, V, Ts, 'Name', 'MR damper', ...
    'InputName', 'v', 'OutputName', 'f',...
    'InputUnit', 'cm/s', 'OutputUnit', 'N');
```

Data Preparation for Estimation and Validation

Split this data set z into two subsets, first 2000 samples to be used for estimation (ze), and the rest to be used for validation of results (zv).

```
ze = z(1:2000);     % estimation data
zv = z(2001:end);   % validation data
```

Let us plot the two data sets together to visually verify their time ranges:

```
plot(ze, zv)
legend('ze','zv')
```
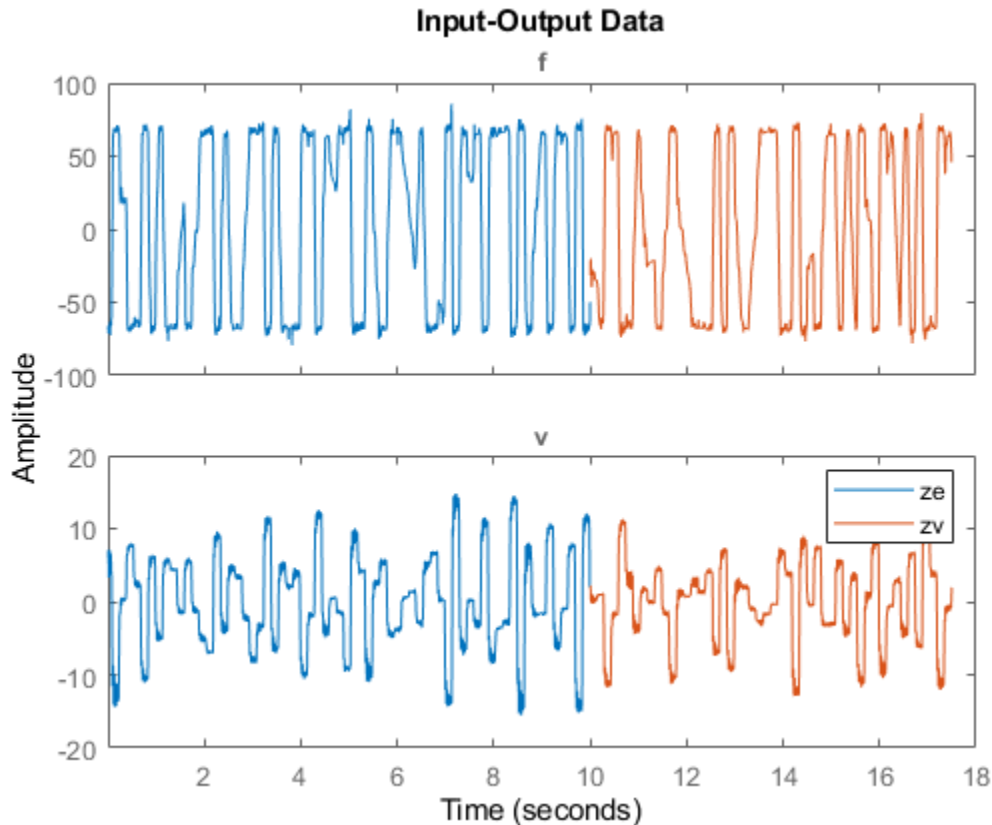

Model Order Selection

The first step in estimating black box models is to choose model orders. The definition of orders depends upon the type of model.

- For linear and nonlinear ARX models, the orders are represented by three numbers: na, nb and nk which define the number of past outputs, past inputs and input delays used for predicting the value of output at a given time. The set of time-delayed I/O variables defined by the orders are called "regressors". For more flexibility in creating regressors, you can use the regressor specification objects (see linearRegressor, polynomialRegressor, customRegressor).
- For Hammerstein-Wiener models, which represent linear models with static I/O nonlinearities, the orders define the number of poles and zeros and input delay of the linear component. They are defined by numbers nb (number of zeros +1), nf (number of poles), and nk (input delay in number of lags).

Typically model orders are chosen by trial and error. However, the orders of the linear ARX model may be computed automatically using functions such as `arxstruc` and `selstruc`. The orders thus obtained give a hint about the possible orders to use for the nonlinear models as well. So let us first try to determine the best order for a linear ARX model.

```
V = arxstruc(ze,zv,struc(1:5, 1:5,1:5)); % try values in the range 1:5 for na, nb, nk
Order = selstruc(V,'aic') % selection of orders by Akaike's Information Criterion


Order =

    2    4    1
```

The AIC criterion has selected Order = [na nb nk] = [2 4 1], meaning that in the selected ARX model structure, damper force f(t) is predicted by the 6 regressors f(t-1), f(t-2), v(t-1), v(t-2), v(t-3) and v(t-4).

For more information on model order selection, see the example titled "Model Structure Selection: Determining Model Order and Input Delay".

Preliminary Analysis: Creating Linear Models

It is advisable to try linear models first because they are simpler to create. If linear models do not provide satisfactory results, the results provide a basis for exploration of nonlinear models.

Let us estimate linear ARX and Output-Error (OE) models of orders suggested by output of SELSTRUC above.

```
LinMod1 = arx(ze, [2 4 1]); % ARX model Ay = Bu + e
LinMod2 = oe(ze, [4 2 1]);  % OE model  y = B/F u + e
```

Similarly, we may create a linear state-space model whose order (= number of states) will be determined automatically:

```
LinMod3 = ssest(ze); % creates a state-space model of order 3
```

Let us now compare the responses of these models to the measured output data in ze:

```
compare(ze, LinMod1, LinMod2, LinMod3) % comparison to estimation data
```
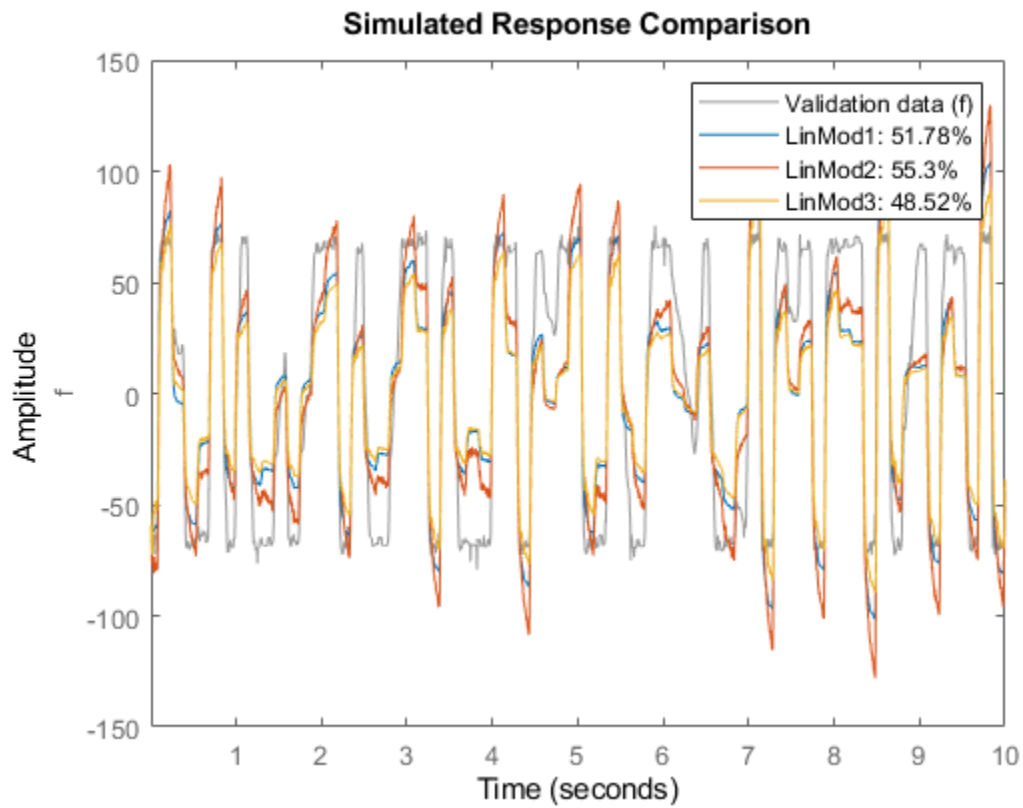
A better test for model quality is to validate it against an independent data set. Hence we compare the model responses against data set zv.

```
compare(zv, LinMod1, LinMod2, LinMod3) % comparison to validation data
```
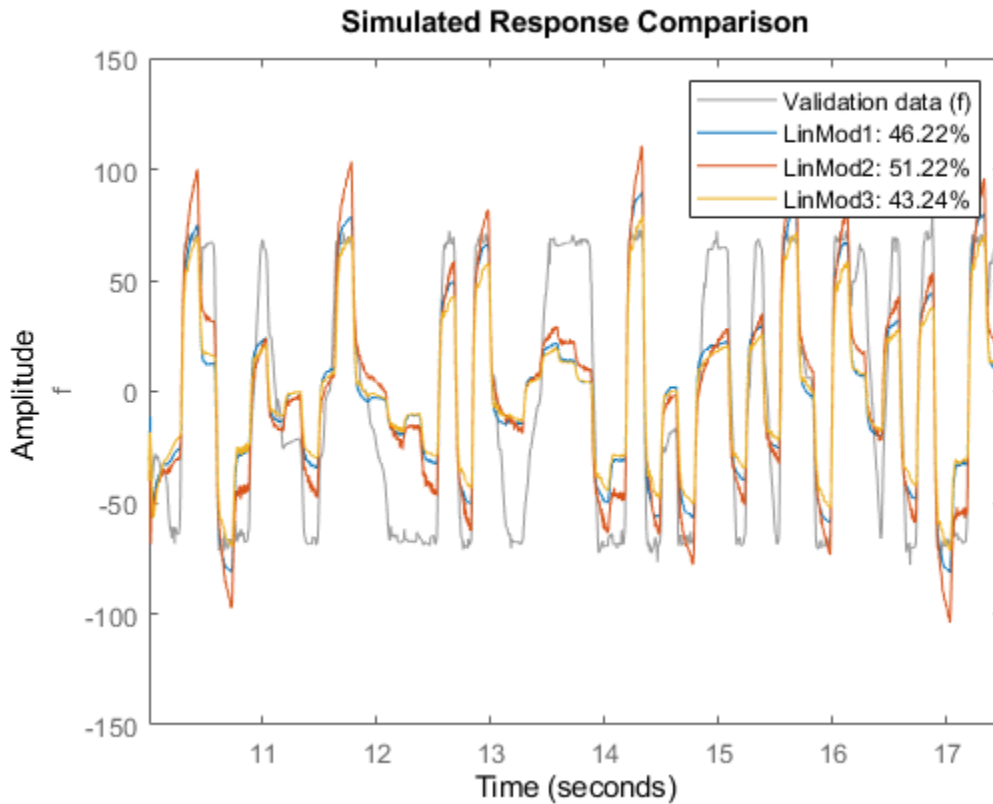
As observed, the best of these linear models has a fit of 51% on the validation data set.

Creating Nonlinear ARX Models

The linear model identification revealed that an ARX model provided less than 50% fit to the validation data. To achieve better results we now explore the use of Nonlinear ARX (IDNLARX) models. The need for nonlinear models for this data is also suggested by the `advice` utility which may be used to inspect data for potential advantage of using a nonlinear model over a linear model.

```
advice(ze, 'nonlinearity')
```

```
There is an indication of nonlinearity in the data.
A nonlinear ARX model of order [4 4 1] and idTreePartition function performs
better prediction of output than the corresponding ARX model of the same order.
Consider using nonlinear models, such as IDNLARX, or IDNLHW. You may also use
the "isnlarx" command to test for nonlinearity with more options.
```

The Nonlinear ARX models can be considered as nonlinear counterparts of ARX models that provide greater modeling flexibility by two means:

1   The regressors may be combined using a nonlinear function rather than a weighted sum employed by ARX models. Nonlinear functions such as sigmoid network, binary tree and wavelet network may be used. In the identification context, these functions are called "nonlinear mapping functions".

2   The regressors can themselves be arbitrary (possibly nonlinear) functions of I/O variables in addition to time-delayed variable values employed by ARX models.

Creating Regressors for Nonlinear ARX Models

When linear with contiguous lags, the regressors are most easily created using order matrix [na nb nk], as described above. In the most general case of regressors with arbitrary lags, or when the regressors are based on the absolute values of the variables, using the `linearRegressor` object provides more flexibility. In case the regressors are polynomials of time-delayed variables, they can be created using the `polynomialRegressor` object. For regressors employing arbitrary, user-specified formulas, `customRegressor` objects may be used.

We will explore using various model orders and using various nonlinear mapping functions. Use of polynomial or custom regressors is not explored. For ways of specifying custom regressors in IDNLARX models, see the example titled "Building Nonlinear ARX Models with Nonlinear and Custom Regressors".

Estimating a Default Nonlinear ARX Model

To begin, let us estimate an IDNLARX model of order [2 4 1] and a sigmoid network as the type of nonlinearity. We shall use MaxIterations = 50, and Levenberg-Marquardt search method as estimation options for all estimations below.

```
Options = nlarxOptions('SearchMethod', 'lm');
Options.SearchOptions.MaxIterations = 50;
Narx1 = nlarx(ze, [2 4 1], idSigmoidNetwork, Options)


Narx1 =

<strong>Nonlinear ARX model with 1 output and 1 input</strong>
  Inputs: v
  Outputs: f

Regressors:
  Linear regressors in variables f, v

Output function: Sigmoid network with 10 units
Sample time: 0.005 seconds

Status:
Termination condition: Maximum number of iterations reached..
Number of iterations: 50, Number of function evaluations: 393

Estimated using NLARX on time domain data "MR damper".
Fit to estimation data: 95.8% (prediction focus)
FPE: 6.648, MSE: 6.08
More information in model's "Report" property.
```

`nlarx` is the command used for estimating Nonlinear ARX models. Narx1 is a Nonlinear ARX model with regressors R := [f(t-1), f(t-2), v(t-1) ... v(t-4)]. The nonlinearity is a Sigmoid network that uses a combination of sigmoid unit functions and a linear weighted sum of regressors to compute the output. The mapping function is stored in the `OutputFcn` property of the model.

```
disp(Narx1.OutputFcn)

<strong>Sigmoid Network</strong>
Inputs: f(t-1), f(t-2), v(t-1), v(t-2), v(t-3), v(t-4)
Output: f(t)
```

```
Nonlinear Function: Sigmoid network with 10 units
Linear Function: initialized to [48.3 -3.38 -3.34 -2.7 -1.38 2.15]
Output Offset: initialized to -21.4

        Inputs: {'f(t-1)'  'f(t-2)'  'v(t-1)'  'v(t-2)'  'v(t-3)'  'v(t-4)'}
       Outputs: {'f(t)'}
  NonlinearFcn: '<Sigmoid units and their parameters>'
     LinearFcn: '<Linear function parameters>'
        Offset: '<Offset parameters>'
```

Examine the model quality by comparing the simulated output against the estimated and validated data sets ze and zv:

```
compare(ze, Narx1); % comparison to estimation data
```
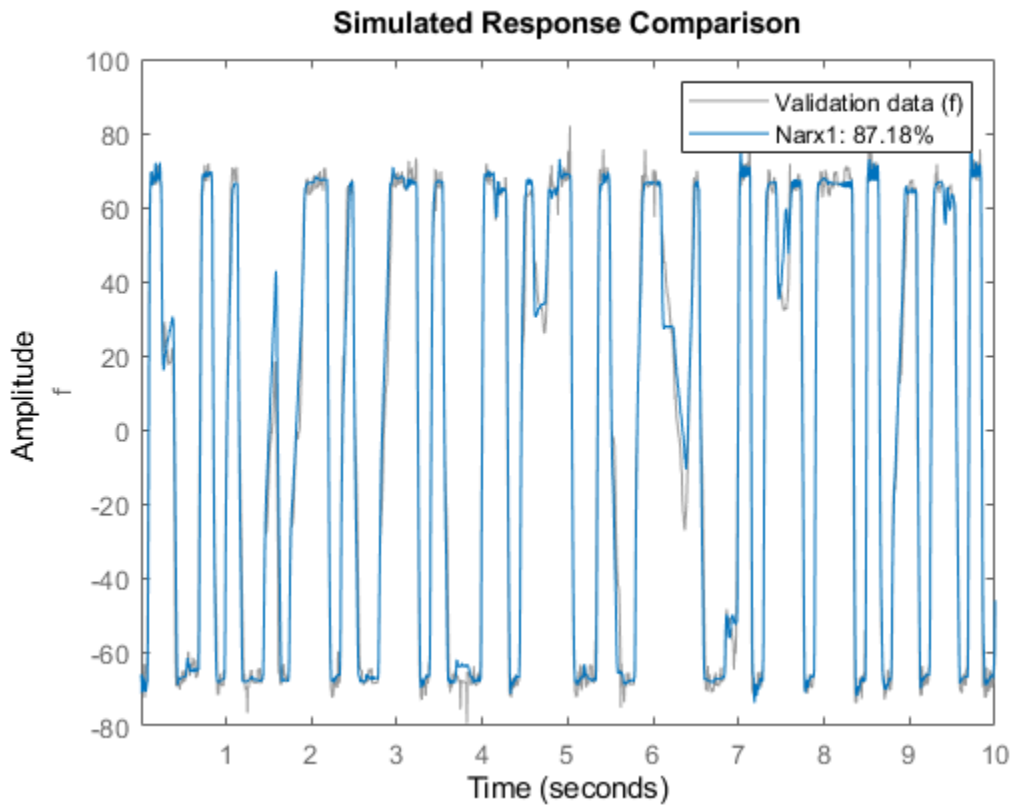


```
compare(zv, Narx1); % comparison to validation data
```

Simulated Response Comparison
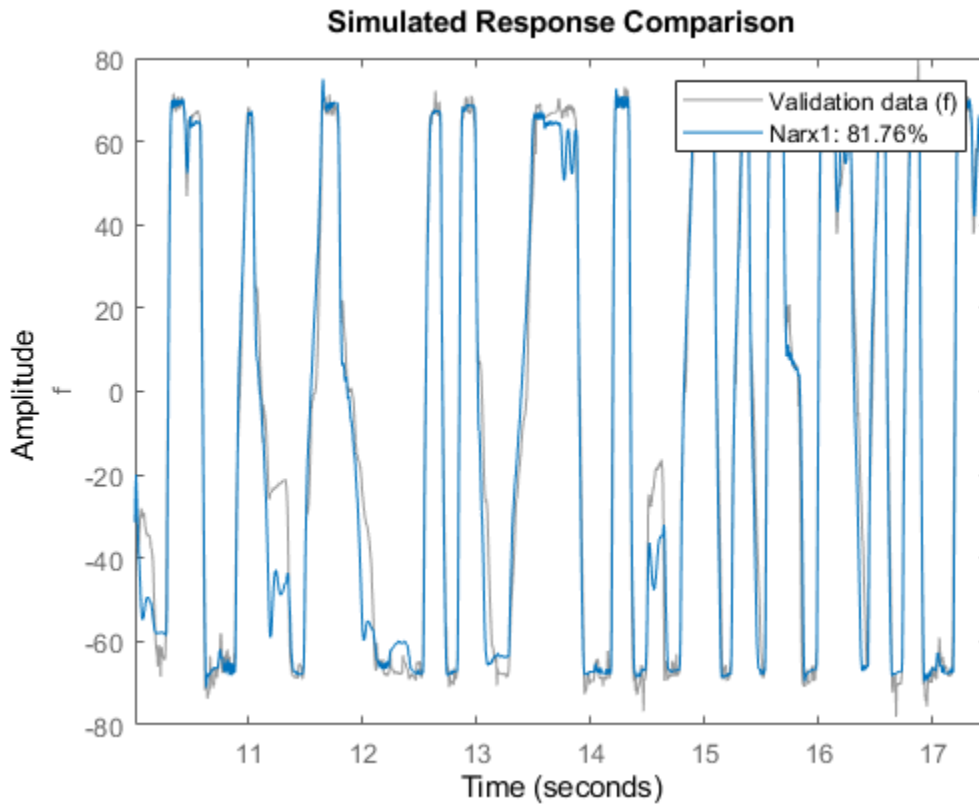
Trying Various Model Orders

We see a better fit as compared to the linear models of same orders. Next, we can try other orders in the vicinity of those suggested by SELSTRUC.

```
NL = idSigmoidNetwork;
Narx2{1} = nlarx(ze, [3 4 1], NL, Options); % use na = 3, nb = 4, nk = 1.
Narx2{1}.Name = 'Narx2_1';
Narx2{2} = nlarx(ze, [2 5 1], NL, Options); Narx2{2}.Name = 'Narx2_2';
Narx2{3} = nlarx(ze, [3 5 1], NL, Options); Narx2{3}.Name = 'Narx2_3';
Narx2{4} = nlarx(ze, [1 4 1], NL, Options); Narx2{4}.Name = 'Narx2_4';
Narx2{5} = nlarx(ze, [2 3 1], NL, Options); Narx2{5}.Name = 'Narx2_5';
Narx2{6} = nlarx(ze, [1 3 1], NL, Options); Narx2{6}.Name = 'Narx2_6';
```

Evaluate the performance of these models on estimation and validation data sets:

```
compare(ze, Narx1, Narx2{:}); % comparison to estimation data
```

Simulated Response Comparison
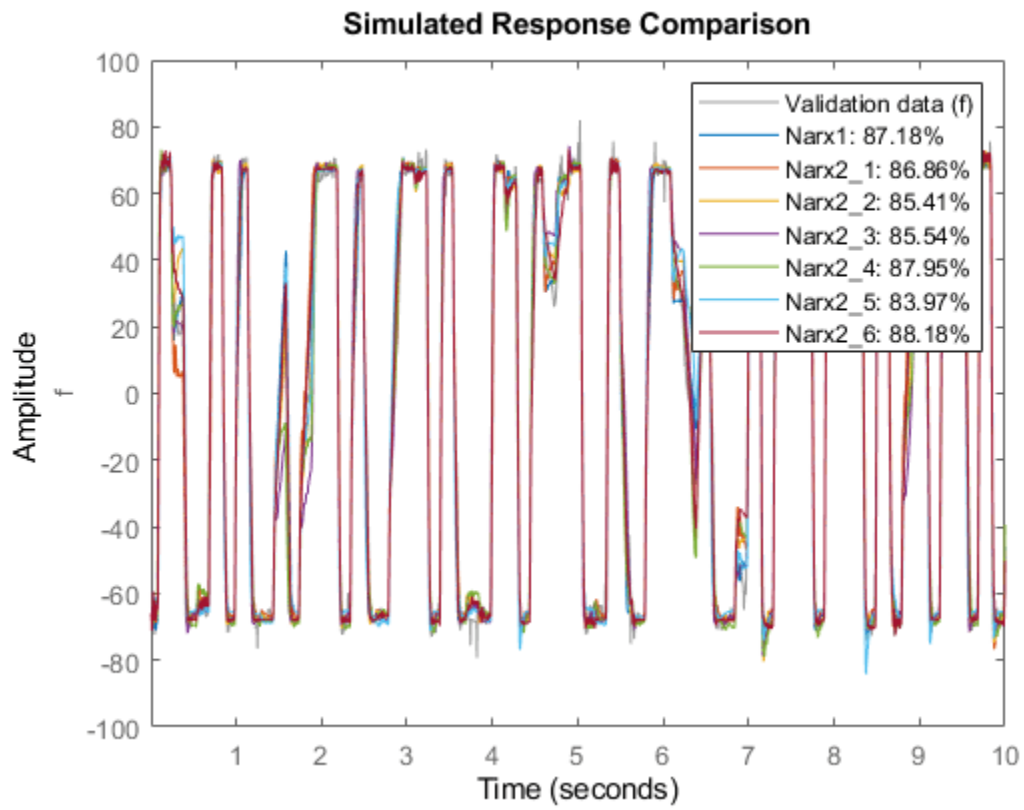
```
compare(zv, Narx1, Narx2{:}); % comparison to validation data
```

Simulated Response Comparison
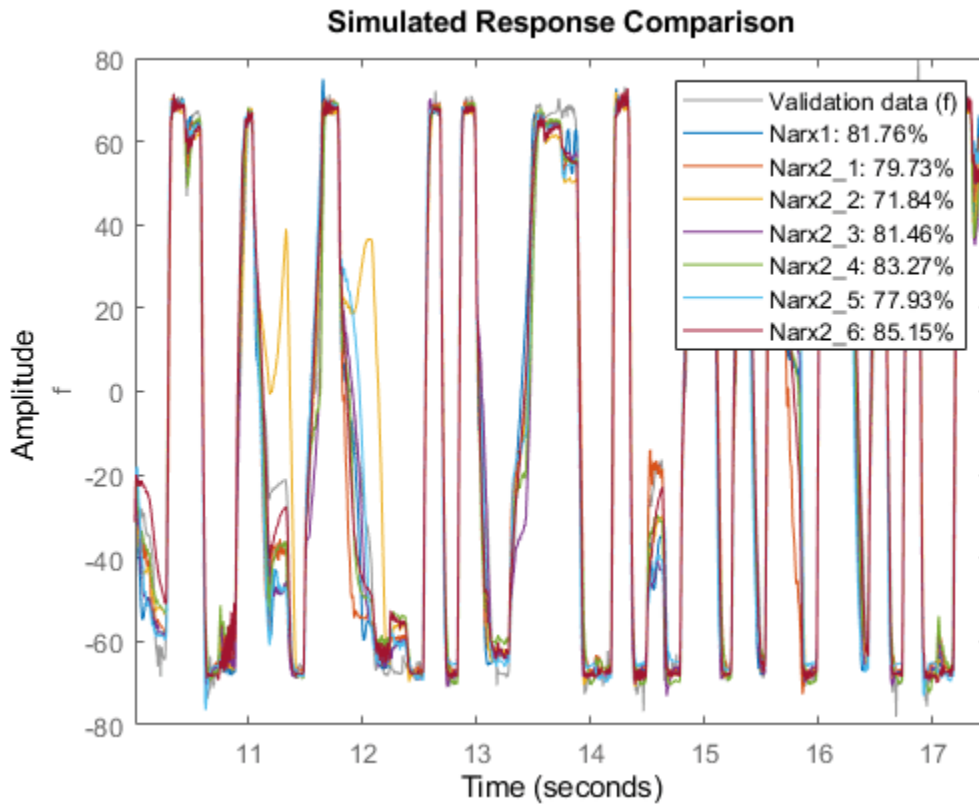
The model Narx2{6} seems to be providing good fits to both estimation and validation data sets while its orders are smaller than those of Narx1. Based on this observation, let us use [1 3 1] as orders for subsequent trials, and retain Nlarx2{6} for fit comparisons. This choice of orders corresponds to using [f(t-1), v(t-1), v(t-2), v(t-3)] as the set of regressors.

Specifying Number of Units for Sigmoid Network Function

Next, let us explore the structure of the Sigmoid Network function. The most relevant property of this estimator is the number of sigmoid units it uses. Create one that uses 12 sigmoid units.

```
Sig = idSigmoidNetwork(12); % create a network that uses 12 units
Narx3 = nlarx(ze, [1 3 1], Sig, Options);
```

We compare this model against Narx1 and Narx2{6}, on estimation and validation data sets:

```
compare(ze, Narx3, Narx1, Narx2{6}); % comparison to estimation data
```
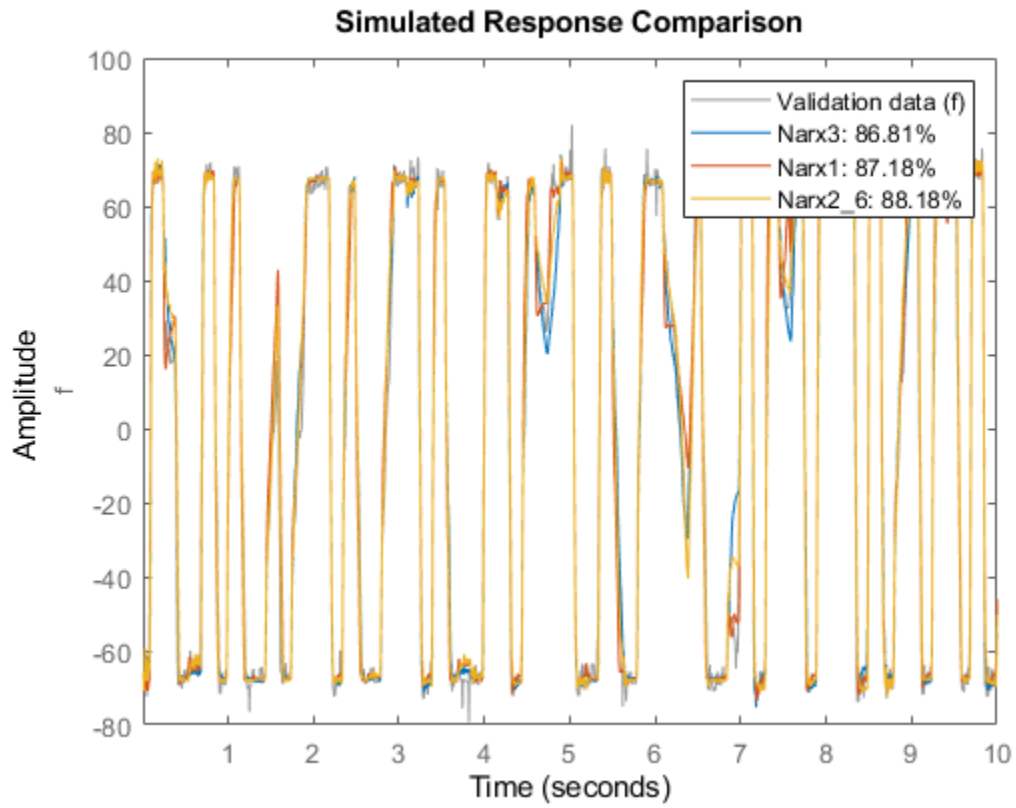
```
compare(zv, Narx3, Narx1, Narx2{6}); % comparison to validation data
```

Simulated Response Comparison
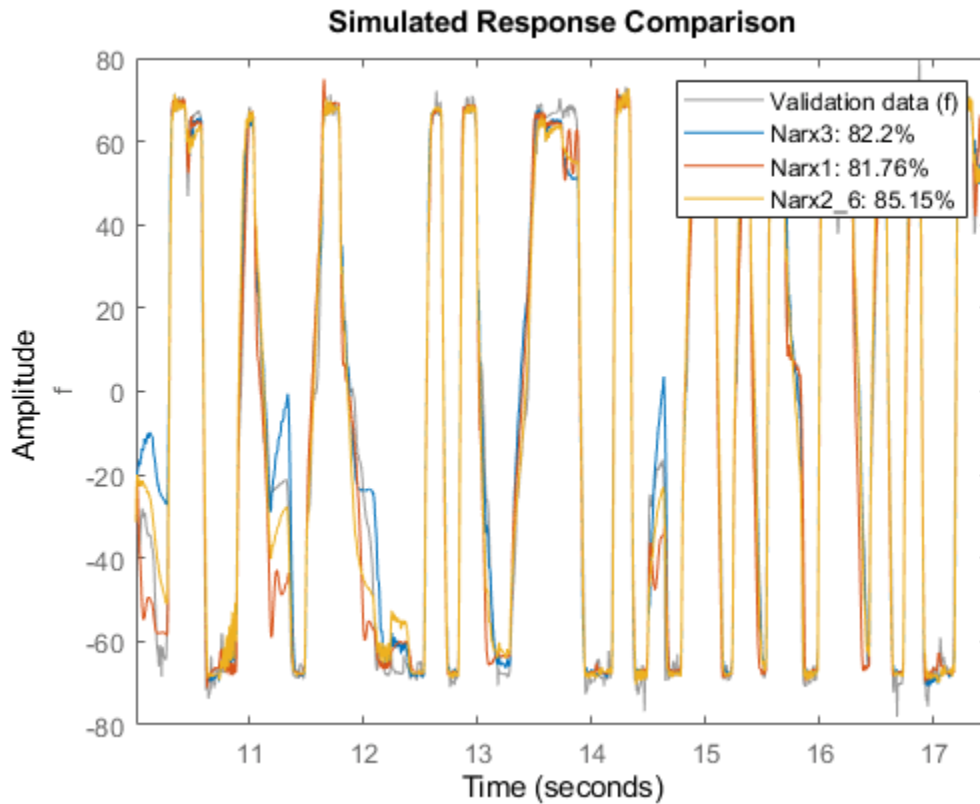
The new model Narx3 provides no better fit than Narx2{6}. Hence we retain the number of units to 10 in subsequent trials.

Choosing Regressor Subset for Nonlinear Mapping Function

Typically, all the regressors defined by chosen orders ([1 3 1] here) are used by the nonlinear function (sigmoid network). If the number of regressors is large, this may increase the model complexity. It is possible to select a subset of regressors to be used by the components of the sigmoid network without modifying the model orders. This is facilitated by the property called 'RegressorUsage'. Its value is a table that specifies which regressor are used by which components. For example, we may explore using only those regressors that are contributed by the input variables to be used by the nonlinear component of the sigmoid function. This can be done as follows:

```
Sig = idSigmoidNetwork(10);
NarxInit = idnlarx(ze.OutputName, ze.InputName, [1 3 1], Sig);
NarxInit.RegressorUsage.("f:NonlinearFcn")(1) = false;
disp(NarxInit.RegressorUsage)
Narx4 = nlarx(ze, NarxInit, Options);
```

	f:LinearFcn	f:NonlinearFcn
f(t-1)	true	false
v(t-1)	true	true
v(t-2)	true	true
v(t-3)	true	true

This causes the regressors v(t-1), v(t-2), and v(t-3) to be used by the sigmoid unit functions. The output variable based regressor f(t-1) is not used. Note that the idSigmoidNetwork function also contains a linear term represented by a weighted sum of all regressors. The linear term uses the full set of regressors.

Create another model that uses regressors {y1(t-1), u1(t-2), u1(t-3)} for its nonlinear component.

```
Use = false(4,1);
Use([1 3 4]) = true;
NarxInit.RegressorUsage{:,2} = Use;
Narx5 = nlarx(ze, NarxInit, Options);
```

The model Narx5 seems to perform very well for both estimation and validation data sets.

```
compare(ze, Narx5); % comparison to estimation data
```
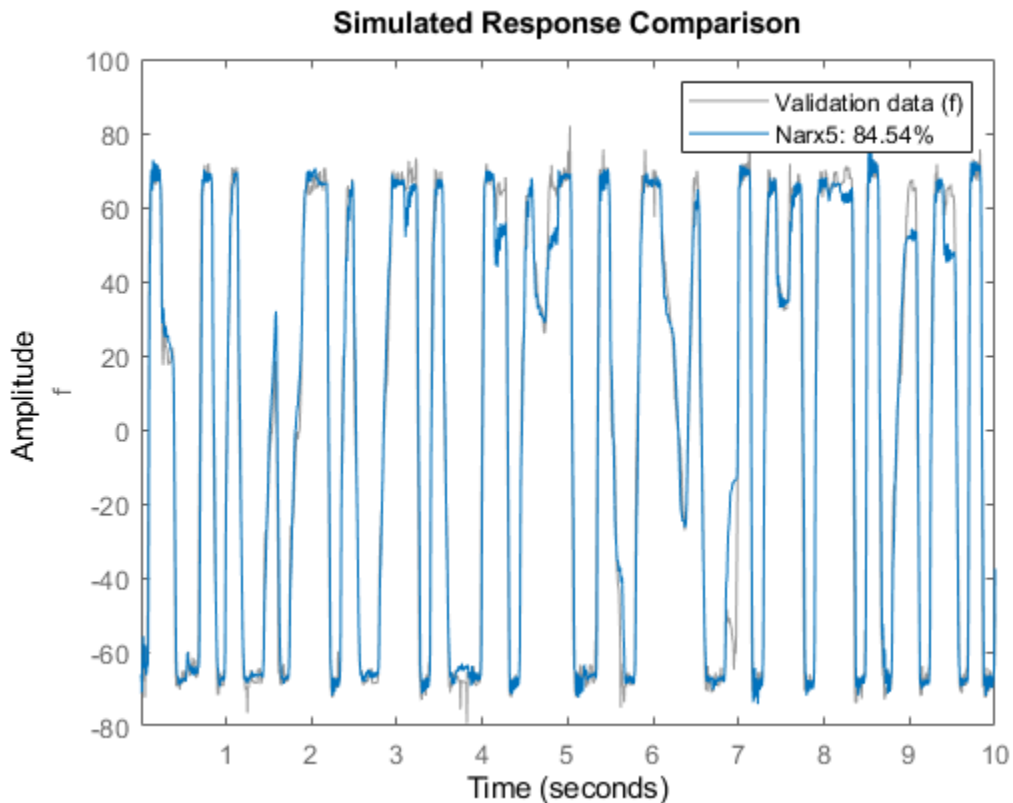


```
compare(zv, Narx5); % comparison to validation data
```

Simulated Response Comparison
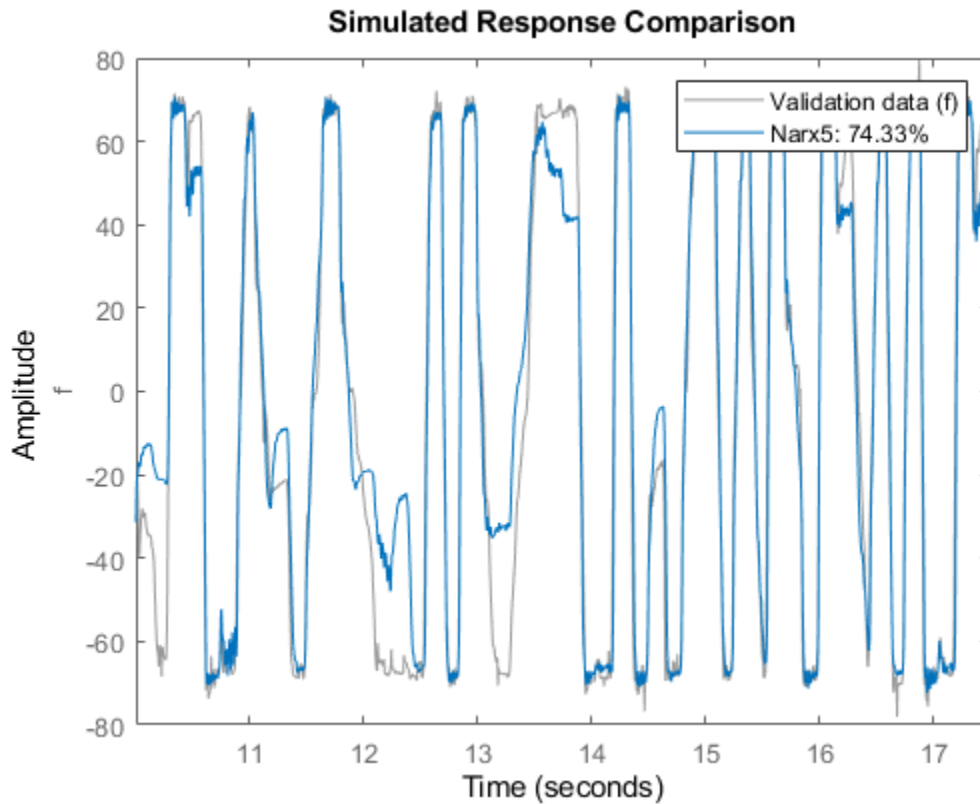
Trying Various Nonlinear Mapping Functions

So far we have explored various model orders, number of units to be used in the Sigmoid Network function and specification of a subset of regressors to be used by the nonlinear component of the sigmoid network. Next, we try using other types of nonlinear functions.

Use a Wavelet Network function with default properties. Like the sigmoid network, a wavelet network maps the regressors to the output by using a sum of linear and nonlinear components; the nonlinear component uses a sum of wavelets.

```
NarxInit = idnlarx(ze.OutputName, ze.InputName, [1 3 1], idWaveletNetwork);
% Use only regressors 1 and 3 for the nonlinear component of network
NarxInit.RegressorUsage.("f:NonlinearFcn")([2 4]) = false;
Narx6 = nlarx(ze, NarxInit, Options);
```

Use a Tree Partition nonlinear function with 20 units:

```
TreeNet = idTreePartition;
TreeNet.NonlinearFcn.NumberOfUnits = 20;
NarxInit.OutputFcn = TreeNet;
Narx7 = nlarx(ze, NarxInit, Options);
```

Compare the results against Narx3 and Narx5

```
compare(ze, Narx3, Narx5, Narx6, Narx7) % comparison to estimation data
```
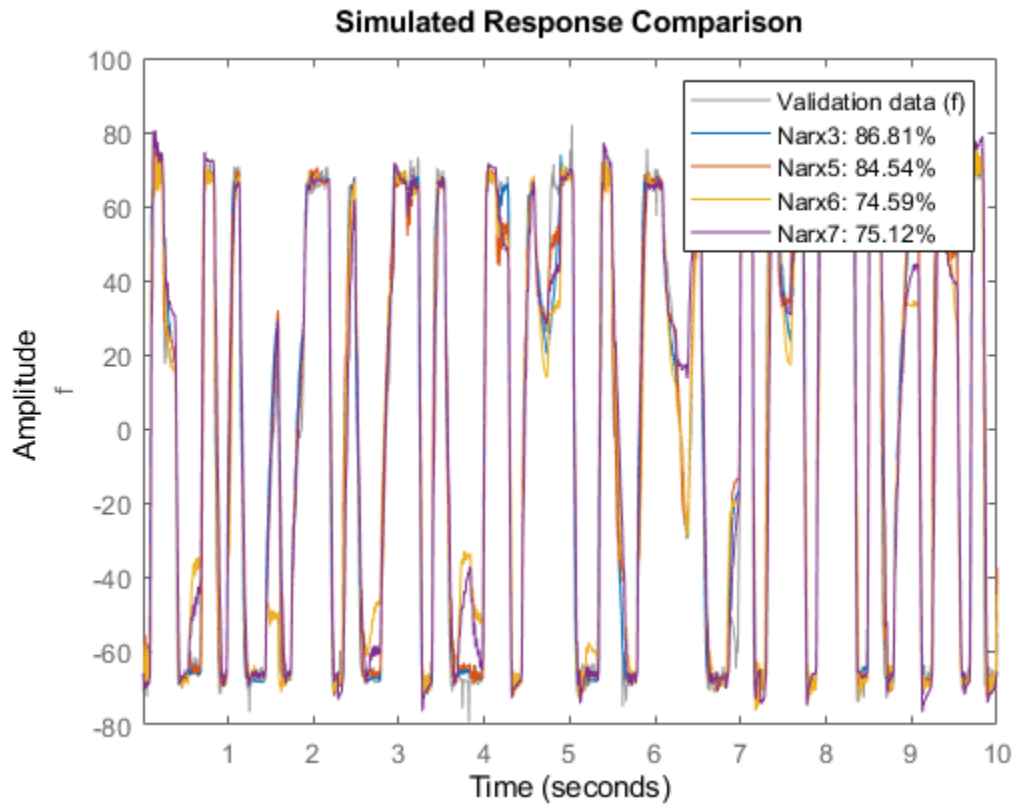
```
compare(zv, Narx3, Narx5, Narx6, Narx7) % comparison to validation data
```
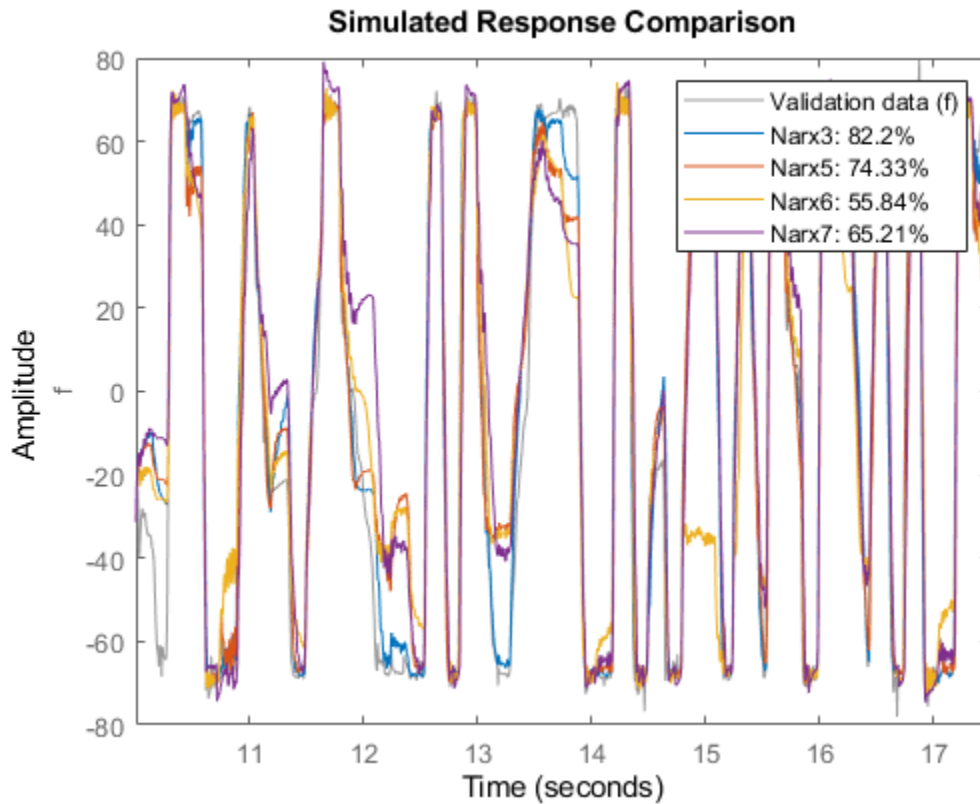
The models Narx6 and Narx7 seem to perform worse than Narx5, although we have not explored all the options associated with their estimation (such as choice of nonlinear regressors, number of units and other model orders).

Analyzing Estimated IDNLARX Models

Once a model has been identified and validated using the `compare` command, we may tentatively select the one that provides the best results without adding too much extra complexity. The selected model may then be analyzed further using command such as `plot` and `resid`.

To get some insight into the nature of the nonlinearities of the model, inspect cross-sections of the nonlinear function F() in the estimated model f(t) = F(f(t-1), f(t-2), v(t-1),...,v(t-4)). For example, in model Narx5, the function F() is a sigmoid network. To explore the shape of the output of F() as a function of the regressors, use the PLOT command on the model:

```
plot(Narx5)
```
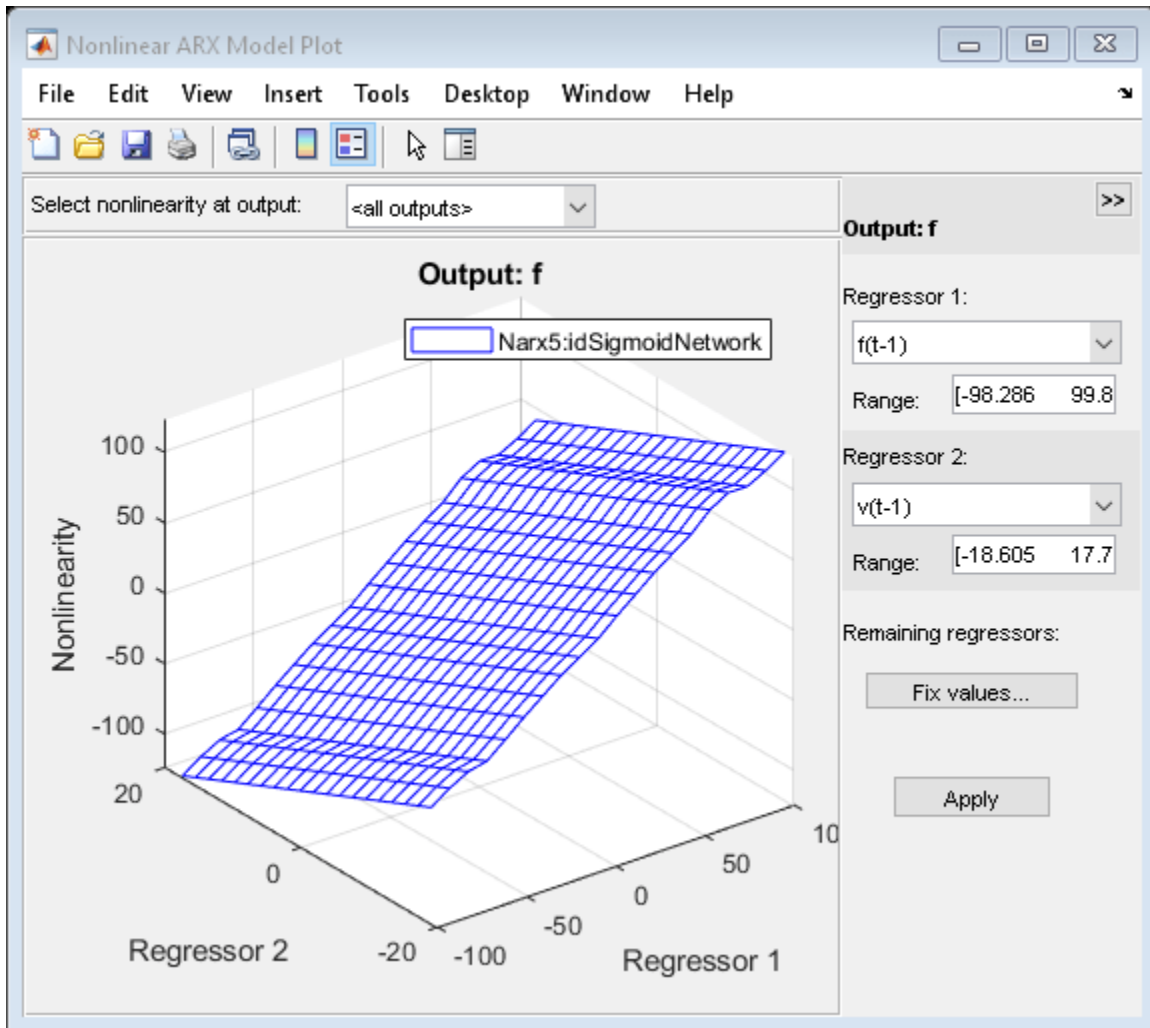
The plot window offers tools for selecting the cross-section regressors and their ranges. For more information, see "help idnlarx/plot".

The residual test can be used to further inspect the model. This test reveals if the prediction errors are white and uncorrelated to the input data.

```
set(gcf,'DefaultAxesTitleFontSizeMultiplier',1,...
    'DefaultAxesTitleFontWeight','normal',...
    'Position',[100 100 780 520]);
resid(zv, Narx3, Narx5)
```

Residue Correlation
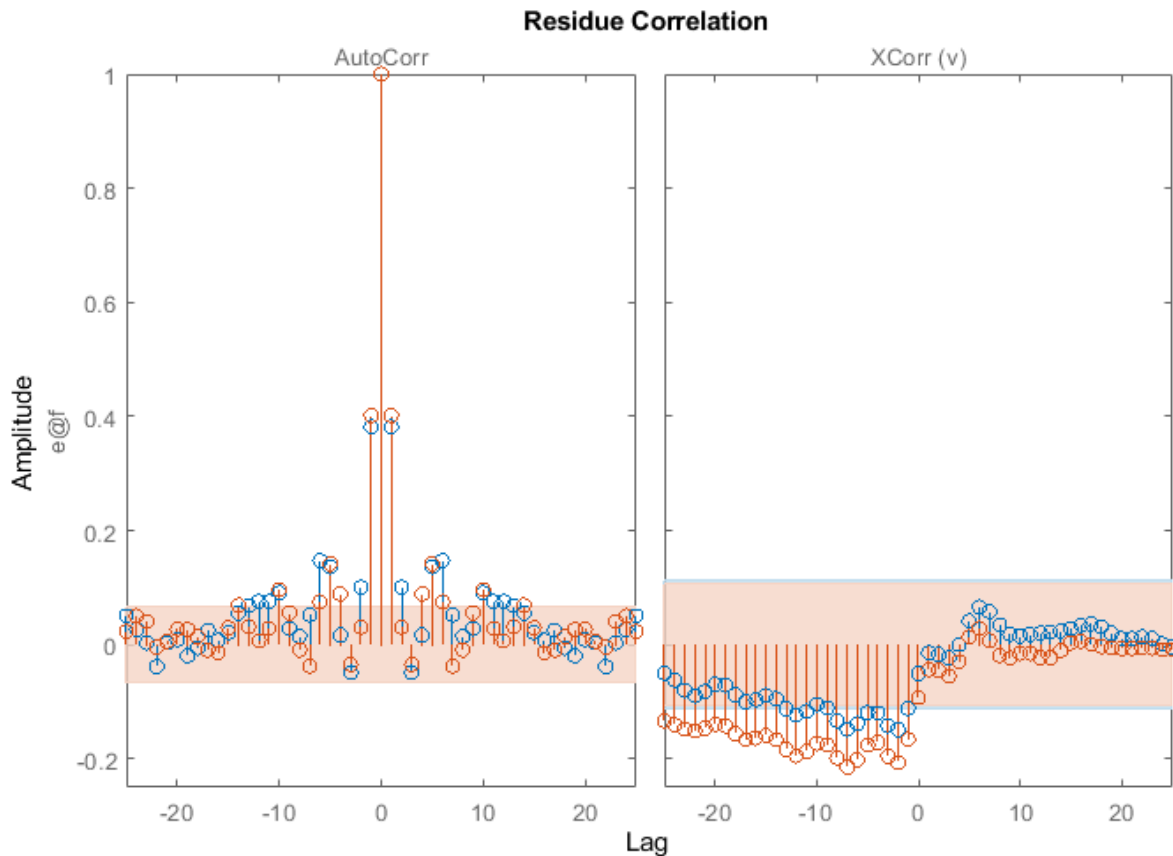
A failure of residual test may point to dynamics not captured by the model. For the model, Narx3, the residuals appear to be mostly within the 99% confidence bounds.

Creating Hammerstein-Wiener Models

The previously estimated nonlinear models are all of Nonlinear ARX (IDNLARX) type. Let us now try Hammerstein-Wiener (IDNLHW) models. These models represent a series connection of static nonlinear elements with a linear model. We may think of them as extensions of a linear output-error (OE) models wherein we subject the input and output signals of the linear model to static nonlinearities such as saturation or dead-zones.

Estimating an IDNLHW Model of Same Order as the Linear OE Model

The linear OE model LinMod2 was estimated using orders nb = 4, nf = 2 and nk = 1. Let us use the same orders for estimation of an IDNLHW model. We will use sigmoid network as nonlinearity for both input and output nonlinearities. The estimation is facilitated by the `nlhw` command. It is analogous to the `oe` command used for linear OE model estimation. However, in addition to model orders, we must also specify the names of, or objects for, the I/O nonlinearities.

```
Opt = nlhwOptions('SearchMethod','lm');
UNL = idSigmoidNetwork;
YNL = idSigmoidNetwork;
Nhw1 = nlhw(ze, [4 2 1], UNL, YNL, Opt)
```

```
Nhw1 =

<strong>Hammerstein-Wiener model with 1 output and 1 input</strong>

Linear transfer function corresponding to the orders nb = 4, nf = 2, nk = 1

Input nonlinearity: Sigmoid network with 10 units
Output nonlinearity: Sigmoid network with 10 units
Sample time: 0.005 seconds

Status:
Termination condition: Maximum number of iterations reached..
Number of iterations: 20, Number of function evaluations: 89

Estimated using NLHW on time domain data "MR damper".
Fit to estimation data: 83.98%
FPE: 94.66, MSE: 88.34
More information in model's "Report" property.
```

Compare the response of this model against estimation and validation data sets:

```
clf
compare(ze, Nhw1); % comparison to estimation data
```
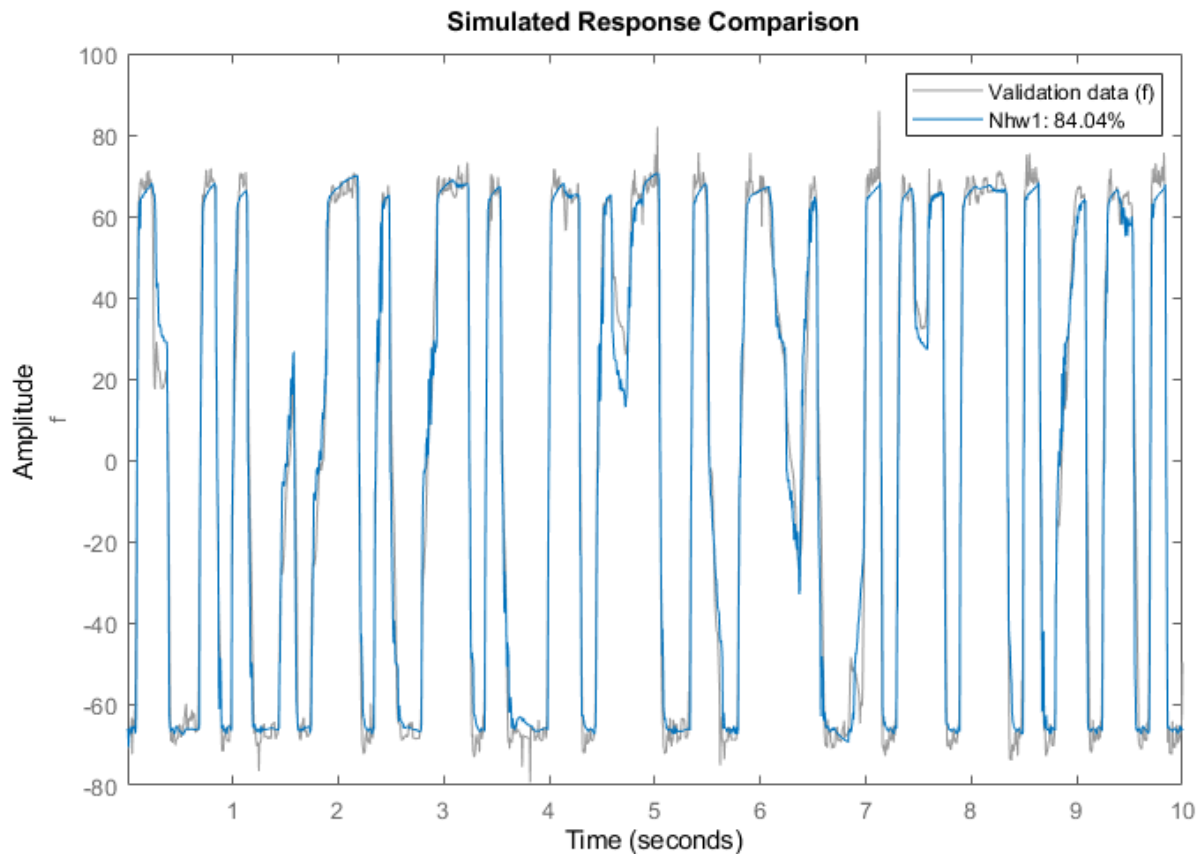
```
compare(zv, Nhw1); % comparison to validation data
```


Simulated Response Comparison
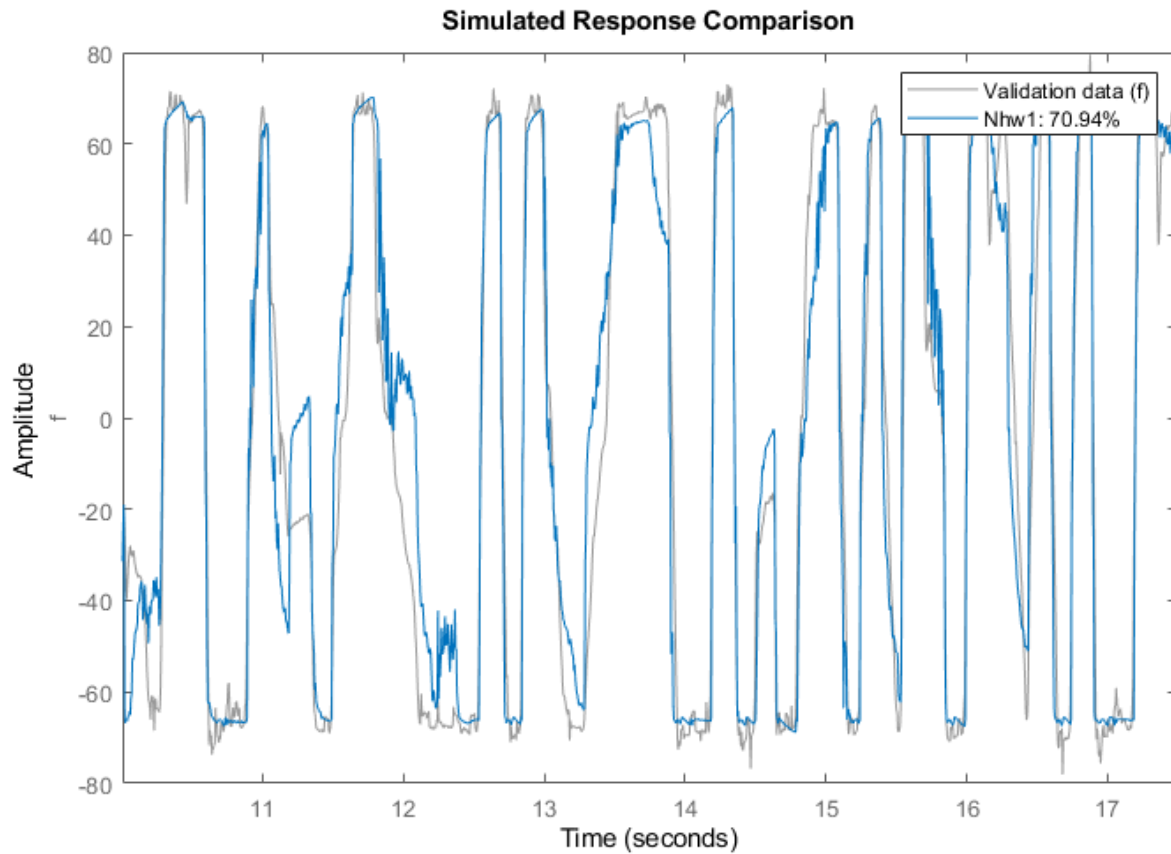
We observe about 70% fit to validation data using model Nhw1.

Analyzing Estimated IDNLHW Model

As for Nonlinear ARX models, the Hammerstein-Wiener models may be inspected for their nature of the I/O nonlinearities and the behavior of the linear component using the PLOT command. For more information, type "help idnlhw/plot" in MATLAB Command Window.

```
plot(Nhw1)
```
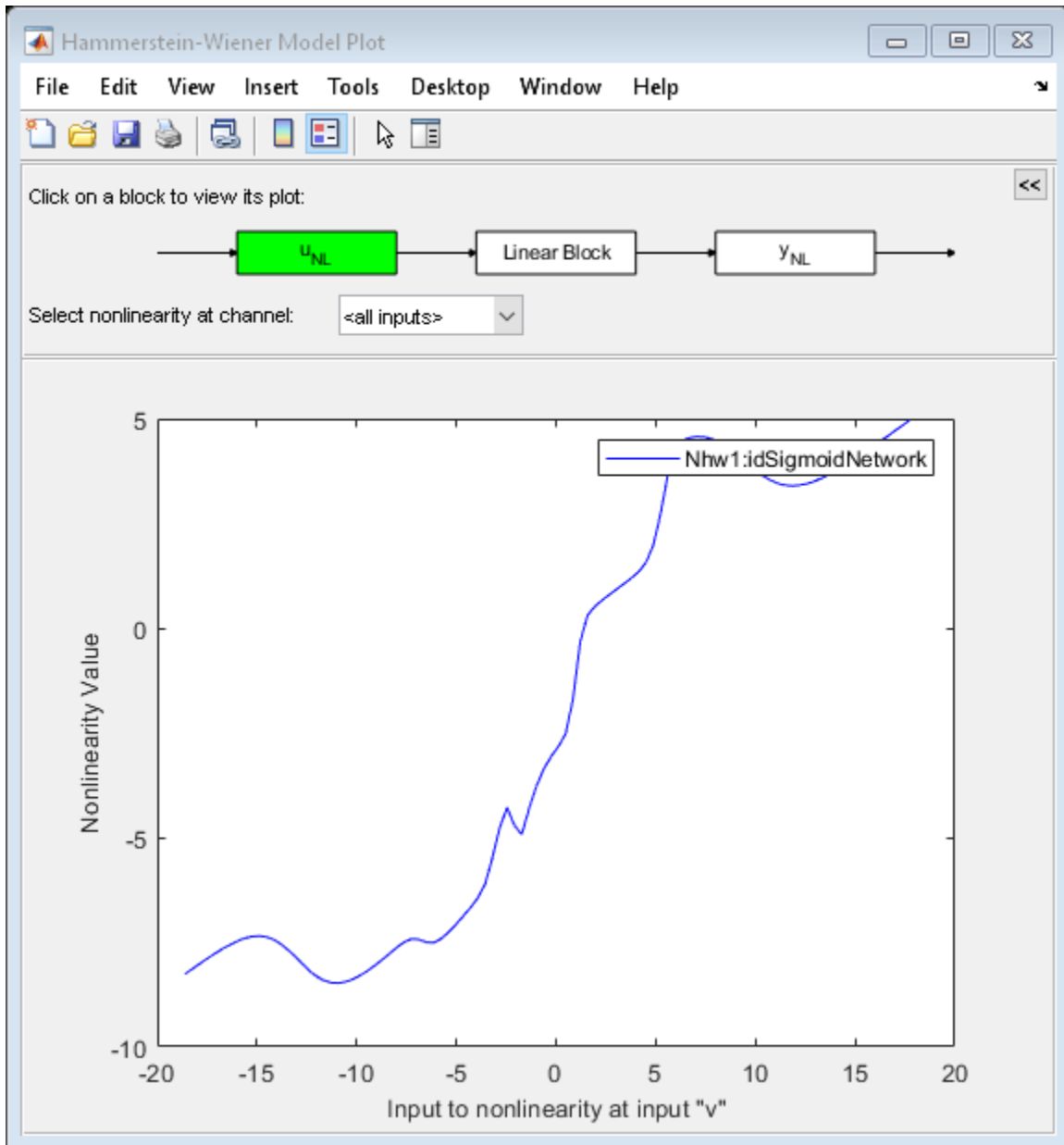
By default the input nonlinearity is plotted, showing that it may be simply a saturation function.

By clicking on the Y_NL icon, the output nonlinearity looks like a piecewise linear function.
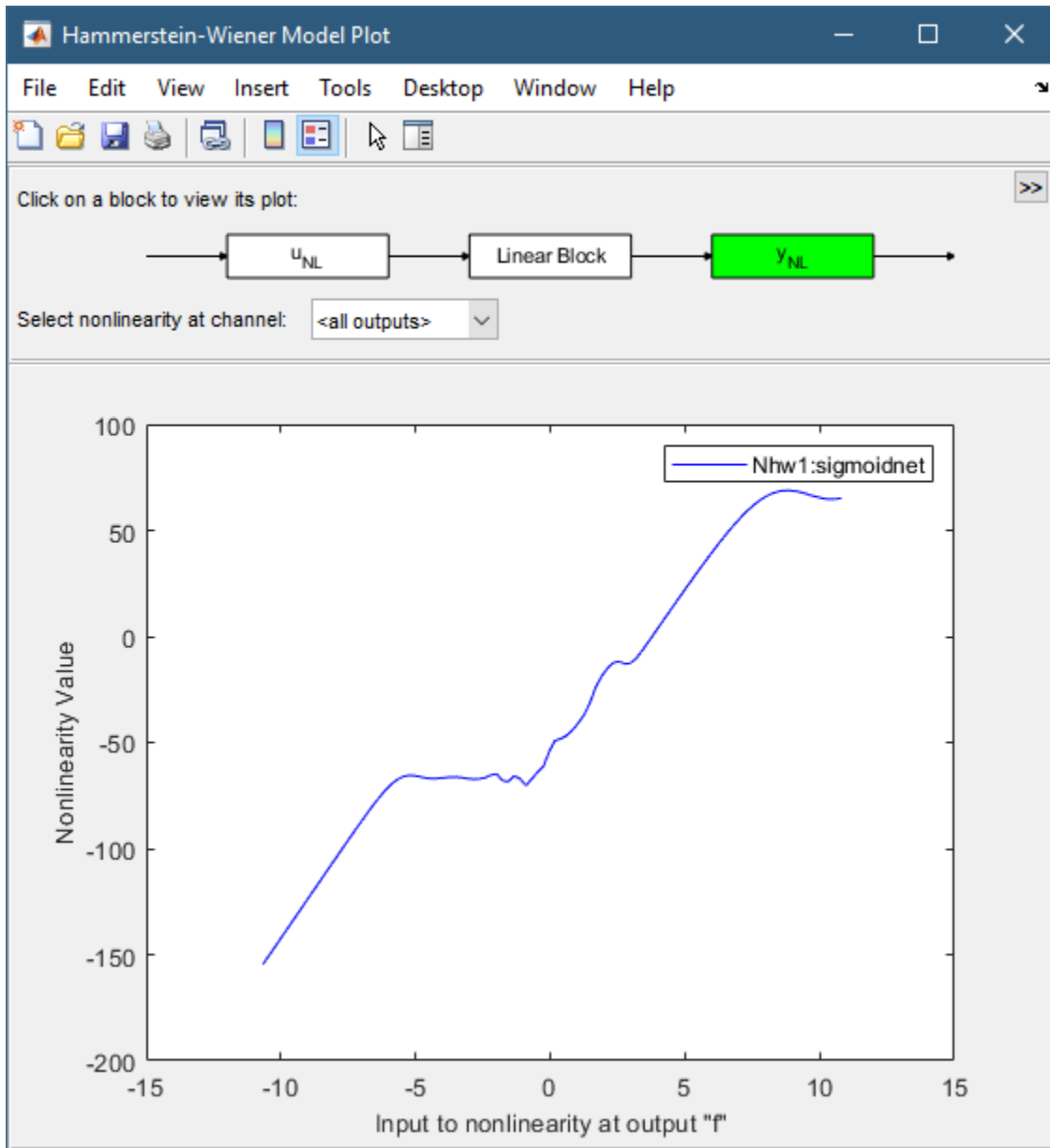
Click on the Linear Block icon and choose Pole-Zero Map in the pull-down menu, it is then observed that a zero and a pole are quite close to each other, indicating that they may be removed, thereby reducing the model orders.
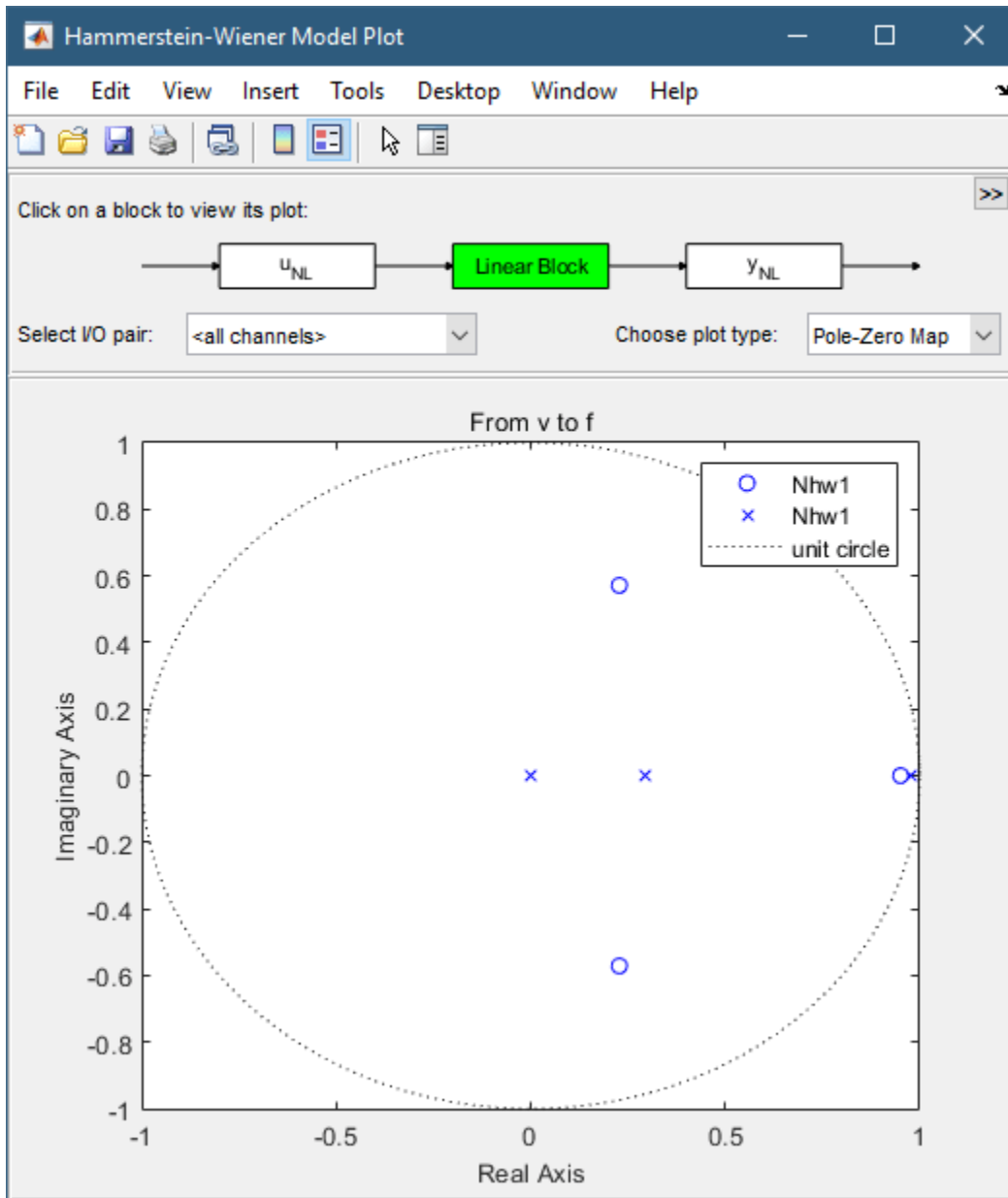
We will use this information to configure the structure of the model as shown next.

Trying Various Nonlinear Functions and Model Orders

Use saturation for input nonlinearity and sigmoid network for output nonlinearity, keeping the order of the linear component unchanged:

```
Nhw2 = nlhw(ze, [4 2 1], idSaturation, idSigmoidNetwork, Opt);
```

Use piecewise-linear nonlinearity for output and sigmoid network for input:

```
Nhw3 = nlhw(ze, [4 2 1], idSigmoidNetwork, idPiecewiseLinear, Opt);
```

Use a lower order linear model:

11-75

```
Nhw4 = nlhw(ze, [3 1 1], idSigmoidNetwork, idPiecewiseLinear, Opt);
```

We may also choose to "remove" nonlinearity at the input, output or both. For example, in order to use only an input nonlinearity (such models are called Hammerstein models), we may specify [] as output nonlinearity:

```
Nhw5 = nlhw(ze, [3 1 1],idSigmoidNetwork, [], Opt);
```

Compare all models

```
compare(ze, Nhw1, Nhw2, Nhw3, Nhw4, Nhw5) % comparison to estimation data
```
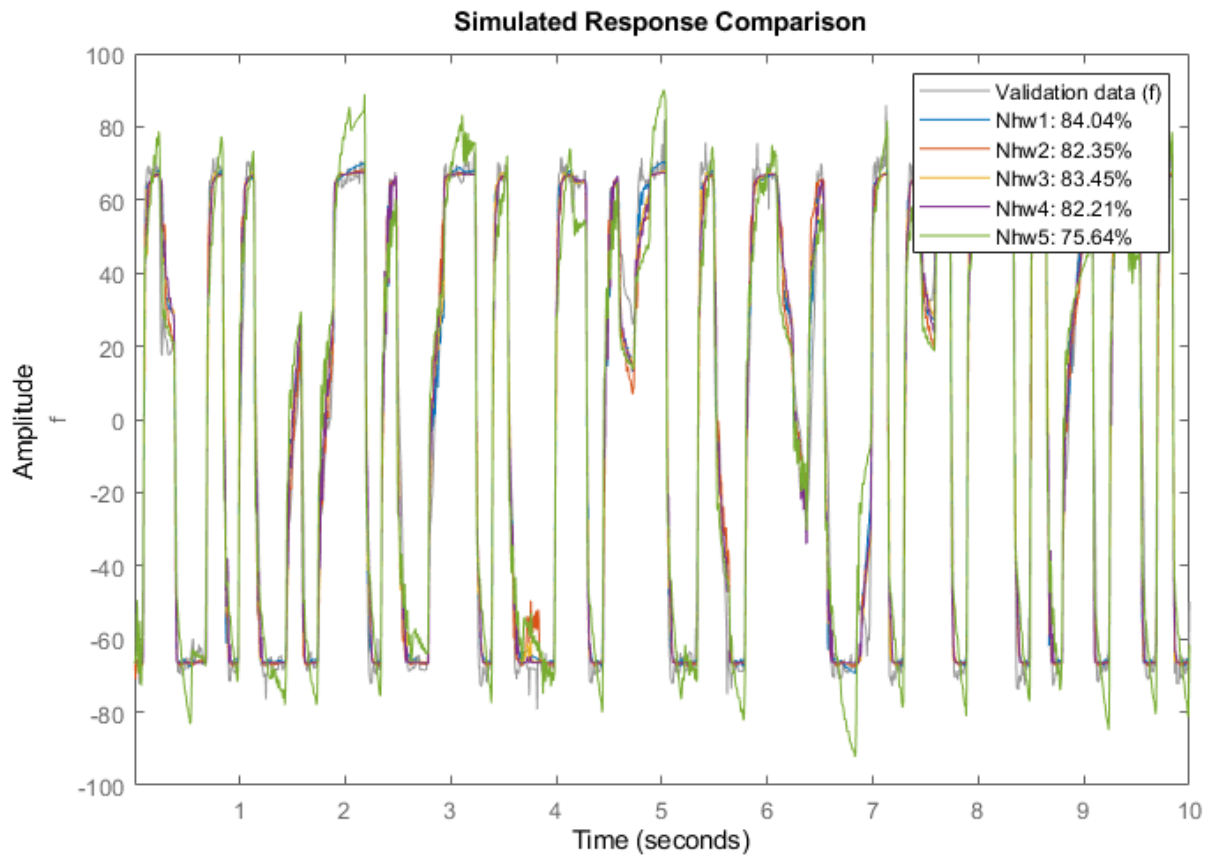


```
compare(zv, Nhw1, Nhw2, Nhw3, Nhw4, Nhw5) % comparison to validation data
```
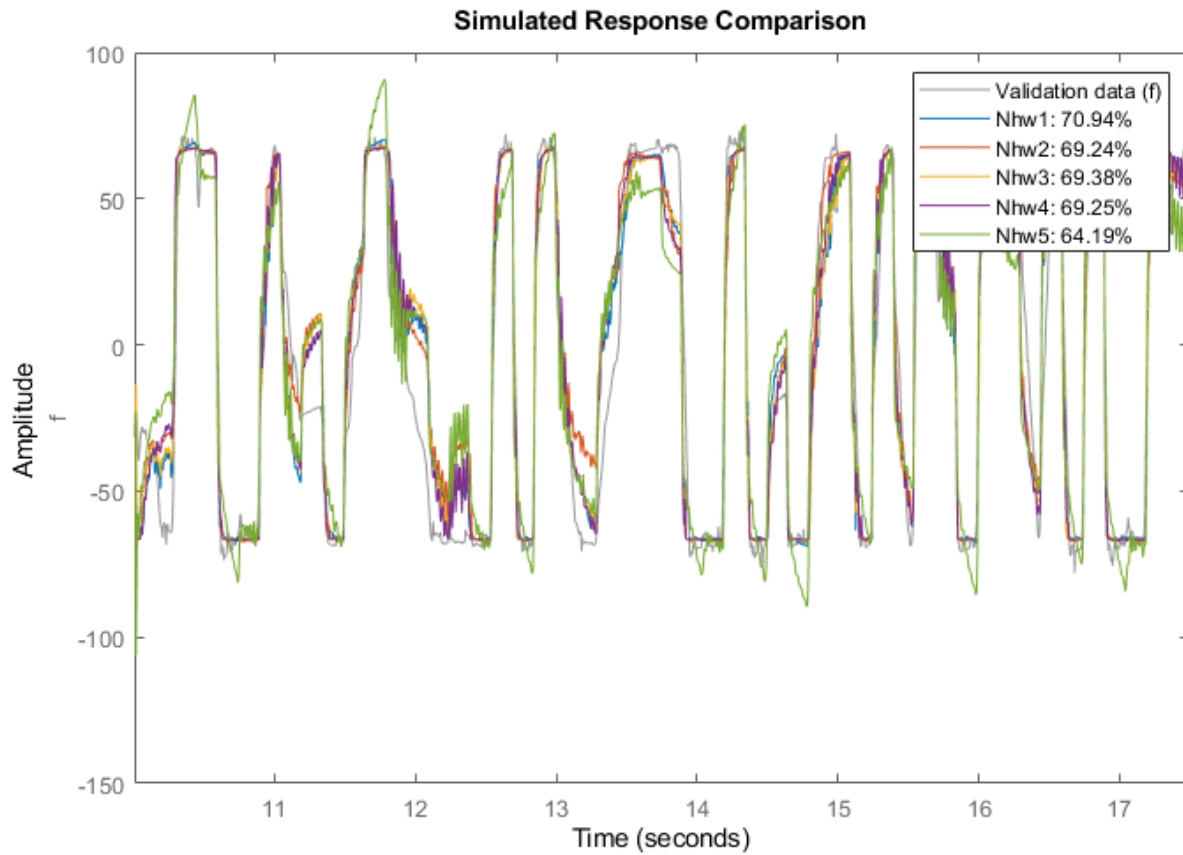
Nhw1 remains the best among all models, as shown by the comparison to validation data, but the other models, except Nhw5, have similar performance.

Conclusions

We explored various nonlinear models for describing the relationship between the voltage input and the damping force output. It was seen that among the Nonlinear ARX models, Narx2{6}, and Narx5 performed the best, while the model Nhw1 was the best among the Hammerstein-Wiener models. We also found that the Nonlinear ARX models provided the best option (best fits) for describing the dynamics of the MR damper.

```
Narx5.Name = 'Narx5';
Nhw1.Name = 'Nhw1';
compare(zv, LinMod2, Narx2{6}, Narx5, Nhw1)
```
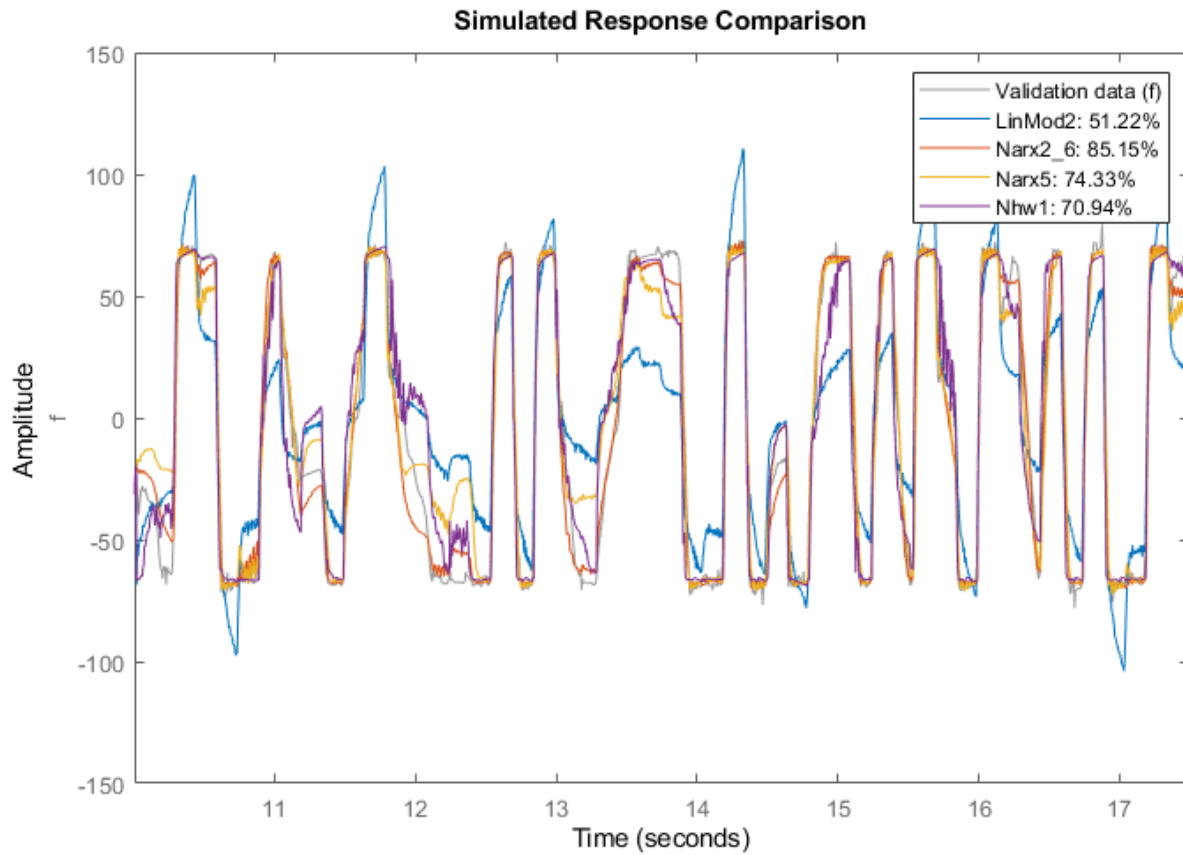
We found that there are multiple options available with each model type to fine-tune the quality of results. For Nonlinear ARX models, we can not only specify the model orders and the type of nonlinear functions, but also configure how the regressors are used and tweak the properties of the chosen functions. For Hammerstein-Wiener models, we can choose the type of input and output nonlinear functions, as well as the order of the linear component. For both model types, we have many choices of nonlinear functions available at our disposal to try and use. In lack of a particular preference for model structure or knowledge of the underlying dynamics, it is recommended to try the various choices and analyze their effect on the quality of the resulting models.

A Tutorial on Identification of Nonlinear ARX and Hammerstein-Wiener Models

This example shows identification of single-input-single-output (SISO) nonlinear black box models using measured input-output data. The example uses measured data from a two-tank system to explore various model structures and identification choices.

The Input-Output Data Set

In this example, the data variables stored in the `twotankdata.mat` file will be used. It contains 3000 input-output data samples of a two-tank system generated using a sample time of 0.2 seconds. The input u(t) is the voltage [V] applied to a pump, which generates an inflow to the upper tank. A rather small hole at the bottom of this upper tank yields an outflow that goes into the lower tank, and the output y(t) of the two tank system is then the liquid level [m] of the lower tank. We create an IDDATA object `z` to encapsulate the loaded data:

This data set is also used in the nonlinear grey box example "Two Tank System: C MEX-File Modeling of Time-Continuous SISO System" where physical equations describing the system are assumed. This example is about black box models, thus no knowledge of the actual physical laws is assumed.
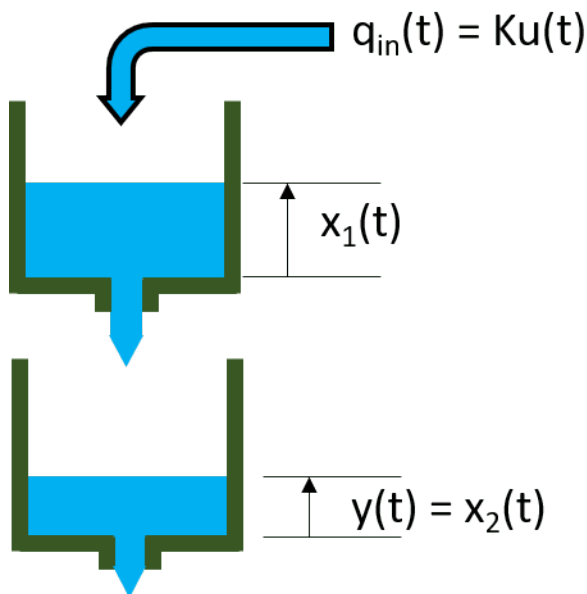

```
load twotankdata
z = iddata(y, u, 0.2, 'Name', 'Two tank system');
```

Splitting the Data and Plotting

Split this data set into 3 subsets of equal size, each containing 1000 samples.

We shall estimate models with `z1`, and evaluate them on `z2` and `z3`. `z1` is similar to `z2`, but not `z3`. Therefore in some sense, `z2` would help us evaluate the interpolation ability of the models, while `z3` would help us evaluate their extrapolation ability.

```
z1 = z(1:1000);
z2 = z(1001:2000);
```

```
z3 = z(2001:3000);
plot(z1,z2,z3)
legend('Estimation','Validation 1', 'Validation 2')
```
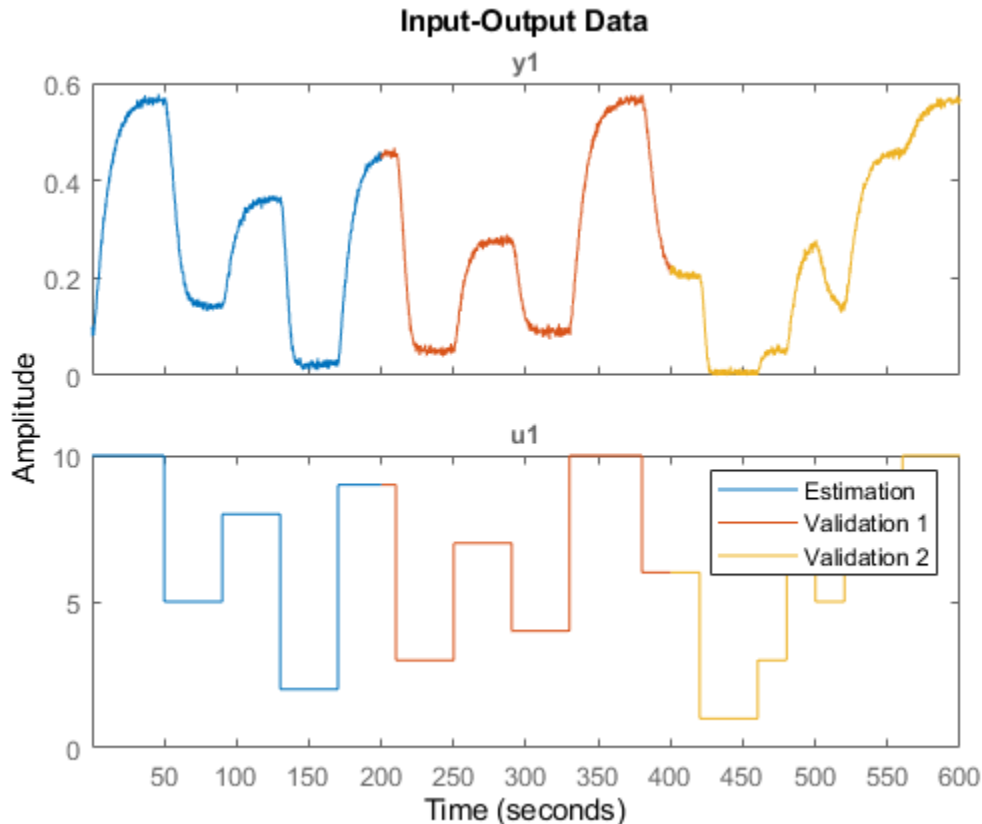

Input-Output Data

Regressor Selection

The first step in estimating nonlinear black box models is to pick the regressor set. The regressors are simple formulas based on time-delayed input/output variables, the simplest case being the variables lagged by a small set of consecutive values. For example, if "u" is the name of an input variable, and "y" the name of an output variable, then an example regressor set can be { y(t-1), y(t-2), u(t), u(t-1), u(t-2) }, where "t" denotes the time variable. Another example involving polynomial terms could be {abs(y(t-2)), y(t-2)*u(t-1), u(t-4)^2}. More complex, arbitrary formulas in the delayed variables are also possible.

In the System Identification Toolbox™, the regressors sets don't have to be created explicitly; a large collection can be generated implicitly by just specifying their form (such as polynomial order), the contributing variables and their lags. Specification objects `linearRegressor`, `polynomialRegressor` and `customRegressor` are used for this purpose. For example R = linearRegressor({'y','u'},{[1 2 4], [2 10]}) implies the regressor set {y(t-1), y(t-2), y(t-4), u(t-2), u(t-10)}. Similarly, R = polynomialRegressor({'y'},{[1 2]},3) implies 3rd order polynomial regressors in variable 'y' with lags 1 and 2, that is, the set {y(t-1)^3, y(t-2)^3}. More configuration choices are available such as using only the absolute values, and/or allowing mix of lags in the regressor formulas.

Linear Regressor Specification Using an Order Matrix

When the model uses only linear regressors that are based on contiguous lags, it is often easier to specify them using an order matrix, rather than using a `linearRegressor` object. The order matrix [na nb nk], defines the numbers of past outputs, past inputs and the input delay used in the regressor formulas. This is the same idea that is used when estimating the linear ARX models (see ARX, IDPOLY). For example, NN = [2 3 4] implies that the output variable uses lags (1,2) and the input variable uses the lags (4,5,6) leading to the regressor set {y(t-1), y(t-2), u(t-4), u(t-5), u(t-6)}.

Usually model orders are chosen by trial and error. Sometimes it is useful to try linear ARX models first in order to get hints about the best orders. So let us first try to determine the optimal orders for a linear ARX model, using the functions `arxstruc` and `selstruc`.

```
V = arxstruc(z1,z2,struc(1:5, 1:5,1:5));
% select best order by Akaike's information criterion (AIC)
nn = selstruc(V,'aic')


nn =

    5     1     3
```

The AIC criterion has selected nn = [na nb nk] = [5 1 3], meaning that, in the selected ARX model structure, y(t) is predicted by the 6 regressors y(t-1),y(t-1),...,y(t-5) and u(t-3). We will try to use these values when estimating nonlinear models.

Nonlinear ARX (IDNLARX) Models

In an IDNLARX model, the model output is a nonlinear function of regressors, like y(t) = Fcn(y(t-1),y(t-1),...,y(t-5), u(t-3)).

IDNLARX models are typically estimated with the syntax:

```
M = NLARX(Data, Orders, Fcn).
```

It is similar to the linear ARX command, with the additional argument "Fcn" specifying the type of nonlinear function. A Nonlinear ARX model produces its output by using a 2-stage transformation: 1. Map the training data (input-output signals) to a set of regressors. Numerically, the regressor set is a double matrix R, with one column of data for each regressor. 2. Map the regressors to a single output, y = Fcn®, where Fcn() is a scalar nonlinear function.

To estimate an IDNLARX model, after the choice of model orders, we need to pick the type of nonlinear mapping function Fcn() to use. Let us first try a Wavelet Network (idWaveletNetwork) function.

```
mw1 = nlarx(z1,[5 1 3], idWaveletNetwork);
```

The Wavelet Network uses a combination of wavelet units to map the regressors to the output of the model. The model `mw1` is an @idnalrx object. It stores the nonlinear mapping function (idWaveletNetwork object here) in its `OutputFcn` property. The number of these units to use can be specified in advance, or we can leave it to the estimation algorithm to determine an optimal value automatically. The property NonlinearFcn.NumberOfUnits stores the chosen number of units in the model.

```
NLFcn = mw1.OutputFcn;
NLFcn.NonlinearFcn.NumberOfUnits
```

```
ans =

    3
```

Examine the result by comparing the response of the model `mw1` to the measured output in the dataset `z1`.

```
compare(z1,mw1);
```
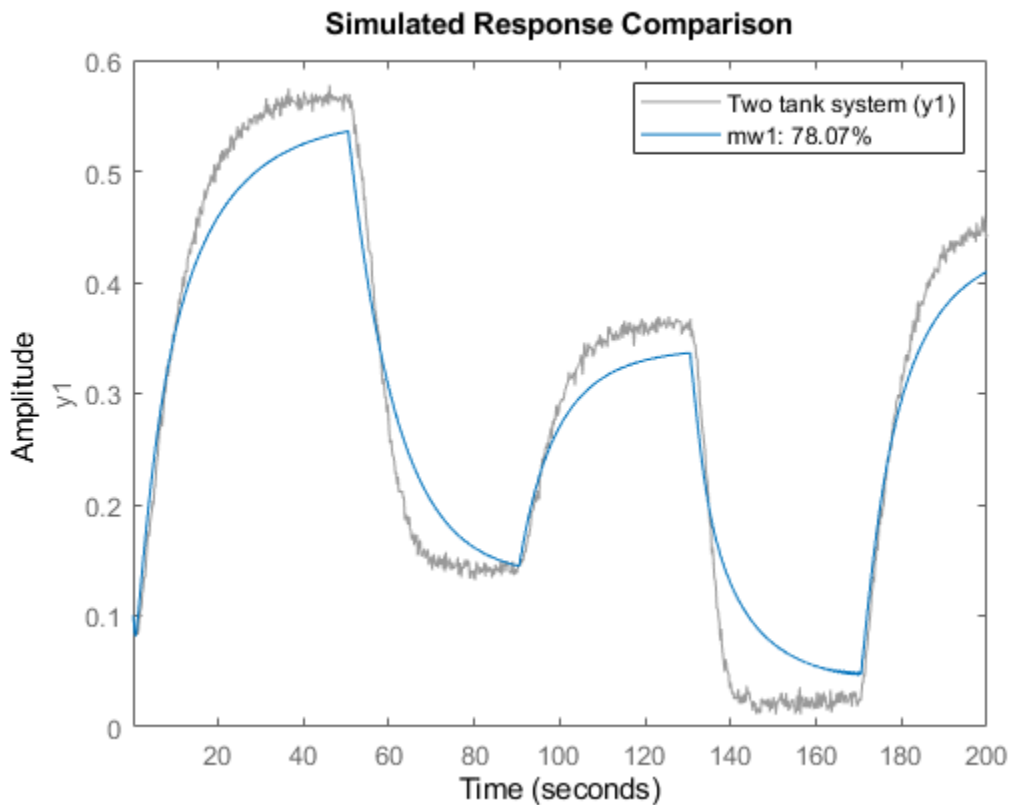

Using Model Properties

The first model was not quite satisfactory, so let us try to modify some properties of the model. Instead of letting the estimation algorithm automatically choose the number of units in the idWaveletNetwork function, this number can be explicitly specified.

```
mw2 = nlarx(z1,[5 1 3], idWaveletNetwork(8));
```

Default InputName and OutputName have been used.

```
mw2.InputName
mw2.OutputName
```

```
ans =

  1×1 cell array
```

```
    {'u1'}


ans =

  1×1 cell array

    {'y1'}
```

The regressors of the IDNLARX model can be viewed using the GETREG command. The regressors are made from `mw2.InputName`, `mw2.OutputName` and time delays.

```
getreg(mw2)


ans =

  6×1 cell array

    {'y1(t-1)'}
    {'y1(t-2)'}
    {'y1(t-3)'}
    {'y1(t-4)'}
    {'y1(t-5)'}
    {'u1(t-3)'}
```

Note that the order matrix is equivalent to the linear regressor set created using the specification object, as in:

```
R = linearRegressor([z1.OutputName; z1.InputName],{1:5, 3});
mw2_a = nlarx(z1, R, idWaveletNetwork(8));
getreg(mw2_a)


ans =

  6×1 cell array

    {'y1(t-1)'}
    {'y1(t-2)'}
    {'y1(t-3)'}
    {'y1(t-4)'}
    {'y1(t-5)'}
    {'u1(t-3)'}
```

`mw2_a` is essentially the same model as `mw2`. The use of a specification object allows more flexibility; use it when you do not want to use continguous lags, or have a minimum lag in output variables that is greater than one.

Assigning Regressors to Mapping Functions

The output of an IDNLARX model is a static function of its regressors. Usually this mapping function has a linear block, a nonlinear block, plus a bias term (output offset). The model output is the sum of the outputs of these blocks and the bias. In order to reduce model complexity, only a subset of

regressors can be chosen to enter the nonlinear block and the linear block. The assignment of the regressors to the linear and nonlinear components of the mapping function is performed using the RegressorUsage property of the model.

```
RegUseTable = mw2.RegressorUsage


RegUseTable =

  6×2 table

              y1:LinearFcn     y1:NonlinearFcn
              _____     _____

    y1(t-1)       true              true
    y1(t-2)       true              true
    y1(t-3)       true              true
    y1(t-4)       true              true
    y1(t-5)       true              true
    u1(t-3)       true              true
```

RegUseTable is a table that shows which regressors are used as inputs to the linear and the nonlinear components of the Wavelet Network. Each row corresponds to one regressor. Suppose we want to use only those regressors that are functions of input variable 'u1' in the nonlinear component. This configuration can be achieved as follows.

```
RegNames = RegUseTable.Properties.RowNames;
I = contains(RegNames,'u1');
RegUseTable.("y1:NonlinearFcn")(~I) = false;
```

As an example, consider a model that uses polynomial regressors of order 2 in addition to the linear ones used by mw2. First, generate a polynomial regressor specification set.

```
P = polynomialRegressor({'y1','u1'},{1:2, 0:3},2)


P =

<strong>Order 2 regressors in variables y1, u1</strong>
               Order: 2
           Variables: {'y1'  'u1'}
                Lags: {[1 2]  [0 1 2 3]}
         UseAbsolute: [0 0]
    AllowVariableMix: 0
         AllowLagMix: 0
        TimeVariable: 't'
```

Now add the specification P to the model's Regressors property.

```
mw3 = mw2; % start with the mw2 structure for new estimation
mw3.Regressors = [mw3.Regressors; P]; % add the polynomial set to the model structure
```

Finally, update the parameters of the model to fit the data by minimizing 1-step ahead prediction errors.

```
mw3 = nlarx(z1, mw3)
```

```
mw3 =

<strong>Nonlinear ARX model with 1 output and 1 input</strong>
  Inputs: u1
  Outputs: y1

Regressors:
  1. Linear regressors in variables y1, u1
  2. Order 2 regressors in variables y1, u1

Output function: Wavelet network with 8 units
Sample time: 0.2 seconds

Status:
Estimated using NLARX on time domain data "Two tank system".
Fit to estimation data: 96.76% (prediction focus)
FPE: 3.499e-05, MSE: 3.338e-05
```

Evaluating Estimated Models

Different models, both linear and nonlinear, can be compared together in the same COMPARE command.

```
mlin = arx(z1,[5 1 3]);   % linear ARX model
%
compare(z1,mlin,mw2,mw3)
```
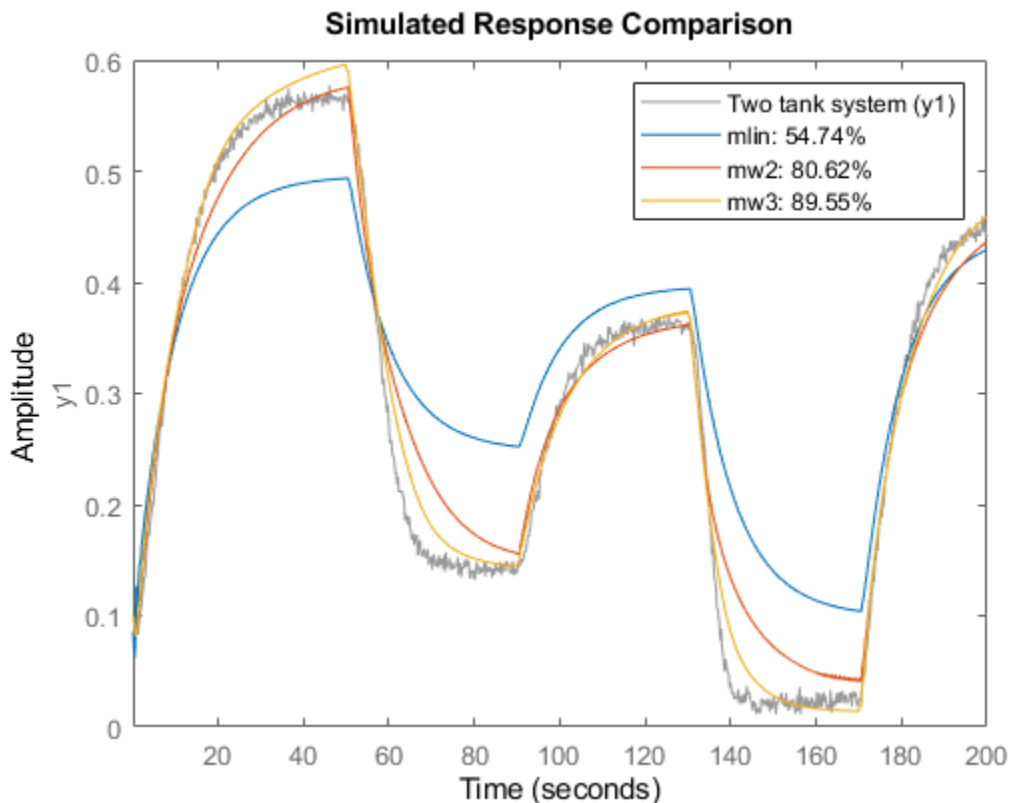
Model validation can be carried out by running the COMPARE command on the validation datasets. First, validate the models by checking how well their simulated responses match the measured output signal in z2.

```
compare(z2,mlin,mw2,mw3)
```
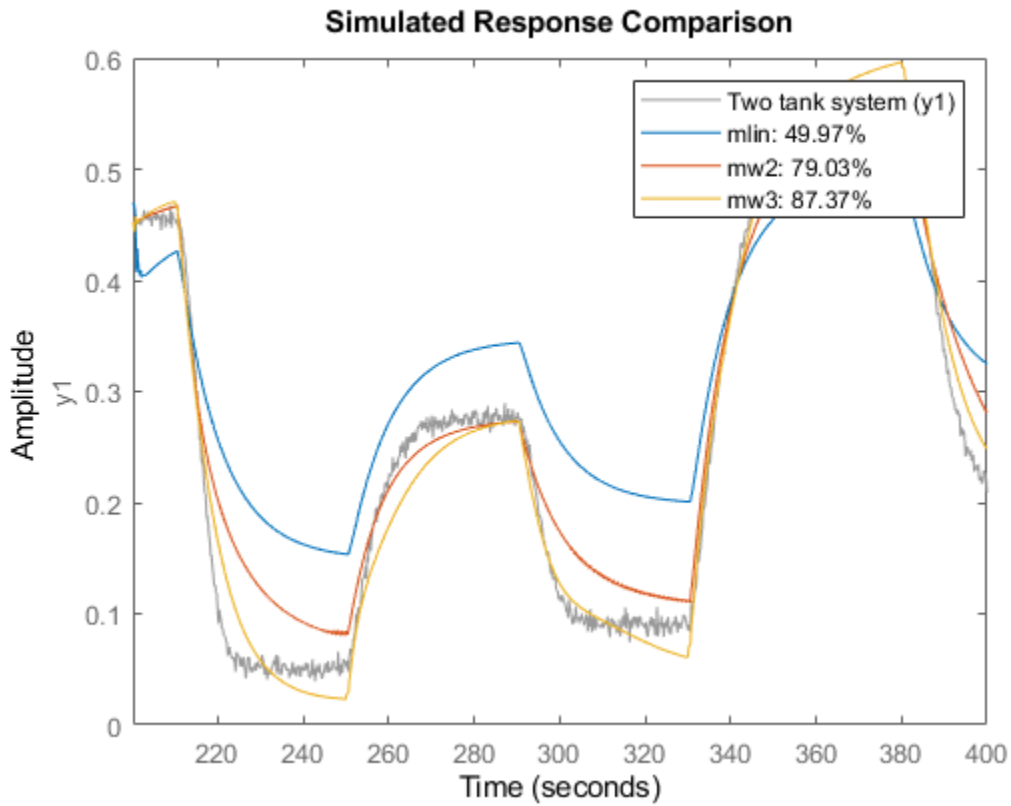

Similarly, validate the estimated models against the dataset z3.

```
compare(z3,mlin,mw2,mw3)
```
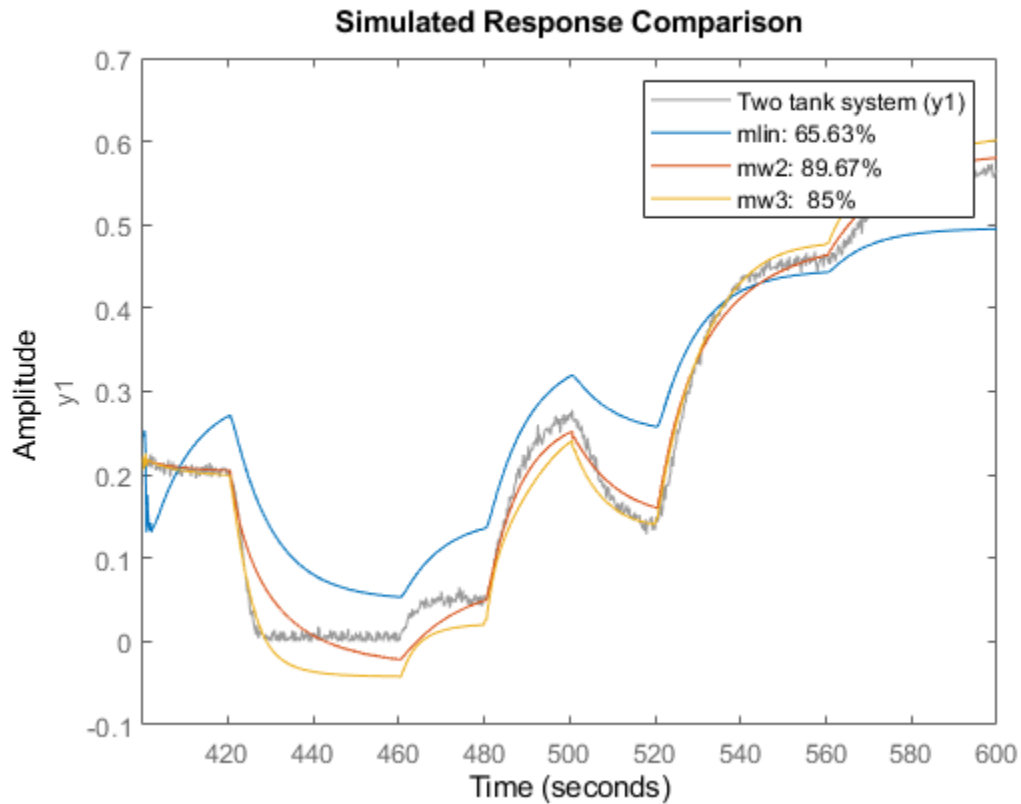
An analysis of the model residues may also be performed to validate the model quality. We expect the residues to be white (uncorrelated with itself except at lag 0) and uncorrelated with the input signal. The resideus are the 1-step ahead prediction errors. The residues can be visualized with bounds on insignificant values by using the RESID command. For example on the valiation dataset z2, the residues for the three identified models are shown below.

```
resid(z2,mlin,mw2,mw3)
legend show
```
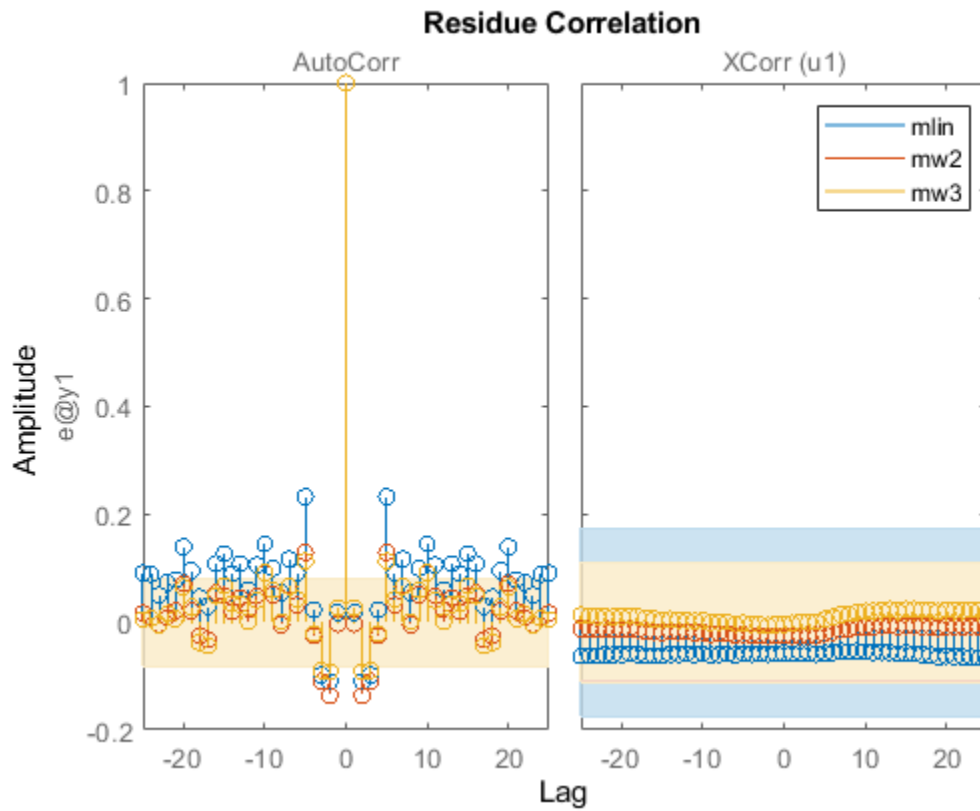
The residues are all uncorrelated with the input signal of `z2` (z2.InputData). However, they exhibit some auto-correlation for the models `mlin` and `mw3`. The model `mw2` performs better on both (auto- and cross-correlation) tests.

Function PLOT may be used to view the nonlinearity responses of various IDNLARX models. The plot essentially shows the characteristics of the nonlinear mapping function Fcn().

```
plot(mw2,mw3) % plot nonlinearity response as a function of selected regressors
```
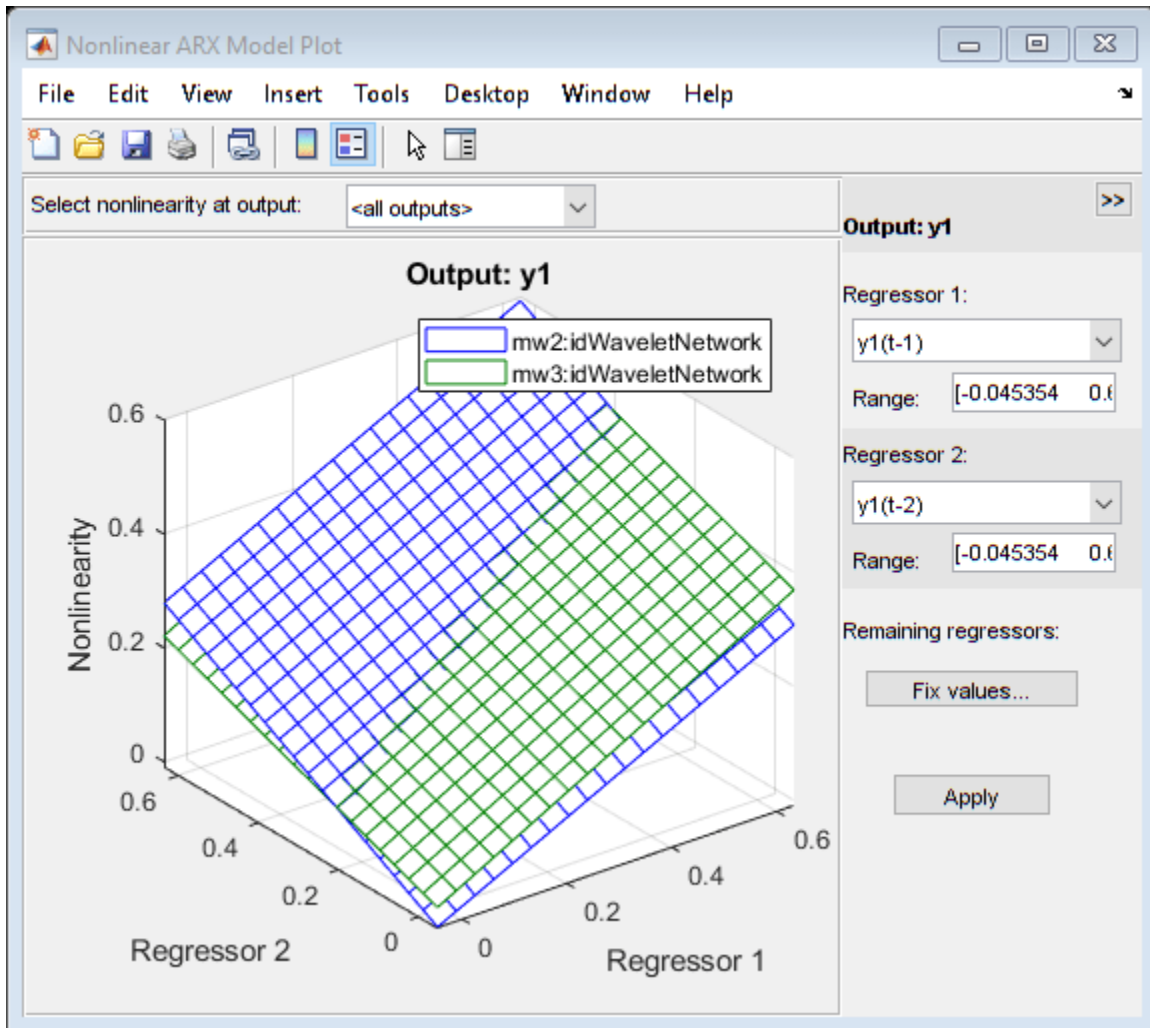
Nonlinear ARX Model with Sigmoid Network

In place of a Wavelet Network, other nonlinear functions can be used in the model. Try the idSigmoidNetwork function, which maps the regressors to the output using a sum of sigmoid units (plus a linear and an offset term). Configure the network to use 8 units of sigmoid (dilated and translated by different amounts).

```
ms1 = nlarx(z1,[5 1 3], idSigmoidNetwork(8));
compare(z1,ms1)
```
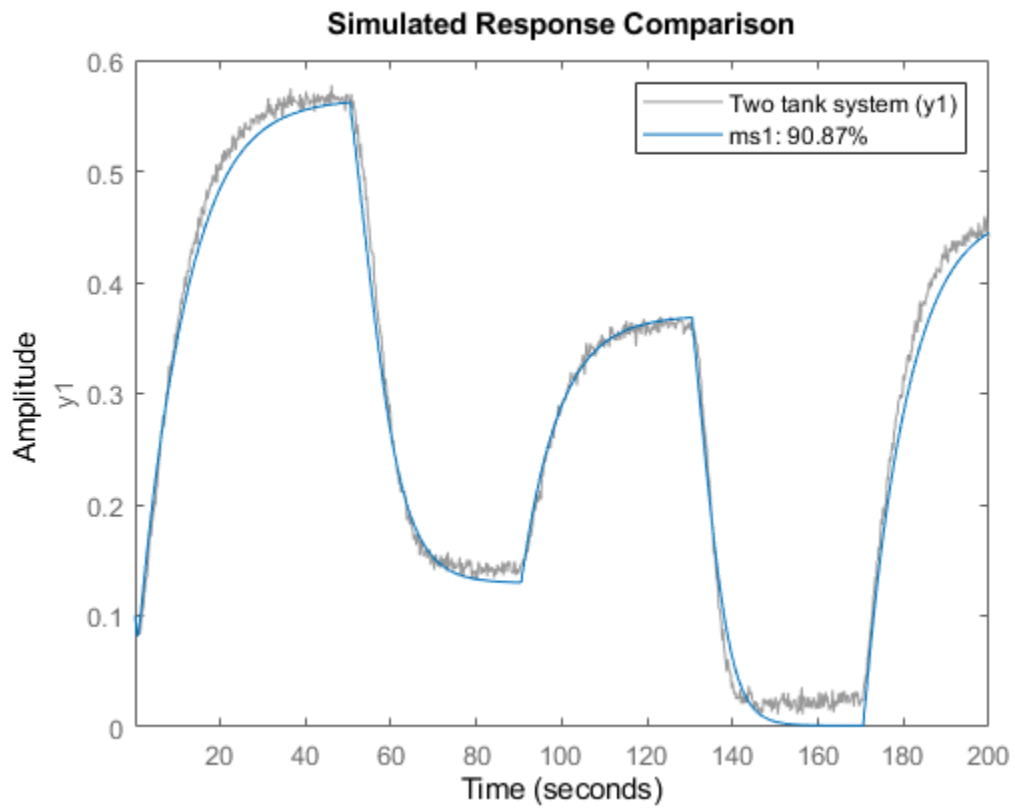
Now evaluate the new model on the data set z2.

```
compare(z2,ms1);
```
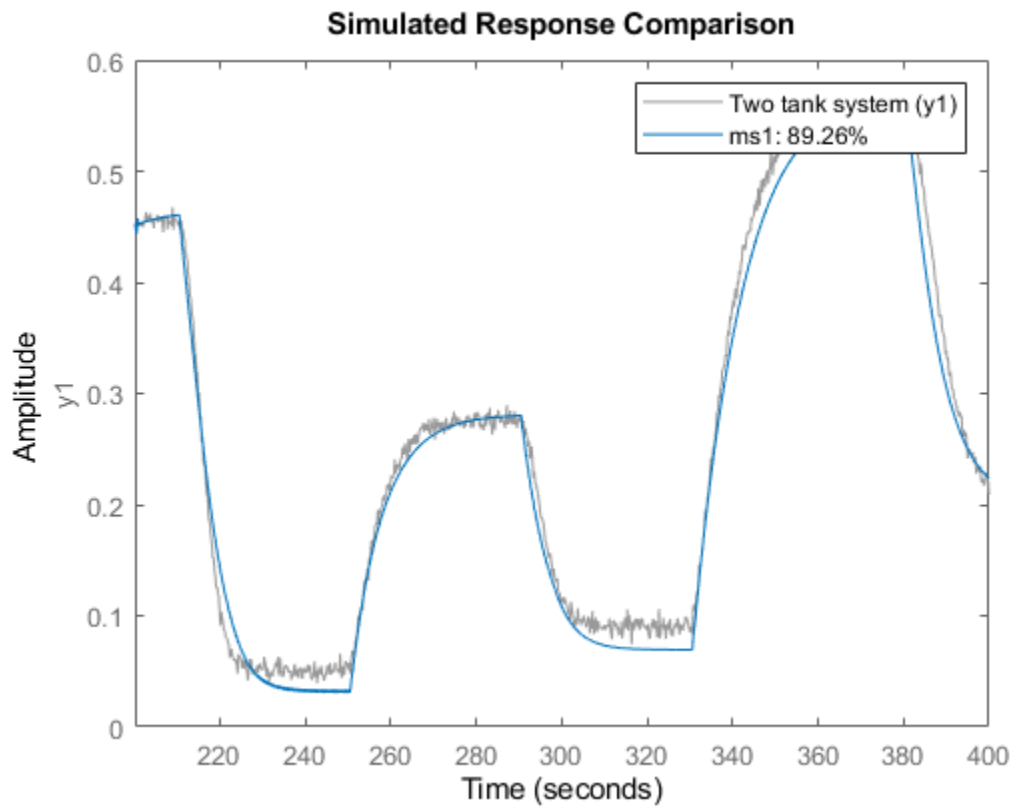
and on the data set z3.

```
compare(z3,ms1);
```
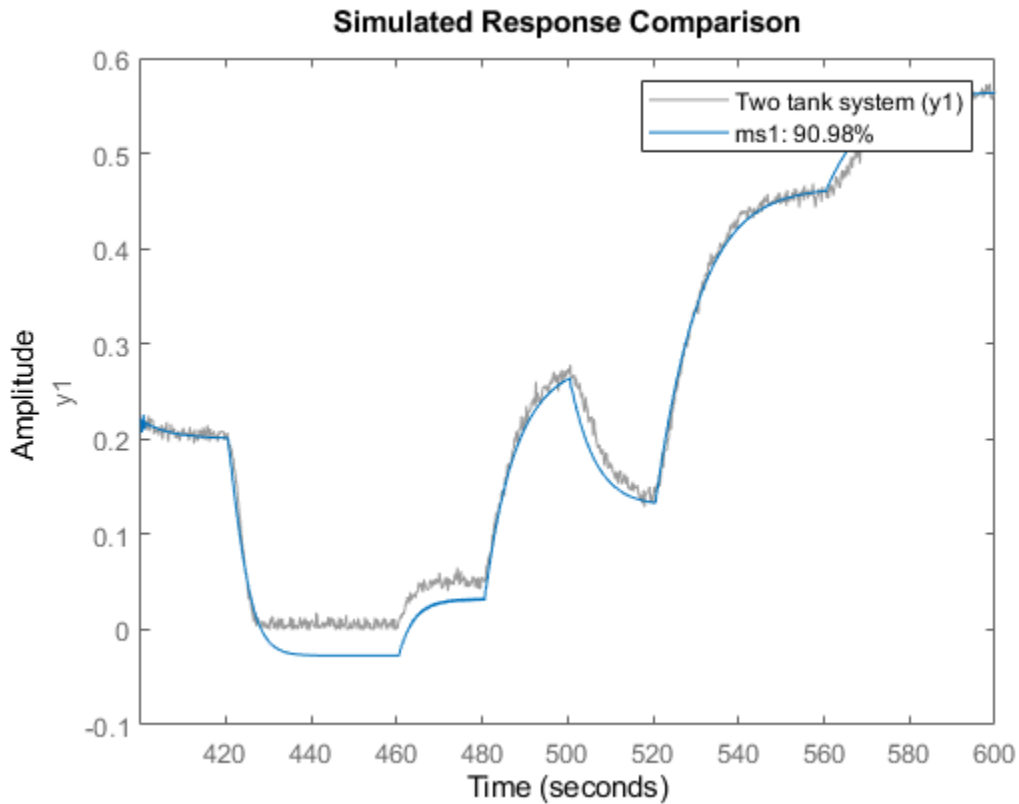
Other Nonlinear Mapping Functions in IDNLARX Model

idCustomNetwork is similar to idSigmoidNetwork, but instead of the sigmoid unit function, users can define their own unit function in MATLAB files. This example uses the gaussunit function.

```
type gaussunit.m
```

```
function [f, g, a] = gaussunit(x)
%GAUSSUNIT Unit function for use in idCustomNetwork (example)
%
%[f, g, a] = GAUSSUNIT(x)
%
% x: unit function variable
% f: unit function value
% g: df/dx
% a: unit active range (g(x) is significantly non zero in the interval [-a a])
%
% The unit function must be "vectorized": for a vector or matrix x, the output
% arguments f and g must have the same size as x, computed element-by-element.

% Copyright 2005-2021 The MathWorks, Inc.
% Author(s): Qinghua Zhang

f =  exp(-x.*x);

if nargout>1
   g = - 2*x .* f;
```

```
    a = 0.2;
end
```

For the customnet based model, use only the third, fifth and the sixth regressors as input to the nonlinear component of the customnet function.

```
mc1_ini = idnlarx('y1','u1', [5 1 3], idCustomNetwork(@gaussunit));
Use = false(6,1);
Use([3 5 6]) = true;
mc1_ini.RegressorUsage{:,2} = Use;
mc1 = nlarx(z1,mc1_ini);
```

Tree partition function is another mapping function that can be used in the Nonlinear ARX model. Estimate a model using idTreePartition function:

```
mt1 = nlarx(z1, [5 1 3], idTreePartition);
```

Compare responses of models `mc1` and `mt1` to data `z1`.

```
compare(z1, mc1, mt1)
```
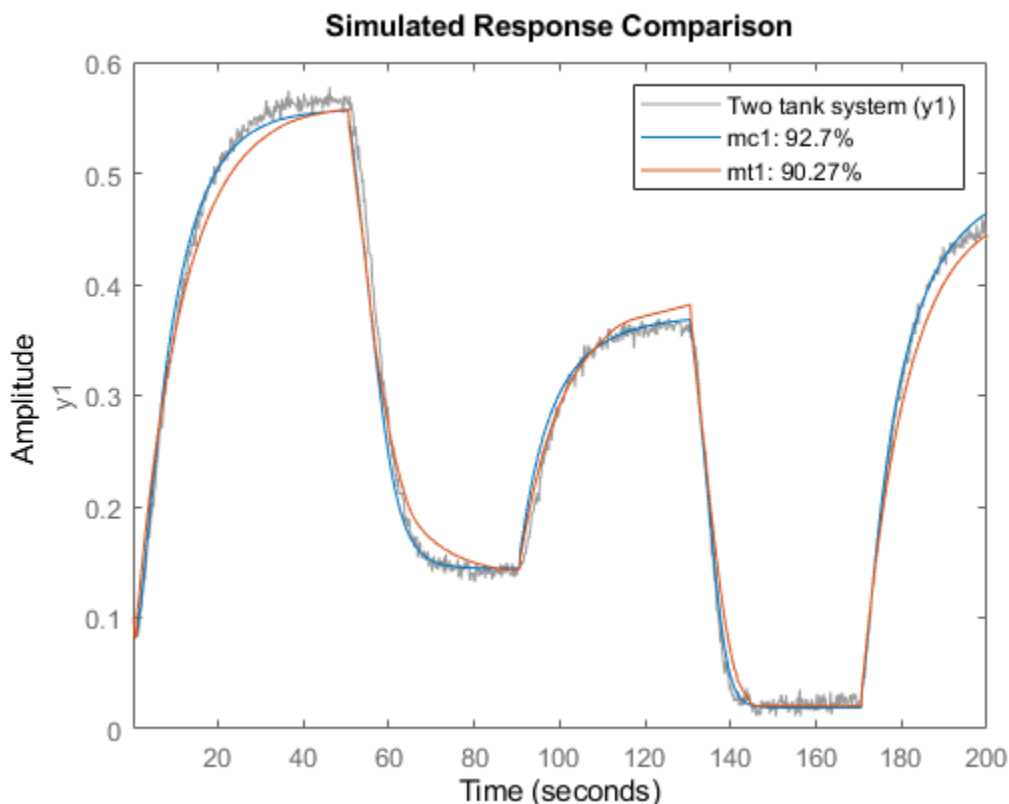

Using Regression Algorithms from Statistics and Machine Learning Toolbox

If you have access to the Statistics and Machine Learning Toolbox™ you can also use gaussian process (GP) functions and bagged or boosted tree ensembles to create your Nonlinear ARX models. The GPs are represented by the idGaussianProcess object, while regression tree ensembles are represented by the idTreeEnsemble object. GPs use squared-exponential kernel by default. Other

kernel types may be specified when creating the object. In the example here, we use a 'Matern32' kernel.

```matlab
if exist('fitrgp','file')==2
    Warn = warning('off','stats:lasso:MaxIterReached');
    mp1 = nlarx(z1, [5 1 3], idGaussianProcess('Matern32'))

    % Create boosted ensemble of 200 regression trees
    En = idTreeEnsemble;
    En.EstimationOptions.FitMethod = 'lsboost-resampled';
    En.EstimationOptions.NumLearningCycles = 200;
    En.EstimationOptions.ResampleFraction = 1;
    mp2 = nlarx(z1, [0 70 3], En)
    compare(z1, mp1, mp2) % compare model fits to estimation data
    warning(Warn)
end
```

```
mp1 =

<strong>Nonlinear ARX model with 1 output and 1 input</strong>
  Inputs: u1
  Outputs: y1

Regressors:
  Linear regressors in variables y1, u1

Output function: Gaussian process function using a Matern32 kernel.
Sample time: 0.2 seconds

Status:
Estimated using NLARX on time domain data "Two tank system".
Fit to estimation data: 97.1% (prediction focus)
FPE: 2.738e-05, MSE: 2.675e-05

mp2 =

<strong>Nonlinear ARX model with 1 output and 1 input</strong>
  Inputs: u1
  Outputs: y1

Regressors:
  Linear regressors in variables u1

Output function: Bagged Regression Tree Ensemble
Sample time: 0.2 seconds

Status:
Estimated using NLARX on time domain data "Two tank system".
Fit to estimation data: 91.14%
MSE: 0.0002503
```
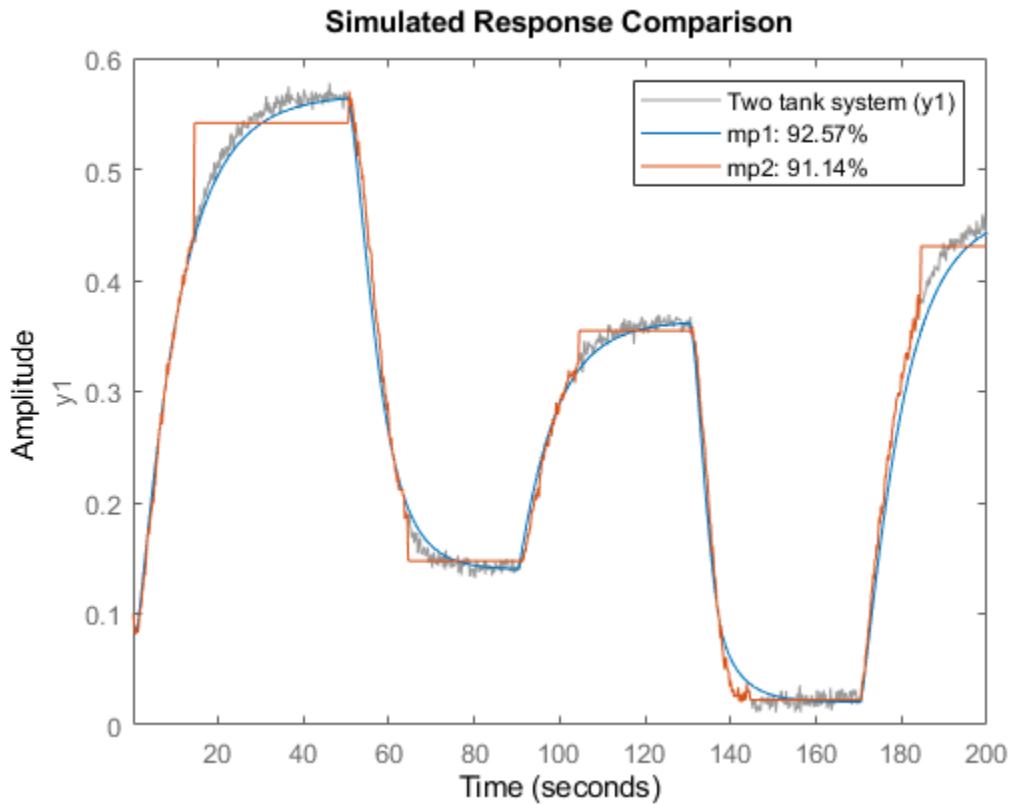
Using the Network Object from Deep Learning Toolbox

If you have the Deep Learning Toolbox™ available, you can also use a neural network to define the mapping function Fcn() of your IDNLARX model. This network must be a feed-forward network (no recurrent component).

Here, we will create a single-output network with an unknown number of inputs (denote by input size of zero in the network object). The number of inputs is set to the number of regressors during the estimation process automatically.

```matlab
if exist('feedforwardnet','file')==2 % run only if Deep Learning Toolbox is available
    ff = feedforwardnet([5 20]);  % use a feed-forward network to map regressors to output
    ff.layers{2}.transferFcn = 'radbas';
    ff.trainParam.epochs = 50;
    % Create a neural network mapping function that wraps the feed-forward network
    OutputFcn = idFeedforwardNetwork(ff);
    mn1 = nlarx(z1,[5 1 3], OutputFcn); % estimate network parameters
    compare(z1, mn1) % compare fit to estimation data
end
```
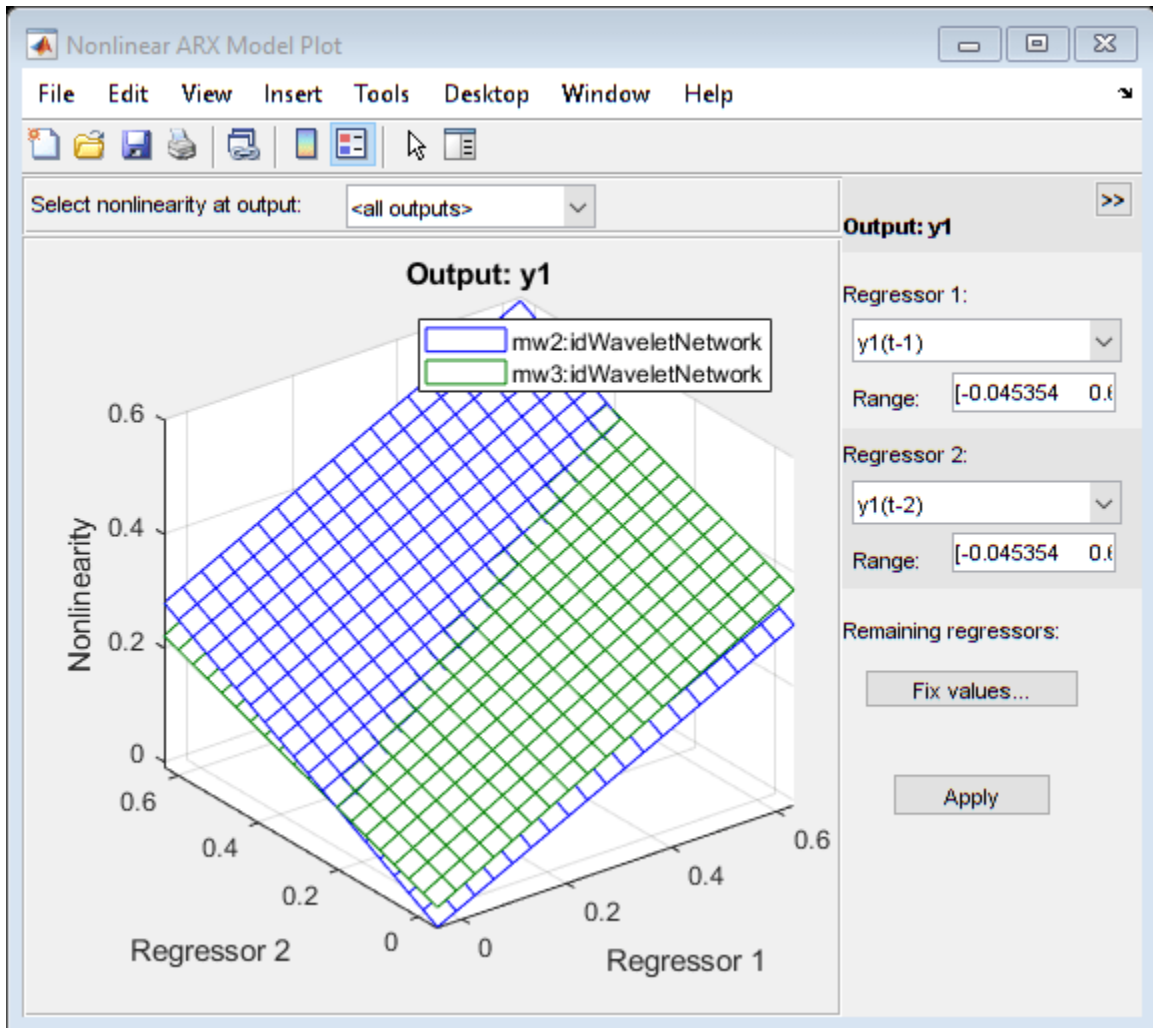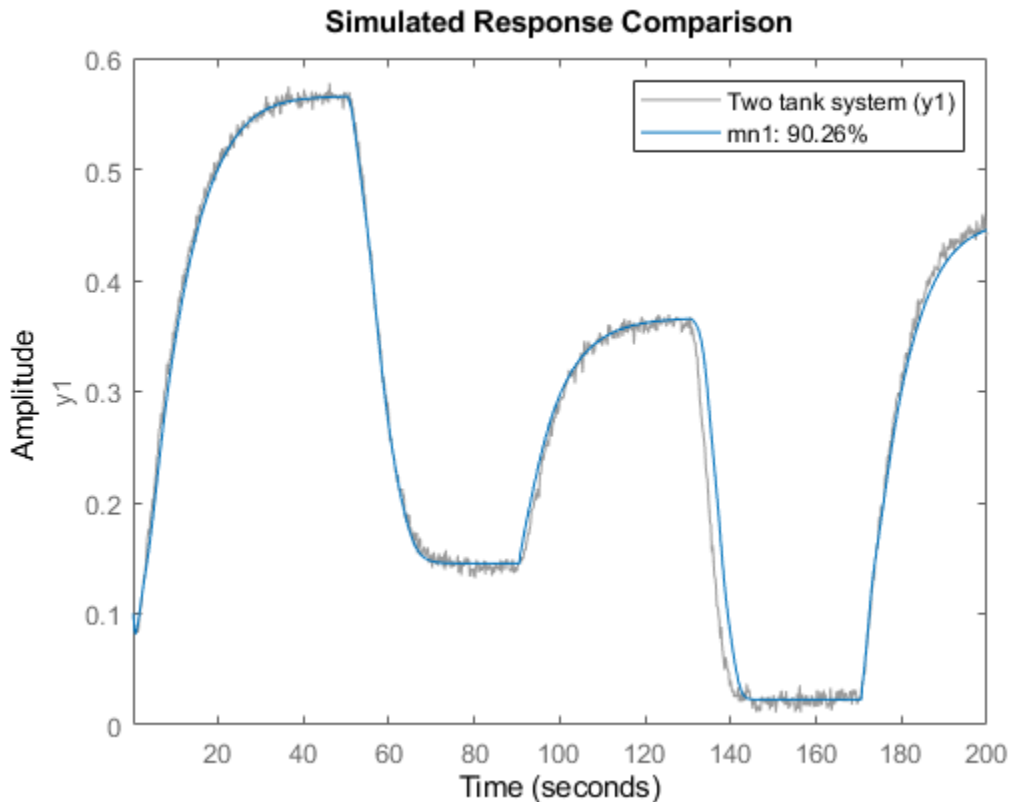
Hammerstein-Wiener (IDNLHW) Models

A Hammerstein-Wiener model is composed of up to 3 blocks: a linear transfer function block is preceded by a nonlinear static block and/or followed by another nonlinear static block. It is called a Wiener model if the first nonlinear static block is absent, and a Hammerstein model if the second nonlinear static block is absent.

IDNLHW models are typically estimated with the syntax:

```
M = NLHW(Data, Orders, InputNonlinearity, OutputNonlinearity).
```

where Orders = [nb bf nk] specifies the orders and delay of the linear transfer function. InputNonlinearity and OutputNonlinearity specify the nonlinear functions for the two nonlinear blocks, at the input and output end of the linear block. M is an @idnlhw model. The linear Output-Error (OE) model corresponds to the case of trivial (unit gain) nonlinearities, that is, if the input and the output nonlinear functions of an IDNLHW model are both unit gains, the model structure is equivalent to a linear OE model.

Hammerstein-Wiener Model with the Piecewise Linear Nonlinear Function

The idPiecewiseLinear (piecewise linear) nonlinear function is useful in general IDNLHW models.

Notice that, in Orders = [nb nf nk], nf specifies the number of poles of the linear transfer function, somewhat related to the "na" order of the IDNLARX model. "nb" is related to the number if zeros.

```
mhw1 = nlhw(z1, [1 5 3], idPiecewiseLinear, idPiecewiseLinear);
```

The result can be evaluated with COMPARE as before.

```
compare(z1,mhw1)
```
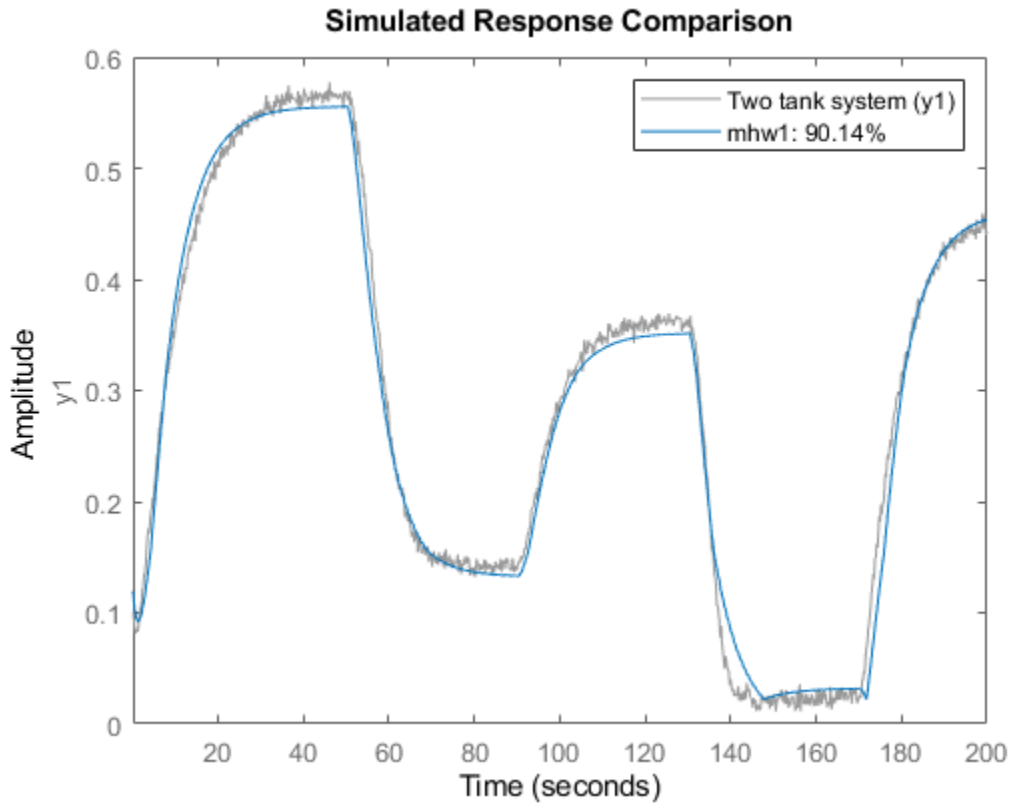

Model properties can be visualized by the PLOT command.

Click on the block-diagram to choose the view of the input nonlinearity (UNL), the linear block or the output nonlinearity (YNL).

For the linear block, it is possible to choose the type of plots within Step response, Bode plot, Impulse response and Pole-zero map.

```
plot(mhw1)
```
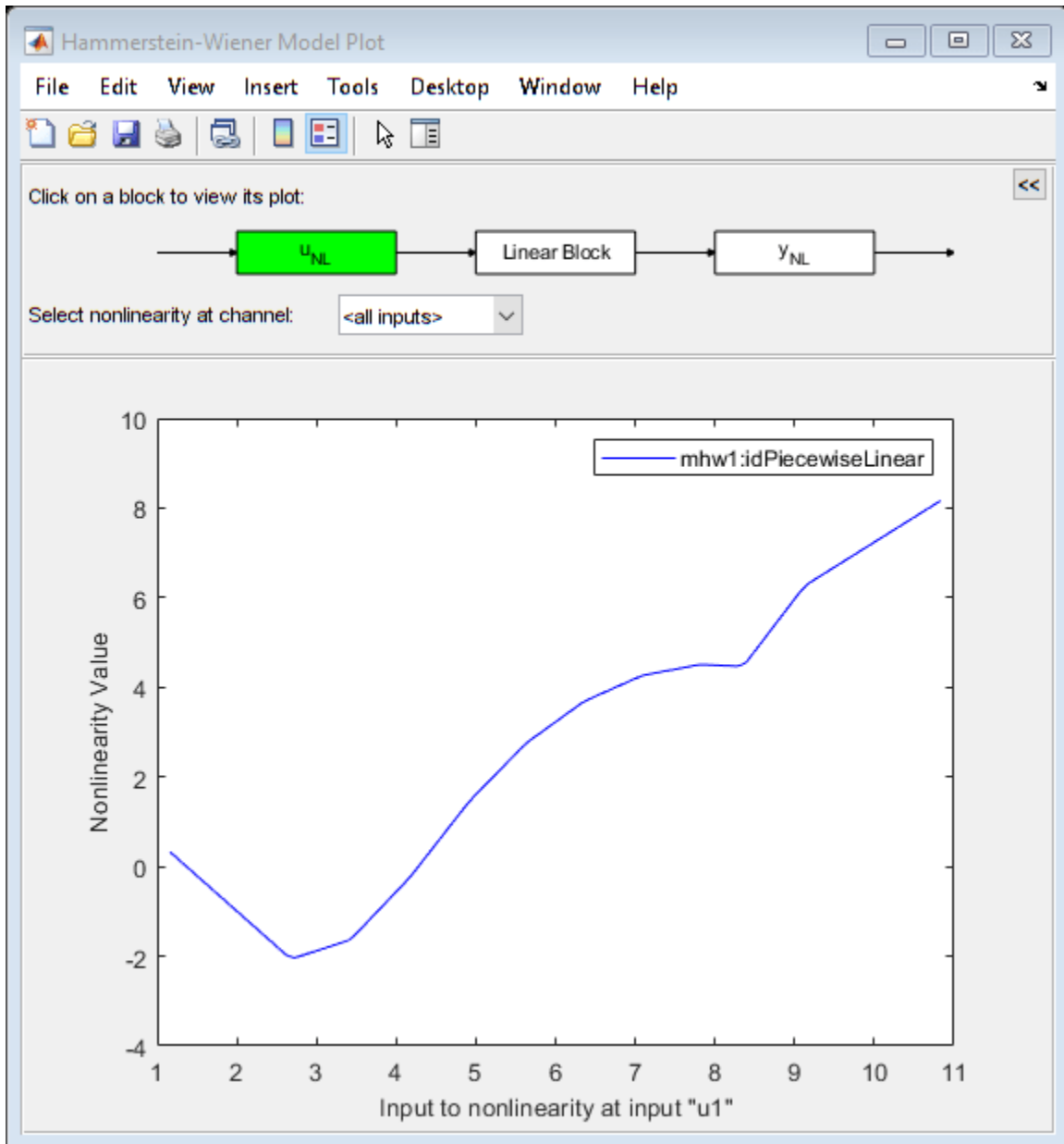
The break points of the two piecewise linear functions can be examined. This is a two row matrix, the first row corresponds to the input and the second row to the output of the idPiecewiseLinear function.

```
mhw1.InputNonlinearity.BreakPoints
```

```
ans =

  Columns 1 through 7

    2.6795    3.4262    4.1729    4.9195    5.6324    6.3599    7.0874
   -2.0650   -1.6316   -0.2396    1.4602    2.7554    3.6889    4.2632

  Columns 8 through 10
```

```
   7.8148    8.3543    9.1260
   4.5074    4.4653    6.2634
```

Hammerstein-Wiener Model with Saturation and Dead Zone Nonlinearities

The saturation and dead-zone functions are physics inspired nonlinear functions. They can be used to describe actuator/sensor saturation scenario and dry friction effects.

```
mhw2 = nlhw(z1, [1 5 3], idDeadZone, idSaturation);
compare(z1,mhw2)
```
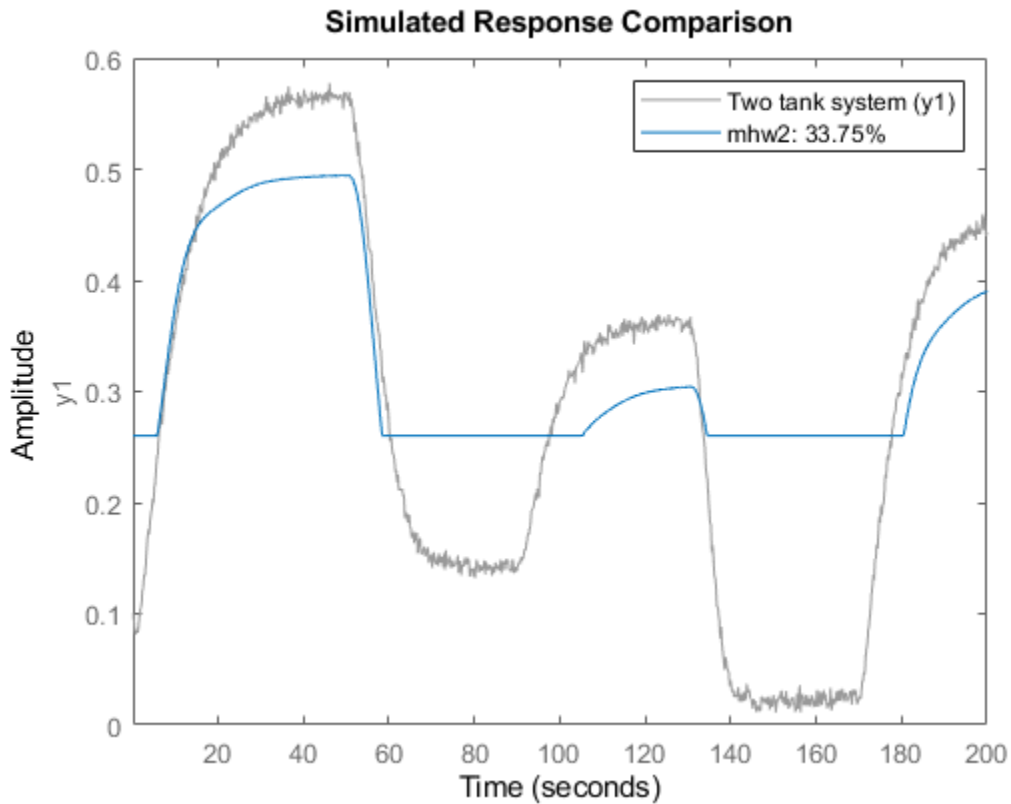

The absence of a nonlinearity is indicated by the IDUNITGAIN object, or by the empty "[]" value. A unit gain is a just a feed-through of an input signal to the output without any modification.

```
mhw3 = nlhw(z1, [1 5 3], idDeadZone, idUnitGain); % no output nonlinearity
mhw4 = nlhw(z1, [1 5 3], [],idSaturation); % no input nonlinearity
```

The limit values of dead-zone and saturation functions can be examined after estimation.

```
mhw2.InputNonlinearity.ZeroInterval
mhw2.OutputNonlinearity.LinearInterval
```

```
ans =

   -0.9982    4.7565
```

```
ans =

    0.2601    0.5848
```

The initial guess of ZeroInterval for dead-zone or LinearInterval for saturation function can be specified while estimating the model.

```
mhw5 = nlhw(z1, [1 5 3], idDeadZone([0.1 0.2]), idSaturation([-1 1]));
```

Hammerstein-Wiener Model - Specifying Properties

The estimation algorithm options can be specified using the NLHWOPTIONS command.

```
opt = nlhwOptions();
opt.SearchMethod = 'gna';
opt.SearchOptions.MaxIterations = 7;
mhw6 = nlhw(z1, [1 5 3], idDeadZone, idSaturation, opt);
```

An already-estimated model can be refined by more estimation iterations.

Evaluated on the estimation data `z1`, `mhw7` should have a better fit than `mhw6`.

```
opt.SearchOptions.MaxIterations = 30;
mhw7 = nlhw(z1, mhw6, opt);
compare(z1, mhw6, mhw7)
```
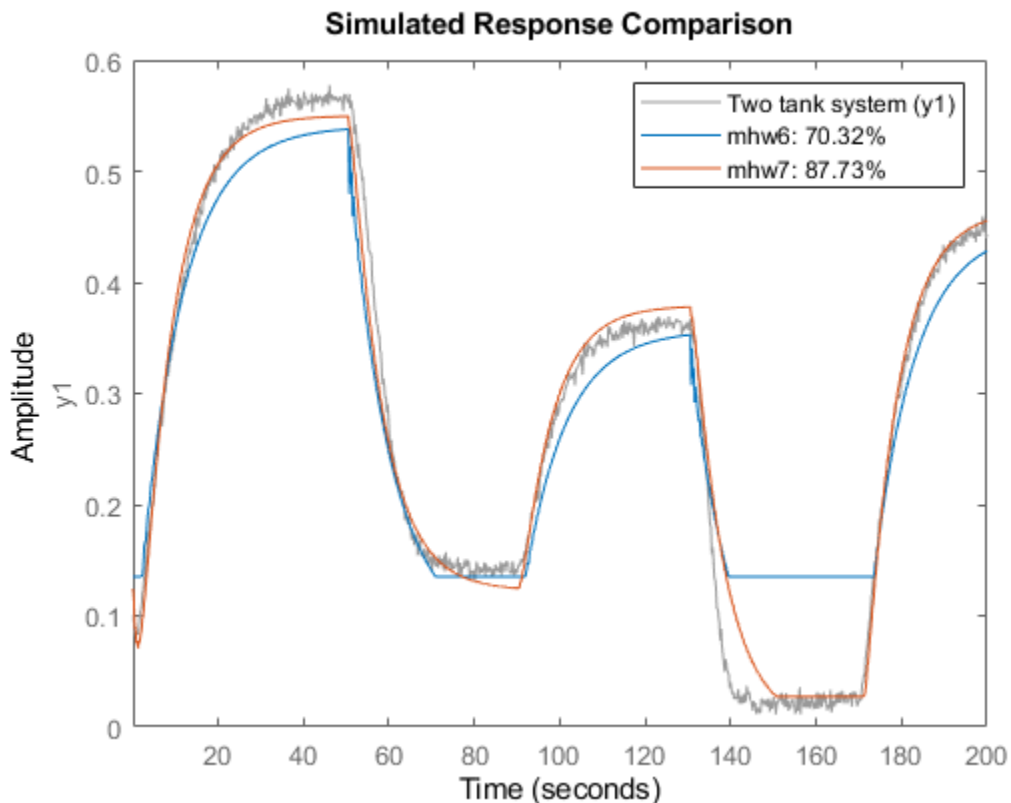
Hammerstein-Wiener Model - Use Other Nonlinear Functions

The nonlinear functions Piecewise Linear (idPiecewiseLinear), Saturation (idSaturation) and Dead Zone (idDeadZone) have been mainly designed for use in IDNLHW models. They can only estimate single variable nonlinearities. The other more general nonlinear functions, Sigmoid Network (idSigmoidNetwork), Custom Network (idCustomNetwork) and Wavelet Network (idWaveletNetwork) can also be used in IDNLHW models (besides the IDNLARX models). However, the non-differentiable functions Tree-partition (idTreePartition), Multi-layer Neural Network (idFeedforwardNetwork), Gaussian Process functions (idGaussianProcess) and Regression Tree Ensembles (idTreeEnsemble) cannot be used because the estimation of IDNLHW models require differentiable nonlinear functions.

Motorized Camera - Multi-Input Multi-Output Nonlinear ARX and Hammerstein-Wiener Models

This example shows how to estimate multi-input multi-output (MIMO) nonlinear black box models from data. Two types of nonlinear black box models are offered in the System Identification Toolbox™ - Nonlinear ARX and Hammerstein-Wiener models.

The Measured Data Set

The data saved in the file motorizedcamera.mat will be used in this example. It contains 188 data samples, collected from a motorized camera with a sample time of 0.02 seconds. The input vector u(t) is composed of 6 variables: the 3 translation velocity components in the orthogonal X-Y-Z coordinate system fixed to the camera [m/s], and the 3 rotation velocity components around the X-Y-Z axes [rad/s]. The output vector y(t) contains 2 variables: the position (in pixels) of a point which is the image taken by the camera of a fixed point in the 3D space. We create an IDDATA object z to hold the loaded data:


```
load motorizedcamera
z = iddata(y, u, 0.02, 'Name', 'Motorized Camera', 'TimeUnit', 's');
```

Nonlinear ARX (IDNLARX) Model - Picking Regressors

Let us first try nonlinear ARX models. Two important elements need to be chosen: the model regressors and the nonlinear mapping functions.

The regressors are simple formulas based on time-delayed I/O variables, the simplest case being the variables lagged by a small set of consecutive values. For example, if "u" in the name of the input variable, and "y" the name of the output variables, then an example regressor set can be { y(t-1), y(t-2), u(t), u(t-1), u(t-2) }, where "t" denotes the time variable. Another example involving polynomial terms could be {y(t-2)^4, y(t-2)*u(t-1), u(t-4)^2}. More complex, arbitrary formulas in the delayed variables are also possible.

The easiest way of specifying linear regressors is by using an "orders matrix". This matrix takes the form NN = [na nb nk] and it specifies by how many lags each output (na) and each input variable (nb, nk) are delayed. This is the same idea that is used when estimating the linear ARX models (see ARX, IDPOLY). For example, NN = [2 3 4] implies the regressor set {y(t-2), u(t-4), u(t-5), u(t-6)}. In the

general case of models with NY outputs and NU inputs, NN is a matrix with NY rows and NY+2*NU rows.

Nonlinear ARX (IDNLARX) Model - Preliminary Estimation Using Wavelet Network

To start, let us choose the orders NN = [na nb nk] = [ones(2,2), ones(2,6), ones(2,6)]. It means that each output variable is predicted by all the output and input variables, each being delayed by 1 sample. The model equation can be written as y_i(t) = F_i(y1(t-1), y2(t-1), u1(t-1), u2(t-1), u3(t-1)), i = 1, 2. Let us choose a Wavelet Network (idWaveletNetwork object) as the nonlinear mapping function for both outputs. The function NLARX estimates the parameters of the Nonlinear ARX model.

```
NN = [ones(2,2), ones(2,6), ones(2,6)]; % the orders
mw1 = nlarx(z, NN, idWaveletNetwork);
```

Examine the result by comparing the output simulated with the estimated model and the output in the measured data z:

```
compare(z,mw1)
```
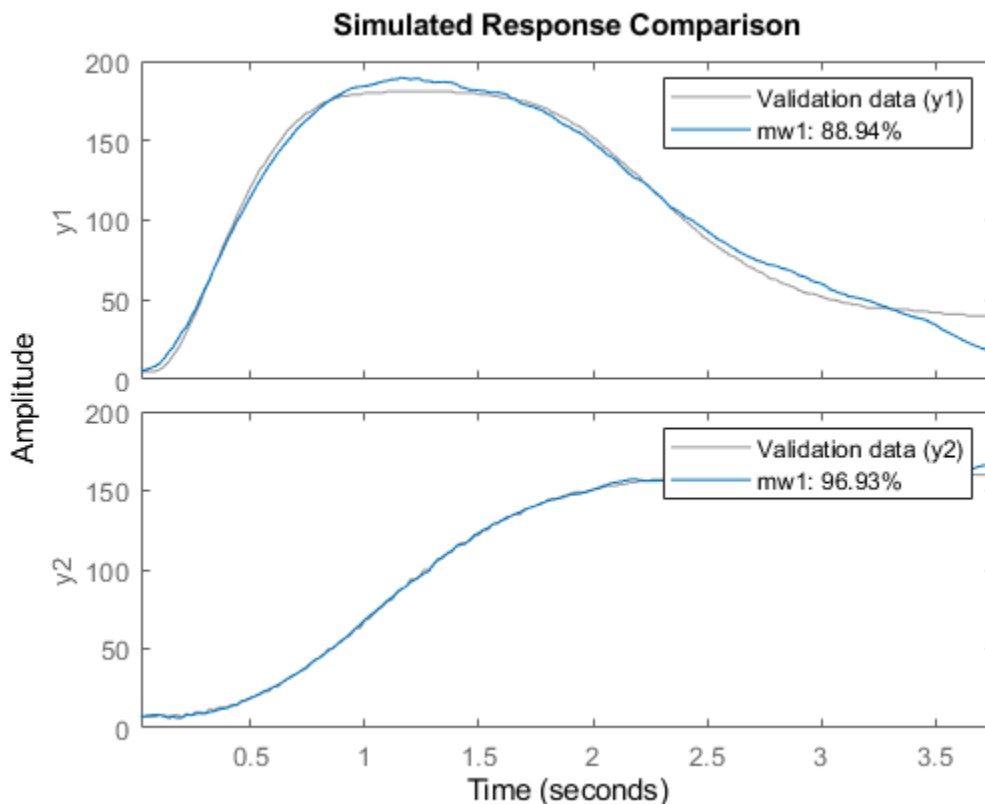

Nonlinear ARX Model - Trying Higher Orders

Let us check if we can do better with higher orders. Note that when identifying models using basis function expansions to express the nonlinearity, the number of model parameters can exceed the number of data samples. In such cases, some estimation metrics such as Noise Variance and Final Prediction Error (FPE) cannot be determined reliably. For the current example, we turn off the warning informing us about this limitation.

```
ws = warning('off','Ident:estimation:NparGTNsamp');
%
mw2 = nlarx(z, [ones(2,2), 2*ones(2,6), ones(2,6)], idWaveletNetwork)
compare(z,mw2)
```

```
mw2 =

Nonlinear ARX model with 2 outputs and 6 inputs
  Inputs: u1, u2, u3, u4, u5, u6
  Outputs: y1, y2

Regressors:
  Linear regressors in variables y1, y2, u1, u2, u3, u4, u5, u6

Output functions:
  Output 1:  Wavelet network with 27 units
  Output 2:  Wavelet network with 25 units

Sample time: 0.02 seconds

Status:
Estimated using NLARX on time domain data "Motorized Camera".
Fit to estimation data: [99.22;99.15]% (prediction focus)
FPE: 0.1262, MSE: 0.4453
More information in model's "Report" property.
```


11-105
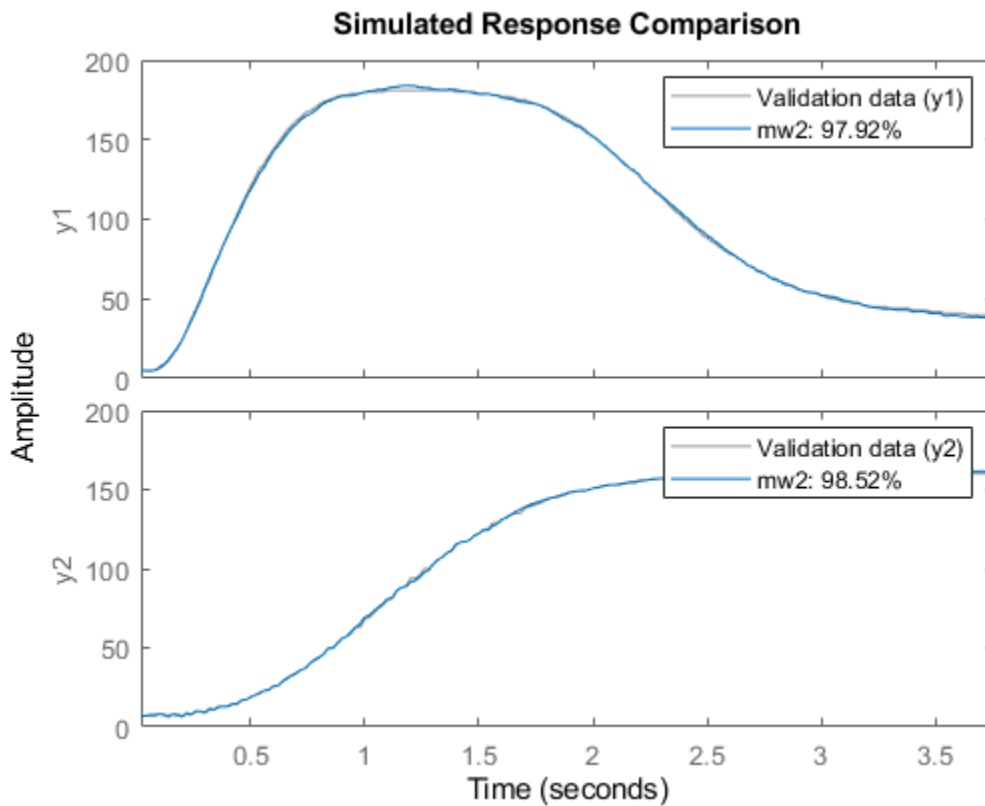
The second model mw2 is pretty good. So let us keep this choice of model orders in the following examples.

```
nanbnk = [ones(2,2), 2*ones(2,6), ones(2,6)]; % final orders
```

To view the regressor set implied by this choice of orders, use the GETREG command:

```
getreg(mw2)
```

```
ans =

  14x1 cell array

    {'y1(t-1)'}
    {'y2(t-1)'}
    {'u1(t-1)'}
    {'u1(t-2)'}
    {'u2(t-1)'}
    {'u2(t-2)'}
    {'u3(t-1)'}
    {'u3(t-2)'}
    {'u4(t-1)'}
    {'u4(t-2)'}
    {'u5(t-1)'}
    {'u5(t-2)'}
    {'u6(t-1)'}
    {'u6(t-2)'}
```

The numbers of units ("wavelets") of the two idWaveletNetwork functions have been automatically chosen by the estimation algorithm. These numbers are displayed below.

```
mw2.OutputFcn(1).NonlinearFcn.NumberOfUnits
mw2.OutputFcn(2).NonlinearFcn.NumberOfUnits
```

```
ans =

    27


ans =

    25
```

Nonlinear ARX Model - Adjusting Number of Units of Nonlinear Functions

The number of units in the idWaveletNetwork function can be explicitly specified instead of being automatically chosen by the estimation algorithm. Suppose we want to use 10 units for the first nonlinear mapping function, and 5 for the second one (note that the model for each output uses its own mapping function; the array of all mapping functions is stored in the model's "OutputFcn" property).

```
Fcn1 = idWaveletNetwork(10); % output function for the first output
Fcn2 = idWaveletNetwork(5);  % output function for the second output
mw3 = nlarx(z, nanbnk, [Fcn1; Fcn2])
```

```
mw3 =

<strong>Nonlinear ARX model with 2 outputs and 6 inputs</strong>
  Inputs: u1, u2, u3, u4, u5, u6
  Outputs: y1, y2

Regressors:
  Linear regressors in variables y1, y2, u1, u2, u3, u4, u5, u6

Output functions:
  Output 1:  Wavelet network with 10 units
  Output 2:  Wavelet network with 5 units

Sample time: 0.02 seconds

Status:
Estimated using NLARX on time domain data "Motorized Camera".
Fit to estimation data: [99.01;98.89]% (prediction focus)
FPE: 0.2273, MSE: 0.7443
More information in model's "Report" property.
```

Nonlinear ARX Model - Trying Other Nonlinear Mapping Functions

In place of the idWaveletNetwork function, other nonlinear functions can also be used. Let us try the idTreePartition for both outputs.

```
mt1 = nlarx(z, nanbnk, idTreePartition);
```

In the above call, a tree partition mapping function with default configuration is used for both the outputs.

Similarly, we can use a Sigmoid Network (idSigmoidNetwork object) as the mapping function. Estimation options such as maximum iterations (MaxIterations) and iteration display can be specified using the nlarxOptions command.

```
opt = nlarxOptions('Display','on');
opt.SearchOptions.MaxIterations = 5;
ms1 = nlarx(z, nanbnk, idSigmoidNetwork, opt);
```

```
Nonlinear ARX Model Estimation
Data has 2 outputs, 6 inputs and 188 samples.
Output function:
(1) Sigmoid network with 10 units
(2) Sigmoid network with 10 units
Model has linear regressors.
Regressors:
v1(t-1), v2(t-1), u1(t-1), u1(t-2), u2(t-1), u2(t-2), u3(t-1), u3(t-2)
```

ESTIMATION PROGRESS

```
Algorithm: Nonlinear least squares with automatically chosen line search method
------------------------------------------------------------------------------
-----
                        Norm of      First-order    Improvement (%)
Iteration     Cost      step         optimality    Expected  Achieved   Bisections
------------------------------------------------------------------------------
-----
     0   1.04083e+07        -           2.78         200         -          -
     1      0.197274     82.1          23.9         200       100          0
     2      0.196901      0.652        23.1         192         0.189      11
     3      0.196834     22.1          28.3         192         0.034       8
     4      0.193319      4.24         28.7         192         1.79        8
     5      0.192328      8.22         33.1         193         0.512       8
------------------------------------------------------------------------------
-----
```

RESULT

```
Termination condition: Maximum number of iterations reached..
Number of iterations: 5, Number of function evaluations: 46

Status: Estimated using NLARX with prediction focus
Fit to estimation data: [98.89;98.83]%, FPE:
```

■ Stop   Close

This calling syntax for NLARX is very similar to the ones used before - specifying the data, the orders and the nonlinear mapping functions as their primary input arguments. However, in order to modify the estimation options from their default values, we constructed an option set, `opt`, using the nlarxOptions command and passed it to the NLARX command as an additional input argument.

Nonlinear ARX Model with Mixed Nonlinear Functions

It is possible to use different nonlinear functions on different output channels in the same model. Suppose we want to use a tree partition function to describe the first output and use a wavelet network for the second output. The model estimation is shown below. The third input argument (the nonlinear mapping function) is now an array of two different functions.

```
Fcn1 = idTreePartition;
Fcn2 = idWaveletNetwork;
mtw = nlarx(z, nanbnk, [Fcn1; Fcn2]);
```

Sometimes the function that maps the regressors to the model output need not be nonlinear; it could simply be a weighted sum of the regressor vectors, plus an optional offset. This is similar to the linear ARX models (except for the offset term). The absence of nonlinearity in an output channel can be indicated by choosing a idLinear function. The following example means that, in model_output(t) = F(y(t-1), u(t-1), u(t-2)), the function F is composed of a linear component for the first output, and a nonlinear component (idSigmoidNetwork) for the second output.

```
Fcn1 = idLinear;
Fcn2 = idSigmoidNetwork(2);
opt.Display = 'off'; % do not show estimation progress anymore
mls = nlarx(z, nanbnk, [Fcn1; Fcn2], opt)
```

```
mls =

Nonlinear ARX model with 2 outputs and 6 inputs
  Inputs: u1, u2, u3, u4, u5, u6
  Outputs: y1, y2

Regressors:
  Linear regressors in variables y1, y2, u1, u2, u3, u4, u5, u6

Output functions:
  Output 1:  Linear with offset
  Output 2:  Sigmoid network with 2 units

Sample time: 0.02 seconds

Status:
Termination condition: Maximum number of iterations reached..
Number of iterations: 5, Number of function evaluations: 60

Estimated using NLARX on time domain data "Motorized Camera".
Fit to estimation data: [98.72;98.8]% (prediction focus)
FPE: 0.5481, MSE: 1.041
More information in model's "Report" property.
```

There is no nonlinearity in the model for the first output. It is not, however, purely linear because of the presence of an offset term.

```
disp(mls.OutputFcn(1))
```

```
Linear Function
Inputs: y1(t-1), y2(t-1), u1(t-1), u1(t-2), u2(t-1), u2(t-2), u3(t-1), u3(t-2), u4(t-1), u4(t-2)
Output: y1(t)

 Nonlinear Function: Linear with offset
 Linear Function: initialized to [48.3 -32.2 -0.229 -0.0706 -0.113 -0.0516 -0.182 0.297 0.199 -0
 Output Offset: initialized to -9.03e-06

     Inputs: {1x14 cell}
    Outputs: {'y1(t)'}
   LinearFcn: '<Linear function parameters>'
```

11-109

```
        Offset: '<Offset parameters>'
```

We have the option of making it purely linear by choosing to not use the offset term. This choice can be made by fixing the offset to a zero value before estimation.

```
Fcn1.Offset.Value = 0;
Fcn1.Offset.Free = false;
mlsNoOffset = nlarx(z, nanbnk, [Fcn1; Fcn2], opt);
```

Using Regression Algorithms from Statistics and Machine Learning Toolbox

If you have access to the Statistics and Machine Learning Toolbox™ you can also use gaussian process (GP) functions and bagged or boosted tree ensembles to create your Nonlinear ARX models. The GPs are represented by the idGaussianProcess object, while regression tree ensembles are represented by the idTreeEnsemble object. GPs use squared-exponential kernel by default. Other kernel types may be specified when creating the object. In the example here, we use a 'ARDMatern52' kernel based GP for the first output, and a bagged tree ensemble for the second output.

```
if exist('fitrgp','file')==2
    warning('off','stats:lasso:MaxIterReached');
    NL1 = idGaussianProcess('ardmatern52');
    NL2 = idTreeEnsemble;
    % estimate model
    mML = nlarx(z, nanbnk, [NL1; NL2])
end


mML =

<strong>Nonlinear ARX model with 2 outputs and 6 inputs</strong>
  Inputs: u1, u2, u3, u4, u5, u6
  Outputs: y1, y2

Regressors:
  Linear regressors in variables y1, y2, u1, u2, u3, u4, u5, u6

Output functions:
  Output 1:  Gaussian process function using a ARDMatern52 kernel
  Output 2:  Bagged Regression Tree Ensemble

Sample time: 0.02 seconds

Status:
Estimated using NLARX on time domain data "Motorized Camera".
Fit to estimation data: [99.29;98.44]% (prediction focus)
MSE: 0.968
More information in model's "Report" property.
```

Inspection of Estimation Results

Various models can be compared in the same COMPARE command. For example, compare the simulated outputs of the models mw2, ms1, mls, and mlsNoOffset to the measured output in dataset "z".

```
compare(z, mw2, ms1, mls, mlsNoOffset)
```
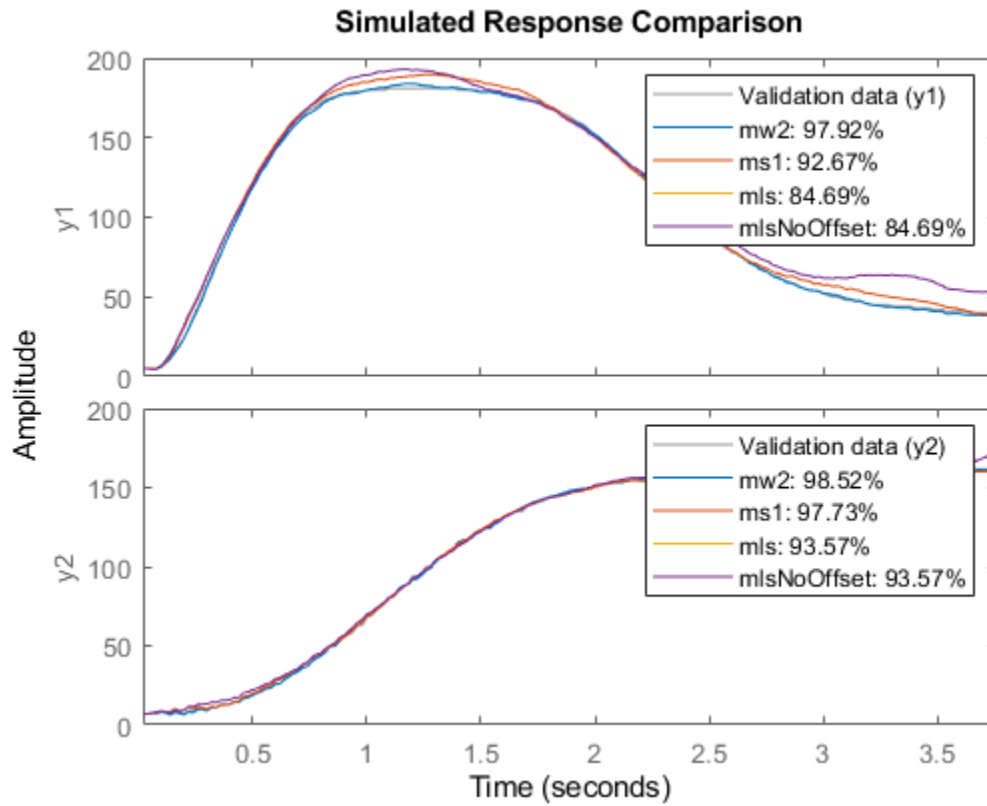
Function PLOT may be used to view the nonlinear mapping function "shape" for various models.

```
plot(mt1,mtw,ms1,mls)
```
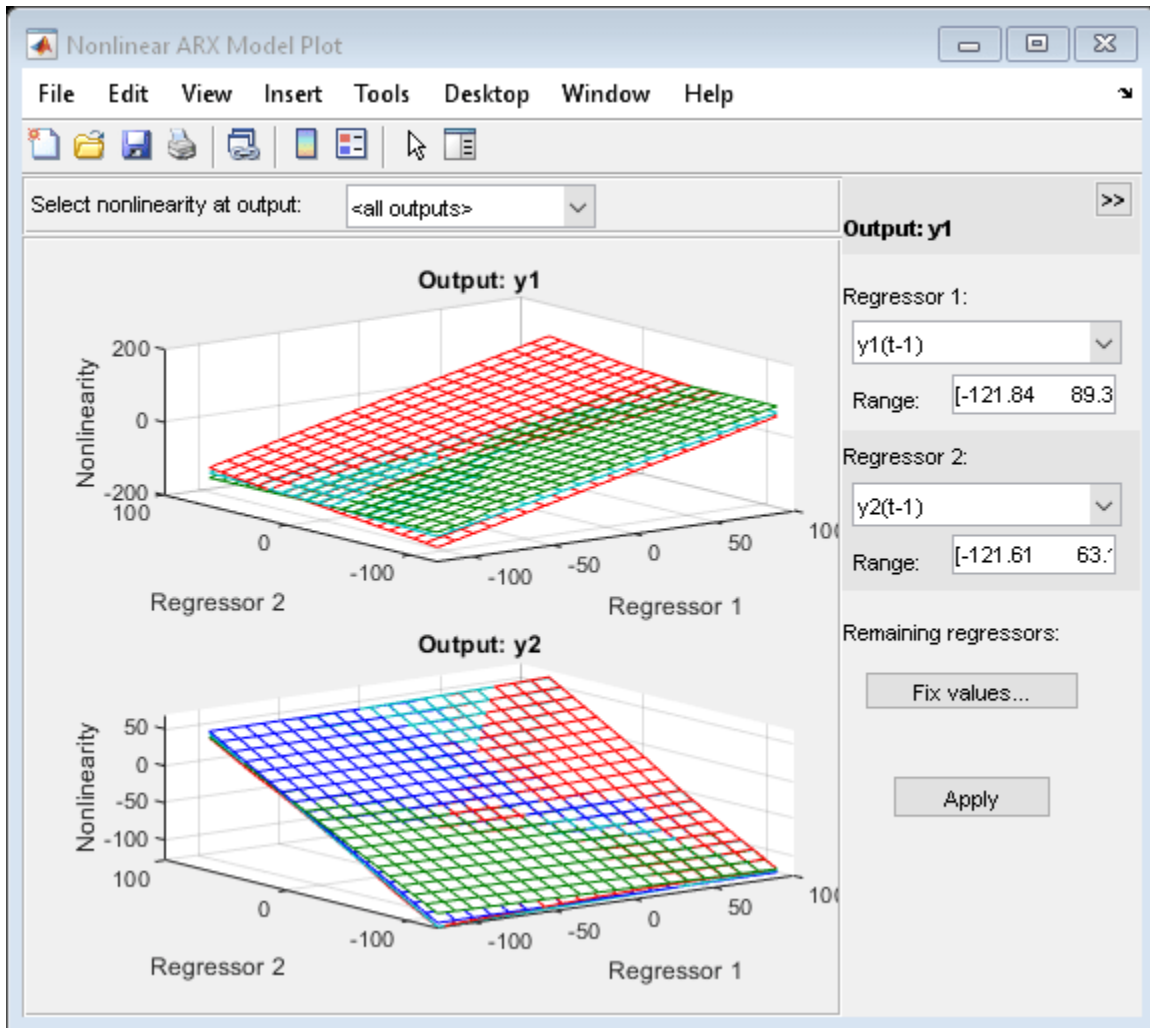
Note that the control panel on the right hand side of the plot allows for regressor selection and configuration.

Other functions such as RESID, PREDICT and PE may be used on the estimated models in the same way as in the case of linear models.

Hammerstein-Wiener (IDNLHW) Model - Preliminary Estimation

A Hammerstein-Wiener model is composed of up to 3 blocks: a linear transfer function block is preceded by a nonlinear static block and/or followed by another nonlinear static block. It is called a Wiener model if the first nonlinear static block is absent, and a Hammerstein model if the second nonlinear static block is absent.

IDNLHW models are typically estimated using the syntax:

`M = NLHW(Data, Orders, InputNonlinearity, OutputNonlinearity).`

where Orders = [nb nf nk] specifies the orders and delay of the linear transfer function. InputNonlinearity and OutputNonlinearity specify the nonlinear functions for the two nonlinear blocks. The linear output error (OE) model corresponds to the case of trivial (unit gain mapping) nonlinearities.

Estimation of Hammerstein Model (No Output Nonlinearity)

Let us choose Orders = [nb bf nk] = [ones(2,6), ones(2,6), ones(2,6)]. It means that, in the linear block, each output is the sum of 6 first order transfer functions driven by the 6 inputs. Try a Hammerstein model (no output nonlinearity) with the input nonlinearity described by a piecewise linear function. Notice that the entered single idPiecewiseLinear object is automatically expanded to all the 6 input channels. idUnitGain indicates absence of nonlinearity.

```
opt = nlhwOptions();
opt.SearchOptions.MaxIterations = 15;
NN = [ones(2,6), ones(2,6), ones(2,6)];
InputNL  = idPiecewiseLinear;  % input nonlinearity
OutputNL = idUnitGain;  % output nonlinearity
mhw1 = nlhw(z, NN, InputNL, OutputNL, opt)
```

```
mhw1 =

<strong>Hammerstein-Wiener model with 2 outputs and 6 inputs</strong>

Linear transfer function matrix corresponding to the orders:
   nb = [1 1 1 1 1 1; 1 1 1 1 1 1]
   nf = [1 1 1 1 1 1; 1 1 1 1 1 1]
   nk = [1 1 1 1 1 1; 1 1 1 1 1 1]

Input nonlinearities:
  Input 1:  Piecewise linear with 10 break-points
  Input 2:  Piecewise linear with 10 break-points
  Input 3:  Piecewise linear with 10 break-points
  Input 4:  Piecewise linear with 10 break-points
  Input 5:  Piecewise linear with 10 break-points
  Input 6:  Piecewise linear with 10 break-points

Output nonlinearities:
  Output 1:  None
  Output 2:  None

Sample time: 0.02 seconds

Status:
Termination condition: Maximum number of iterations reached..
Number of iterations: 15, Number of function evaluations: 297

Estimated using NLHW on time domain data "Motorized Camera".
Fit to estimation data: [98.46;97.93]%
FPE: 7.931, MSE: 2.217
More information in model's "Report" property.
```

Examine the result with COMPARE.

```
compare(z, mhw1);
```
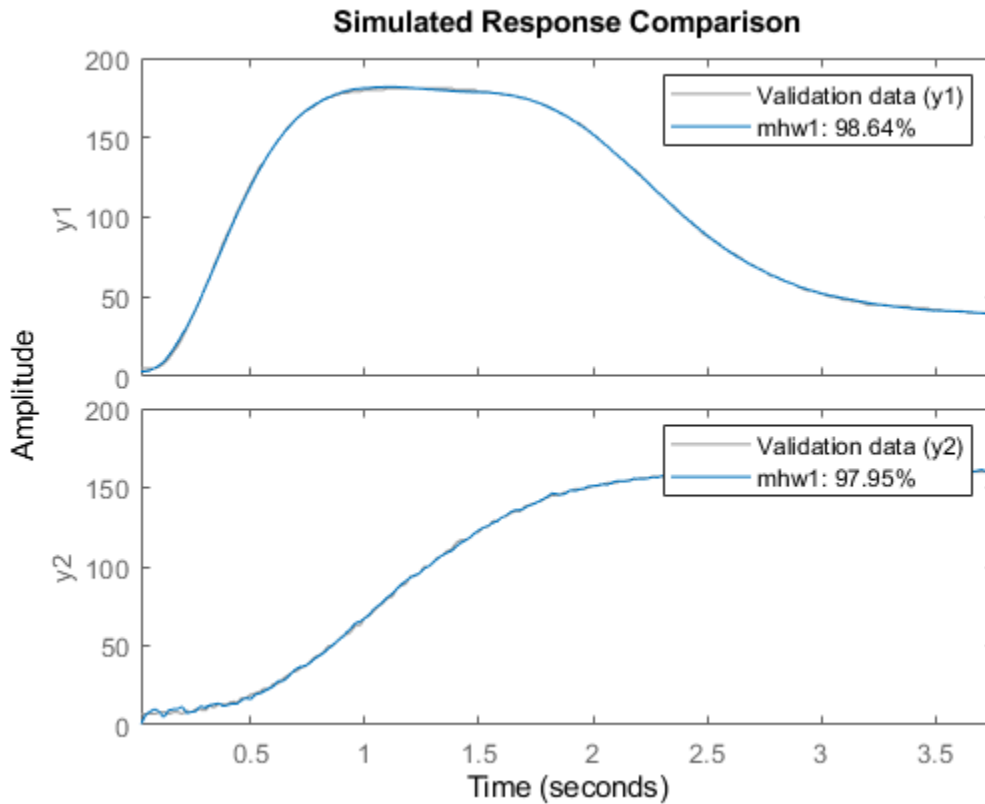
Model properties can be visualized by the PLOT command.

Click on the block-diagram to choose the view of the input nonlinearity (UNL), the linear block or the output nonlinearity (YNL).

When the linear block view is chosen, by default all the 12 channels are shown in very reduced sizes. Choose one of the channels to have a better view. It is possible to choose the type of plots within Step response, Bode plot, Impulse response and Pole-zero map.

```
plot(mhw1)
```
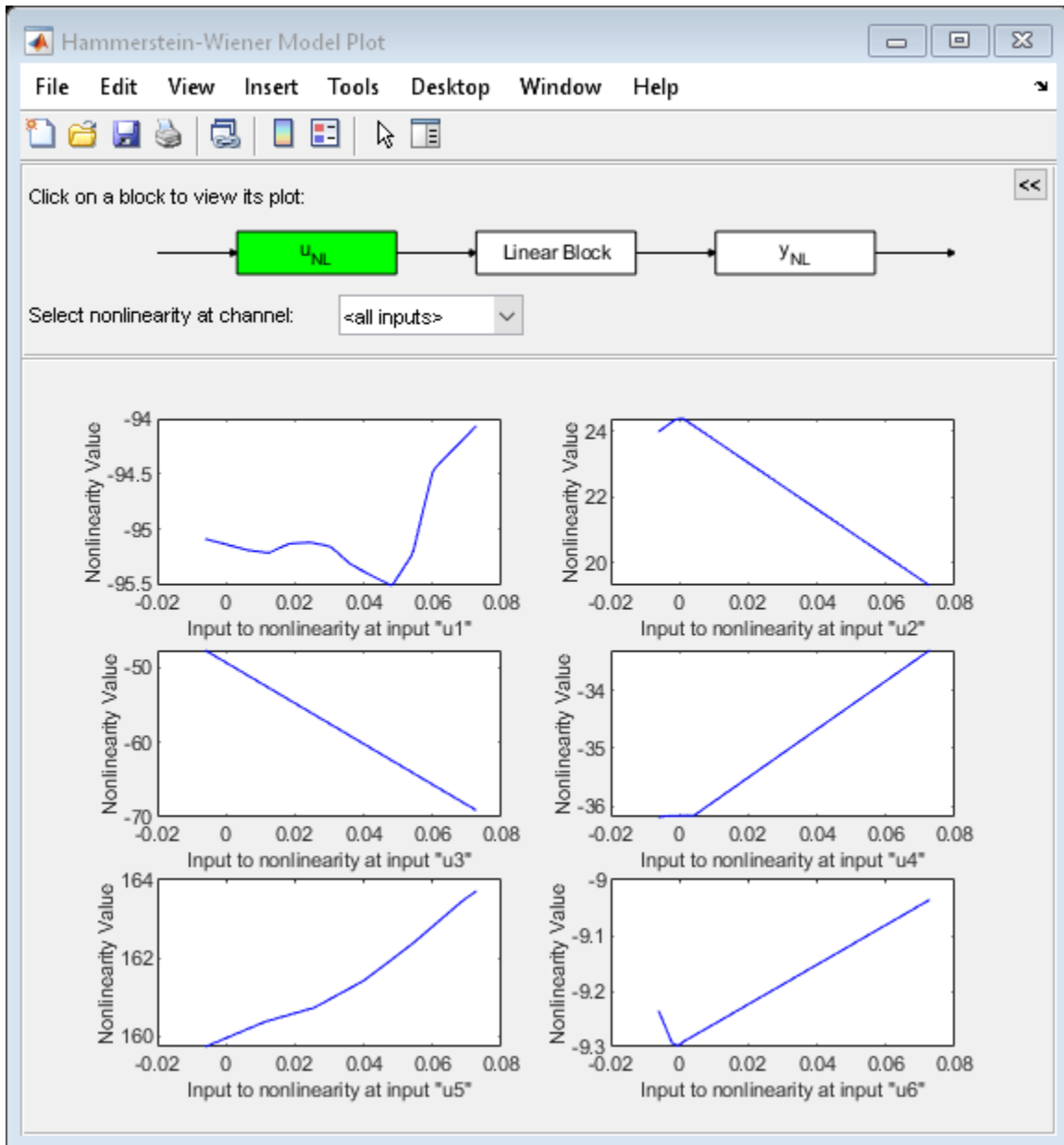
The result shown by COMPARE was quite good, so let us keep the same model orders in the following trials.

```
nbnfnk = [ones(2,6), ones(2,6), ones(2,6)];
```

Estimation of Wiener Model (No Input Nonlinearity)

Let us try a Wiener model. Notice that the absence of input nonlinearity can be indicated by "[]" instead of the idUnitGain object.

```
opt.SearchOptions.MaxIterations = 10;
mhw2 = nlhw(z, nbnfnk, [], idPiecewiseLinear, opt)
```

```
mhw2 =

<strong>Hammerstein-Wiener model with 2 outputs and 6 inputs</strong>

Linear transfer function matrix corresponding to the orders:
   nb = [1 1 1 1 1 1; 1 1 1 1 1 1]
   nf = [1 1 1 1 1 1; 1 1 1 1 1 1]
   nk = [1 1 1 1 1 1; 1 1 1 1 1 1]

Input nonlinearities:
  Input 1:  None
  Input 2:  None
  Input 3:  None
  Input 4:  None
  Input 5:  None
  Input 6:  None

Output nonlinearities:
  Output 1:  Piecewise linear with 10 break-points
  Output 2:  Piecewise linear with 10 break-points

Sample time: 0.02 seconds

Status:
Termination condition: Maximum number of iterations reached..
Number of iterations: 10, Number of function evaluations: 331

Estimated using NLHW on time domain data "Motorized Camera".
Fit to estimation data: [73.85;71.36]%
FPE: 1.314e+05, MSE: 503.8
More information in model's "Report" property.
```

Estimation of Hammerstein-Wiener Model (Both Input and Output Nonlinearities)

Indicate both input and output nonlinearities for a Hammerstein-Wiener model.

```
mhw3 = nlhw(z, nbnfnk,idSaturation, idPiecewiseLinear, opt)


mhw3 =

<strong>Hammerstein-Wiener model with 2 outputs and 6 inputs</strong>

Linear transfer function matrix corresponding to the orders:
   nb = [1 1 1 1 1 1; 1 1 1 1 1 1]
   nf = [1 1 1 1 1 1; 1 1 1 1 1 1]
   nk = [1 1 1 1 1 1; 1 1 1 1 1 1]

Input nonlinearities:
  Input 1:  Saturation with linear interval: [0.0103 0.0562]
  Input 2:  Saturation with linear interval: [-0.00143 0.000909]
  Input 3:  Saturation with linear interval: [-0.0947 -0.0185]
  Input 4:  Saturation with linear interval: [-0.00385 0.00527]
  Input 5:  Saturation with linear interval: [0.0195 0.13]
  Input 6:  Saturation with linear interval: [-0.00302 0.000387]

Output nonlinearities:
```

```
  Output 1:  Piecewise linear with 10 break-points
  Output 2:  Piecewise linear with 10 break-points

Sample time: 0.02 seconds

Status:
Termination condition: Maximum number of iterations reached..
Number of iterations: 10, Number of function evaluations: 341

Estimated using NLHW on time domain data "Motorized Camera".
Fit to estimation data: [86.88;84.55]%
FPE: 1.111e+04, MSE: 137.3
More information in model's "Report" property.
```

The limit values of the idSaturation function can be accessed as follows:

```
mhw3.InputNonlinearity(1).LinearInterval % view Linear Interval of saturation function
```

```
ans =

    0.0103    0.0562
```

Similarly, the break points of the idPiecewiseLinear function can be accessed as follows:

```
mhw3.OutputNonlinearity(1).BreakPoints
```

```
ans =

  Columns 1 through 7

   17.1233   34.2491   51.3726   68.4968   85.6230  102.7478  119.8742
    2.6184   16.0645   45.5178   41.9621   62.3246   84.9038  112.2970

  Columns 8 through 10

  136.9991  154.1238  171.2472
  135.4543  156.1016  173.2701
```

Hammerstein-Wiener Model - Using Mixed Nonlinear Functions

Different nonlinear functions can be mixed in a same model. Suppose we want a model with: - No nonlinearity on any output channels ("Hammerstein Model") - Piecewise linear nonlinearity on input channel #1 with 3 units - Saturation nonlinearity on input channel #2 - Dead Zone nonlinearity on input channel #3 - Sigmoid Network nonlinearity on input channel #4 - No nonlinearity (specified by a unitgain object) on input channel #5 - Sigmoid Network nonlinearity on input channel #6 with 5 units

We can create an array of nonlinear mapping function objects of chosen types and pass it to the estimation function NLHW as input nonlinearity.

```
inputNL = [idPiecewiseLinear; ...
    idSaturation; ...
    idDeadZone; ...
    idSigmoidNetwork; ...
```

```
        idUnitGain; ...
        idSigmoidNetwork(5)];
inputNL(1).NumberOfUnits = 3;
opt.SearchOptions.MaxIterations = 25;
mhw4 = nlhw(z, nbnfnk, inputNL, [], opt); % "[]" as the 4th input argument  denotes no output nor
```

Hammerstein-Wiener Model - Specifying Initial Guess for SATURATION and DEADZONE

The initial guess for the linear interval of saturation and the zero interval of dead zone can be directly indicated when these objects are created; you can also specify constraints on these values such as whether they are fixed to their specified values (by setting Free attribute to false), or if their estimations are subject to minimum/maximum bounds (using the Minimum and Maximum attributes).

Suppose we want to set the saturation's linear interval to [10 200] and the deadzone's zero interval to [11 12] as initial guesses. Furthermore, we want the upper limit of the saturation to remain fixed. We can achieve this configuration as follows.

```
% Create  nonlinear functions with initial guesses for properties.
OutputNL1 = idSaturation([10 200]);
OutputNL1.Free(2) = false; % the upper limit is a fixed value
OutputNL2 = idDeadZone([11 12]);

mhw5 = idnlhw(nbnfnk, [], [OutputNL1; OutputNL2], 'Ts',z.Ts);
```

Notice the use of the IDNLHW model object constructor `idnlhw`, not the estimator `nlhw`. The resulting model object `mhw5` is not yet estimated from data. The limit values of the saturation and deadzone functions can be accessed. They still have their initial values, because they are not yet estimated from data.

```
mhw5.OutputNonlinearity(1).LinearInterval % view the linear interval on saturation at first outpu
mhw5.OutputNonlinearity(2).ZeroInterval   % view the zero interval on dead zone at second output


ans =

    10   200


ans =

    11    12
```

What are the values of these limits after estimation?

```
opt.SearchOptions.MaxIterations = 15;
mhw5 = nlhw(z, mhw5, opt)  % estimate the model from data
mhw5.OutputNonlinearity(1).LinearInterval % show linear interval on saturation at first output cl
mhw5.OutputNonlinearity(2).ZeroInterval   % show zero interval on dead zone at second output ch


mhw5 =

<strong>Hammerstein-Wiener model with 2 outputs and 6 inputs</strong>

Linear transfer function matrix corresponding to the orders:
    nb = [1 1 1 1 1 1; 1 1 1 1 1 1]
    nf = [1 1 1 1 1 1; 1 1 1 1 1 1]
```

```
   nk = [1 1 1 1 1 1; 1 1 1 1 1 1]

Input nonlinearities:
  Input 1:  None
  Input 2:  None
  Input 3:  None
  Input 4:  None
  Input 5:  None
  Input 6:  None

Output nonlinearities:
  Output 1:  Saturation with linear interval: [10 200]
  Output 2:  Dead Zone with zero interval: [11 12]

Sample time: 0.02 seconds

Status:
Termination condition: Maximum number of iterations reached..
Number of iterations: 15, Number of function evaluations: 386

Estimated using NLHW on time domain data "Motorized Camera".
Fit to estimation data: [27.12;6.857]%
FPE: 3.373e+06, MSE: 4666
More information in model's "Report" property.

ans =

    9.9974  200.0000


ans =

   11.0020   12.0011
```

Post Estimation Analysis - Comparing Different Models

Models of different nature (IDNLARX and IDNLHW) can be compared in the same COMPARE command.

```
compare(z,mw2,mhw1)
```

```matlab
warning(ws) % reset the warning state
```
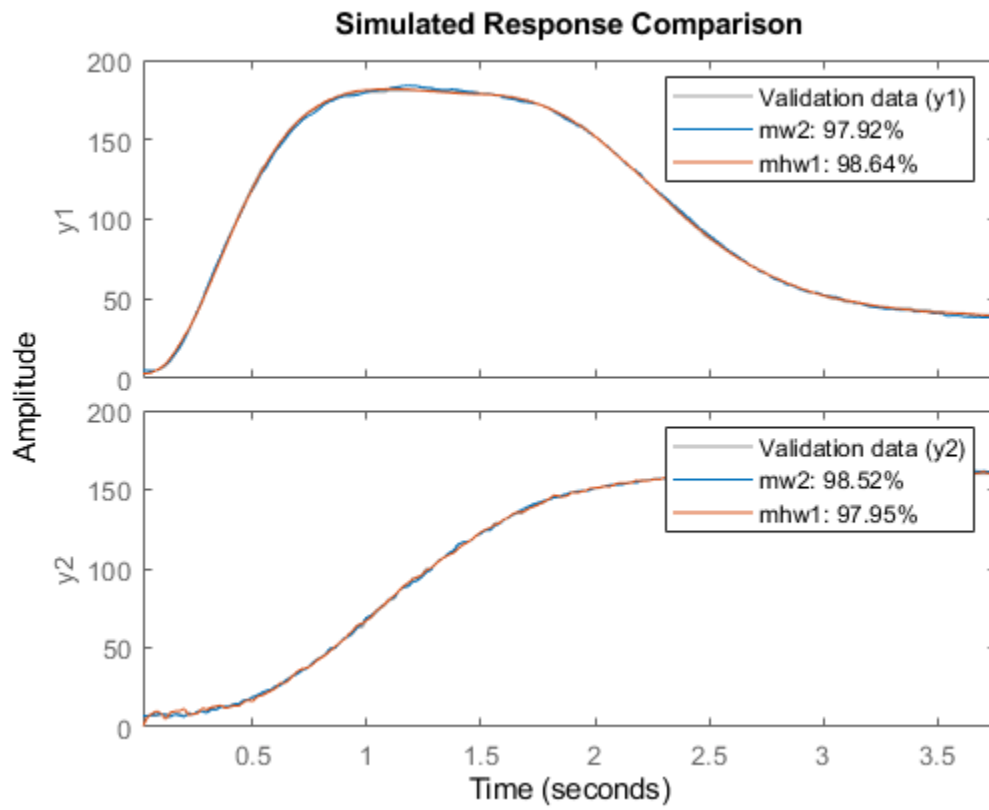
Building Nonlinear ARX Models with Nonlinear and Custom Regressors

This example shows how to use polynomial and custom regressors in Nonlinear ARX (IDNLARX) models.

Introduction

In an IDNLARX model, each output is a function of regressors which are transformations of past inputs and past outputs. Typical regressors are simply delayed input or output variables represented by the **linearRegressor** specification object, or polynomials of the delayed variables represented by the **polynomialRegressor** object. However, other than the ability to impose absoute value (e.g., *abs(y(t-1))*), you cannot create complex mathematical formulas using these types of regressors. For example, if your output function requires a regressor of the form *sin(u(t-3)).*exp(-abs(u(t)))*, you need a way to type-in your custom formula in the expression for the regressor. This is faciliated by the **customRegressor** objects. As the name suggests, you use the customRegressor object to incorporate arbitrary, custom formulas as regressors for your Nonlinear ARX model.

Consider the example of an electric system with both the voltage **V** and the current **I** as inputs. If it is known that the electric power is an important quantity of the system, then it makes sense to form the custom regressor **V*I**. It may be more efficient to use appropriately defined custom regressors than to use the linear and polynomial regressors only.

SISO Example: Modeling an Internal Combustion Engine

The file icEngine.mat contains one data set with 1500 input-output samples collected at the a sampling rate of 0.04 seconds. The input u(t) is the voltage [V] controlling a By-Pass Idle Air Valve (BPAV), and the output y(t) is the engine speed [RPM/100]. The data is loaded and split into a dataset **ze** for model estimation and another dataset **zv** for model validation.

```
load icEngine
z = iddata(y,u,0.04);
ze = z(1:1000);
zv = z(1001:1500);
plot(ze,zv)
legend('Estimation data','Validation data')
```
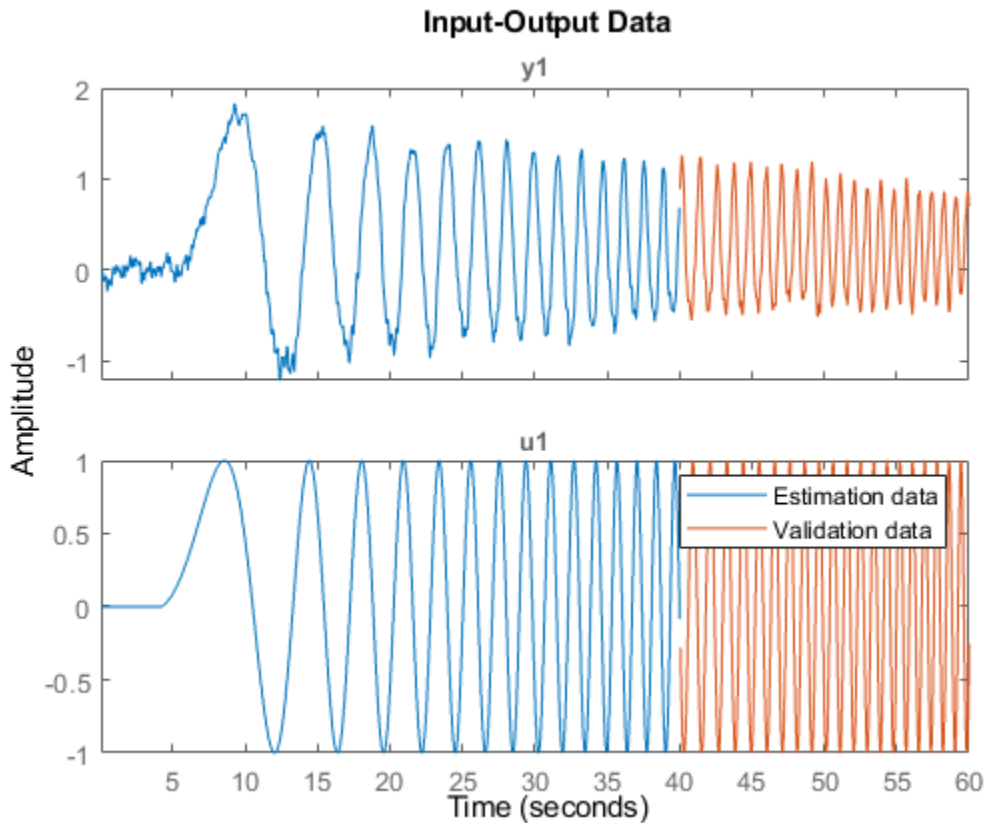
Input-Output Data

Linear Regressors as ARX Orders

Order matrix [na nb nk], which is also used in the linear ARX model, help easily define the linear regressors, which are simply the input/output variables delayed by certain number of samples. The choice of model orders requires trial and error. For this example let us use [na nb nk] = [4 2 10], corresponding to the linear regressors y(t-1), y(t-2), y(t-3), y(t-4), u(t-10), u(t-11). Choose a linear output function, so that the model output is just a weighted sum of its six regressors, plus an offset.
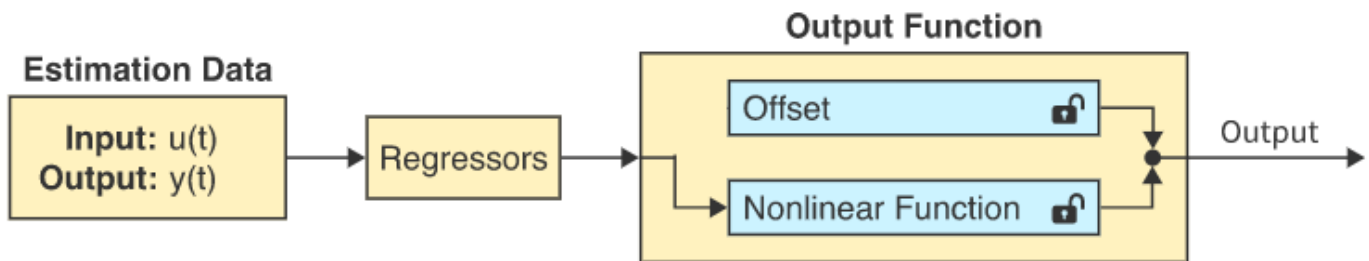

```
sys0 = nlarx(ze, [4 2 10], idLinear);
```

The input name, output name and the list of regressors of this model are displayed below. Notice that the default names 'u1', 'y1' are used.

```
sys0.InputName
```

```
ans = 1×1 cell array
    {'u1'}
```

```
sys0.OutputName
```

```
ans = 1×1 cell array
    {'y1'}
```

```
disp(getreg(sys0))
```

```
    {'y1(t-1)' }
    {'y1(t-2)' }
    {'y1(t-3)' }
    {'y1(t-4)' }
    {'u1(t-10)'}
    {'u1(t-11)'}
```

These are *linear* regressors. They are stored in the model's **Regressors** property as a
linearRegressor object.

```
sys0.Regressors
```

```
ans =
Linear regressors in variables y1, u1
      Variables: {'y1'  'u1'}
           Lags: {[1 2 3 4]  [10 11]}
    UseAbsolute: [0 0]
    TimeVariable: 't'

  Regressors described by this set
```

The "Regressors" property stores the information on the regressor implicitly using a regression
specification object. You can think of the order matrix [4 2 10] as a shortcut way of specifying linear
regressors, where are lags are contiguous and minimum lag in the output variable is fixed to 1.

Linear Regressors Using linearRegressor Specification Object

A more flexible way that allows picking of variables with arbitrary lags is to use the **linearRegressor**
object. In the above configuration, the variable y1 has lags [1 2 3 4], while the variable u1 has lags
[10 11]. Using a linearRegressor object, the same configuration can be achieved as follows:

```
LinReg = linearRegressor({'y1','u1'}, {1:4, [10, 11]});
sys1 = nlarx(ze, LinReg, idLinear)
```

```
sys1 =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  Linear regressors in variables y1, u1
  List of all regressors

Output function: None
Sample time: 0.04 seconds

Status:
```

```
Estimated using NLARX on time domain data "ze".
Fit to estimation data: 94.35% (prediction focus)
FPE: 0.001877, MSE: 0.00183
```

Compare the estimation syntaxes of the models `sys1` and `sys0`; when creating `sys1` you replaced the order matrix with a linear regressor specification object. The resulting models `sys0` and `sys1` are identical.

```
[getpvec(sys0), getpvec(sys1)] % estimated parameter vectors of sys0 and sys1
```

```
ans = 7×2

    0.7528    0.7528
    0.0527    0.0527
   -0.0621   -0.0621
   -0.0425   -0.0425
   -0.0165   -0.0165
   -0.0289   -0.0289
    0.2723    0.2723
```

You will use the linearRegressor object to specify the regressors in place of the order matrix, in the following scenarios:

- you want to use arbitrary (non-contiguous) lags in variables such as the set $\{y1(t-1), y1(t-10), u(t-3), u(t-11)\}$
- you want the minimum lag in the output variable(s) to be different than 1, for example, the set $\{y1(t-4), y1(t-5), \ldots\}$
- you want to use absolute values of the variables such as in the set $\{|y1(t-1)|, |y1(t-10)|, u(t), u(t-2)\}$ where only the absolute value of variable 'y1' is used. This can be achieved by doing

```
LinRegWithAbs = linearRegressor({'y1','u1'},{[1 10], [0 2]},[true, false])
```

```
LinRegWithAbs =
Linear regressors in variables y1, u1
       Variables: {'y1'  'u1'}
            Lags: {[1 10]  [0 2]}
     UseAbsolute: [1 0]
    TimeVariable: 't'

  Regressors described by this set
```

Polynomial Regressors

Often regressors that are polynomials of delayed I/O variables are required. These can be added to the model using the **polynomialRegressor** object. Suppose you want to add $u1(t-10)^2$, $y1(t-1)^2$ as regressors to the model. You first create a specification object with these settings and then add this object to the model.

```
% Create order 2 regressors in variables 'u1' and 'y1', with lags 10 and 1
% respectively
PolyReg = polynomialRegressor({'u1','y1'},{10, 1},2);

% Use LinReg and PolyReg in the model
sys2 = nlarx(ze, [LinReg; PolyReg], idLinear)
```

```
sys2 =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  1. Linear regressors in variables y1, u1
  2. Order 2 regressors in variables u1, y1
  List of all regressors

Output function: None
Sample time: 0.04 seconds

Status:
Estimated using NLARX on time domain data "ze".
Fit to estimation data: 94.47% (prediction focus)
FPE: 0.001804, MSE: 0.001752
```

Regressors Based on Custom Formulas

While the linear and polynomial regressors are most commonly used, sometimes you need to use a different formula that is not described by a polynomial. An example is trignometric functions such as $sin(), cos(), ...$ Another example is the saturation function $x = max(LowerBound, min(x, UpperBound))$. In these situations, you can use an array of customRegressor objects that encapsulate a specific mathematical expression.

In the following example, a regressor is created as the cosine function of the variable named 'u1' and delayed 10 samples, in other words: $x = cos(u_1(t - 10))$. The logic value at the last input argument indicates if the custom regressor is vectorized or not. Vectorized regressors are faster in computations, but require cares in the function indicated at the first input argument.

```
x = customRegressor('u1', 10, @cos)

x =
Custom regressor: cos(u1(t-10))
    VariablesToRegressorFcn: @cos
                  Variables: {'u1'}
                       Lags: {[10]}
                 Vectorized: 1
               TimeVariable: 't'

  Regressors described by this set
```

The specified formula (@cos here) is stored in the `VariablesToRegressorFcn` property of the regressor object. By default, the evaluation of the function is assumed to be vectorized, that is if the input is a matrix with N rows then the output of the function `x.VariablesToRegressorFcn` would be a column vector of length N. Vectorization helps speed up the evaluation of the regressor during the model estimation and simulation process. However, you have the option of disabling it if desired by setting the value of the `Vectorized` property to FALSE.

You can create an array of regressors all sharing the same underlying formula but using different lag values. For example, suppose the formula is $x = u(t - a) * |y(t - b)|$, where 'u' and 'y' are two variables for which measured data is available, and the lags (a,b) can take multiple values over the range 1:10. You can create an array of these regressors with a single call to the `customRegressor` constructor

```
C = customRegressor({'u','y'},{1:10, 1:10}, @(x,y)x.*abs(y))
```

```
C =
Custom regressor: @(x,y)x.*abs(y)
    VariablesToRegressorFcn: @(x,y)x.*abs(y)
                  Variables: {'u'  'y'}
                       Lags: {[1 2 3 4 5 6 7 8 9 10]  [1 2 3 4 5 6 7 8 9 10]}
                 Vectorized: 1
               TimeVariable: 't'

  Regressors described by this set
```

C represents a set of 100 regressors generated by using the formula $u(t - a) .* abs(y(t - b))$ for all combinations of a and b values in the range 1:10.

Using All Three Types of Regressors in the Model for IC engine Dynamics

Suppose trial/error or physical insight suggests that we need to use the regressor set $R = \{u1(t - 10)^3, u1(t - 11)^3, y1(t - 1), sin(y1(t - 4))\}$ in the model. We have a mix of linear, polynomial and trigonometric formulas. We proceed as follows:

```
LinReg    = linearRegressor('y1',1);
PolyReg   = polynomialRegressor('u1',[10 11],3);
CustomReg = customRegressor('y1',4,@(x)sin(x));
% for now no nonlinearity; output is a linear function of regressors
sys3 = nlarx(ze,[LinReg; PolyReg; CustomReg], [])

sys3 =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  1. Linear regressors in variables y1
  2. Order 3 regressors in variables u1
  3. Custom regressor: sin(y1(t-4))
  List of all regressors

Output function: None
Sample time: 0.04 seconds

Status:
Estimated using NLARX on time domain data "ze".
Fit to estimation data: 92.11% (prediction focus)
FPE: 0.003647, MSE: 0.00357

getreg(sys3)

ans = 4×1 cell
    {'y1(t-1)'     }
    {'u1(t-10)^3'  }
    {'u1(t-11)^3'  }
    {'sin(y1(t-4))'}
```

We can extend this workflow to include nonlinear mapping functions, such as Sigmoid Network in the model and also designate only a subset of the regressor set to be used as inputs to its linear and nonlinear components (note: a Sigmoid network is a sum of 3 components - a linear function, an offset term, and a nonlinear function that is a sum of sigmoid units). In the following example, we use a

template-model based workflow wherein we serapately prepare a template IDNLARX model and estimation option set before using them in the NLARX command for parameter estimation.

```
sys4 = idnlarx(ze.OutputName, ze.InputName, [4 2 10], idSigmoidNetwork);
% generate 'u1(t-10)^2', 'y1(t-1)^2', 'u1(t-10)*y1(t-1)'
P = polynomialRegressor({'y1','u1'},{1, 10},2, false, true);
% generate cos(u1(t-10))
C1 = customRegressor('u1',10,@cos);
% generate sin(y1(t-1).*u1(t-10)+u1(t-11))
C2 = customRegressor({'y1','u1','u1'},{1, 10, 11},@(x,y,z)sin(x.*y+z));

% add the polynomial and custom regressors to the model sys2
sys4.Regressors = [sys4.Regressors; P; C1; C2];

% view the regressors and how they are used in the model
disp(sys4.RegressorUsage)
```

	y1:LinearFcn	y1:NonlinearFcn
y1(t-1)	true	true
y1(t-2)	true	true
y1(t-3)	true	true
y1(t-4)	true	true
u1(t-10)	true	true
u1(t-11)	true	true
y1(t-1)^2	true	true
u1(t-10)^2	true	true
cos(u1(t-10))	true	true
sin(y1(t-1).*u1(t-10)+u1(t-11))	true	true

```
% designate only the linear regressors to be used in the nonlinear
% component of the sigmoid network
Usage = sys4.RegressorUsage;
Usage{7:end,2} = false;
sys4.RegressorUsage = Usage;
disp(sys4.RegressorUsage)
```

	y1:LinearFcn	y1:NonlinearFcn
y1(t-1)	true	true
y1(t-2)	true	true
y1(t-3)	true	true
y1(t-4)	true	true
u1(t-10)	true	true
u1(t-11)	true	true
y1(t-1)^2	true	false
u1(t-10)^2	true	false
cos(u1(t-10))	true	false
sin(y1(t-1).*u1(t-10)+u1(t-11))	true	false

```
% Prepapre estimation options: use Levenberg-Marquardt solver with 30 maximum iterations.
% Turn progress display on and set estimation focus to 'simulation'
opt = nlarxOptions;
opt.Focus = 'simulation';
opt.Display = 'on';
```

```
opt.SearchMethod = 'lm';
opt.SearchOptions.MaxIterations = 30;

% estimate parameters of sys4 to fit data ze
sys4 = nlarx(ze, sys4, opt)

sys4 =
Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  1. Linear regressors in variables y1, u1
  2. Order 2 regressors in variables y1, u1
  3. Custom regressor: cos(u1(t-10))
  4. Custom regressor: sin(y1(t-1).*u1(t-10)+u1(t-11))
  List of all regressors

Output function: Sigmoid network with 10 units
Sample time: 0.04 seconds

Status:
Estimated using NLARX on time domain data "ze".
Fit to estimation data: 87.81% (simulation focus)
FPE: 0.001491, MSE: 0.008517
```

We can now validate the models by comparing their responses to the output of the validata dataset zv.

```
compare(zv, sys1, sys2, sys3, sys4)
```
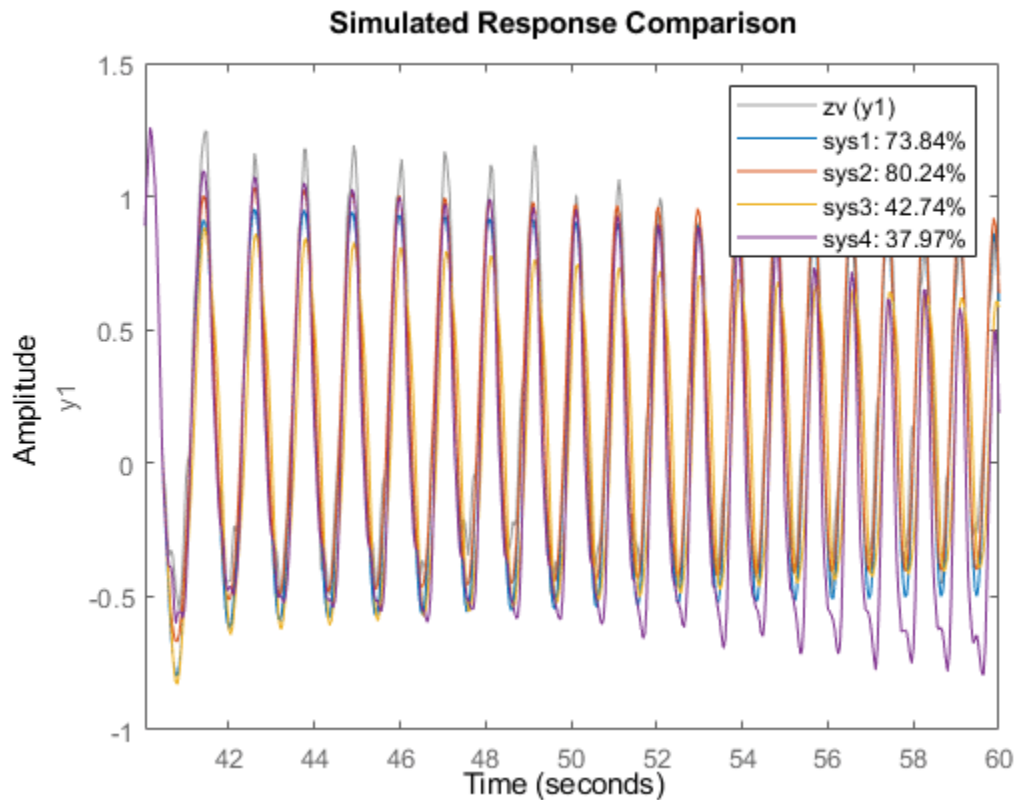
The compare plots indicates `sys2` as the best model. Note that the regressors used in this example were chosen arbitrarily, mainly to show the different way of creating regressors and estimating models. Better results can be obtained by picking regressors more judiciously.

MIMO Example: Modeling a Motorized Camera

The file `motorizedcamera.mat` contains one data set with 188 data samples, collected from a motorized camera at a sampling rate of 0.02 second. The input vector u(t) is composed of 6 variables: the 3 translation velocity components in the orthogonal X-Y-Z coordinate system fixed to the camera [m/s], and the 3 rotation velocity components around the X-Y-Z axis [rad/s]. The output vector y(t) contains 2 variables: the position (in pixel) of a point which is the image taken by the camera of a fixed point in the 3D space. We create an IDDATA object z to hold the loaded data:

```
load motorizedcamera
z = iddata(y, u, 0.02, 'Name', 'Motorized Camera', 'TimeUnit', 's');
plot(z)
```

Input-Output Data
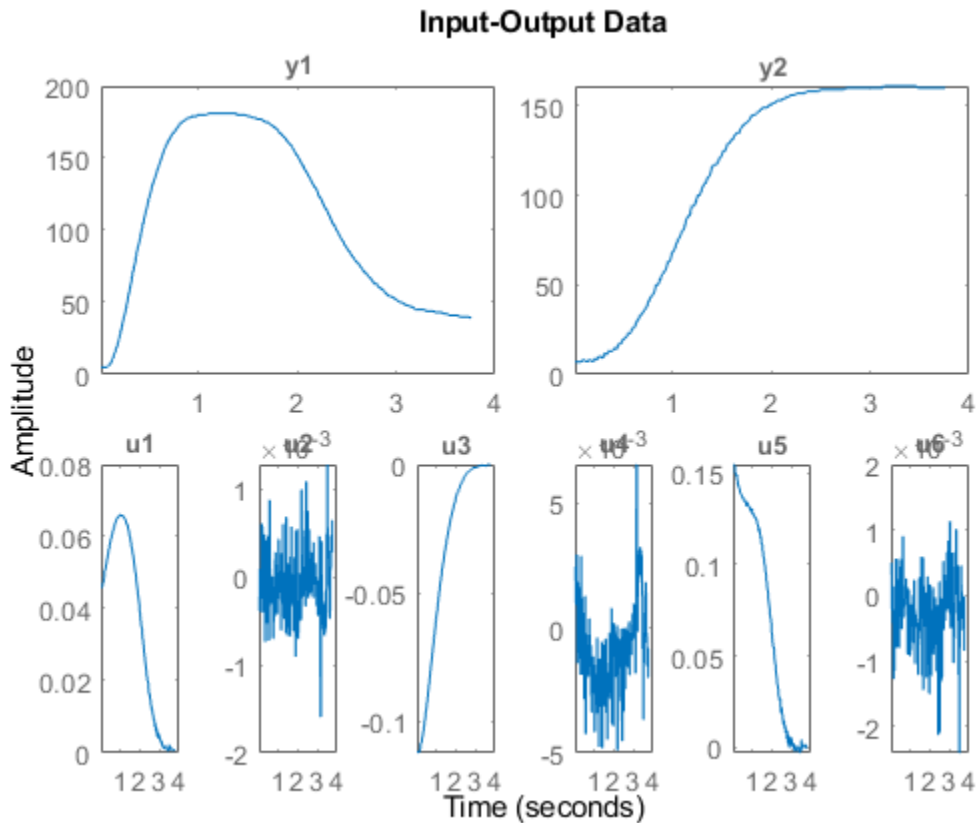
Using different types of regressors in the MIMO case is not very different from the SISO case. All the regressors are used in the dynamics for all the outputs of the system. The `RegressorUsage` property may be used to assign specific regressors to be used for specific outputs.

```
nanbnk = [ones(2,2), 2*ones(2,6), ones(2,6)];
sysMIMO = idnlarx(z.OutputName, z.InputName, nanbnk, idLinear);
C = customRegressor({'u5','u6'},{1 1},@(x,y)x.*y);
P1 = polynomialRegressor('u1',1,2);
P2 = polynomialRegressor('y2',1,3);
sysMIMO.Regressors = [sysMIMO.Regressors; C; P1; P2];
getreg(sysMIMO)
```

```
ans = 17×1 cell
    {'y1(t-1)'          }
    {'y2(t-1)'          }
    {'u1(t-1)'          }
    {'u1(t-2)'          }
    {'u2(t-1)'          }
    {'u2(t-2)'          }
    {'u3(t-1)'          }
    {'u3(t-2)'          }
    {'u4(t-1)'          }
    {'u4(t-2)'          }
    {'u5(t-1)'          }
    {'u5(t-2)'          }
    {'u6(t-1)'          }
    {'u6(t-2)'          }
    {'u1(t-1)^2'        }
```

```
    {'y2(t-1)^3'        }
    {'u5(t-1).*u6(t-1)'}
```

```
    sysMIMO = nlarx(z, sysMIMO)
```

```
sysMIMO =
Nonlinear ARX model with 2 outputs and 6 inputs
  Inputs: u1, u2, u3, u4, u5, u6
  Outputs: y1, y2

Regressors:
  1. Linear regressors in variables y1, y2, u1, u2, u3, u4, u5, u6
  2. Order 2 regressors in variables u1
  3. Order 3 regressors in variables y2
  4. Custom regressor: u5(t-1).*u6(t-1)
  List of all regressors

Output functions:
  Output 1:  None
  Output 2:  None

Sample time: 0.02 seconds

Status:
Estimated using NLARX on time domain data "Motorized Camera".
Fit to estimation data: [99.05;98.85]% (prediction focus)
FPE: 0.2067, MSE: 0.7496
```

Compare model response to estimation data.

```
    compare(z,sysMIMO)
```
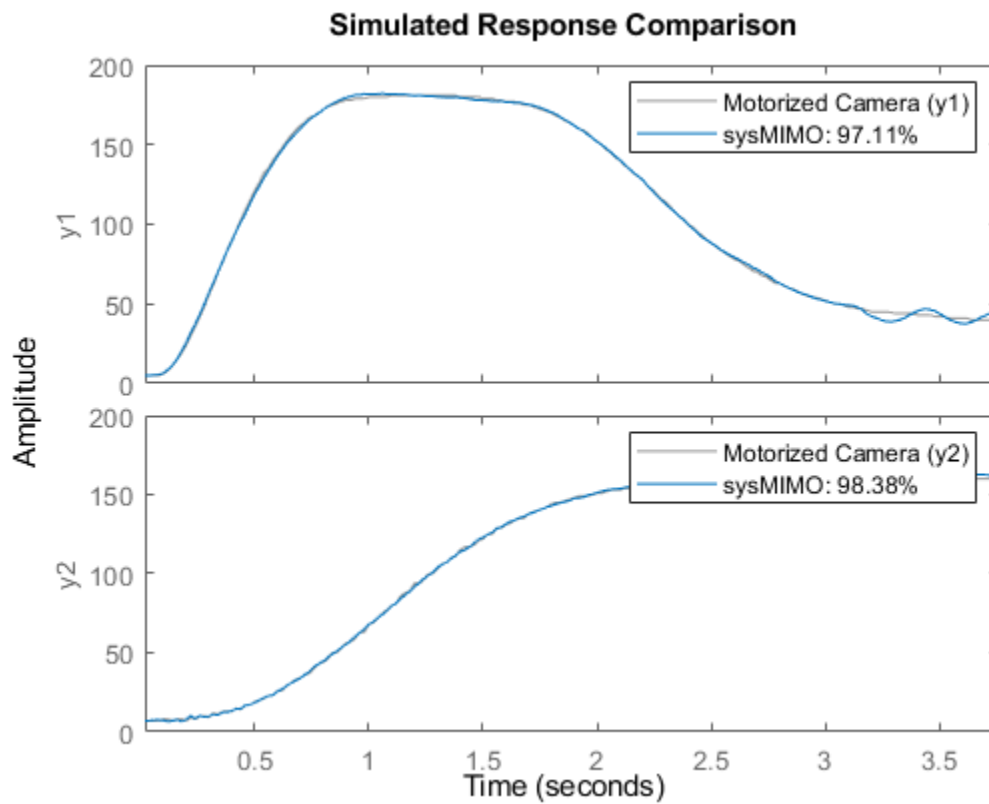
Analyze model residuals for their auto-correlation (whiteness test) and cross-correlation to the input signal (correlation test)

```
fig = figure;
fig.Position(3) = 2*fig.Position(3);
resid(z,sysMIMO)
```

Residue Correlation
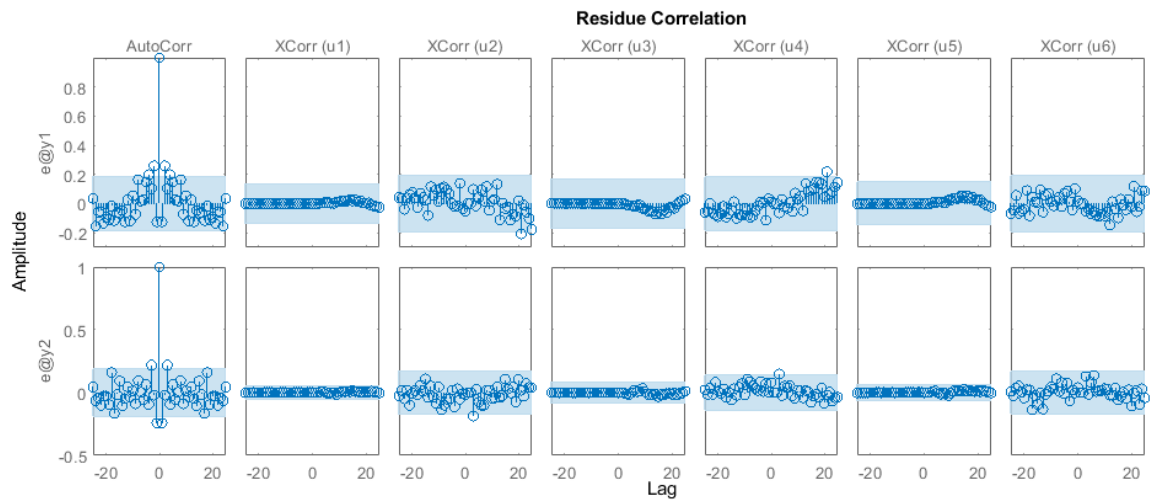
Machine-Learning-Based Identification of Two-Tank System

This example shows how to identify a black-box dynamic model of a nonlinear two-tank system.
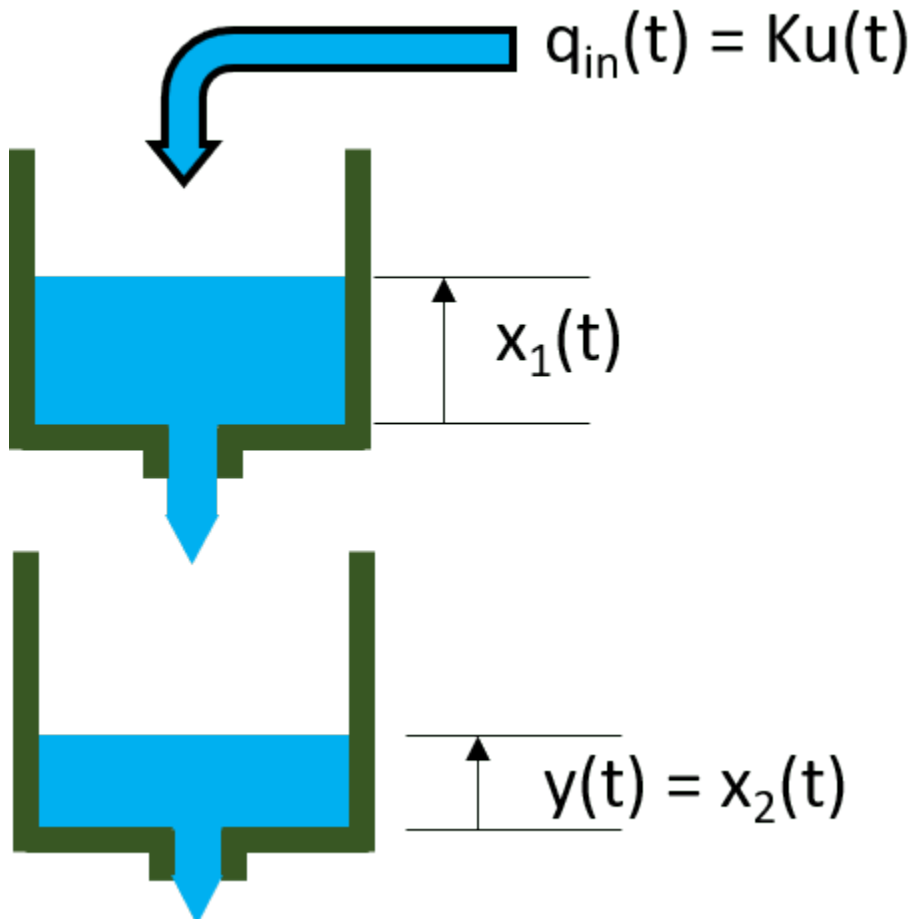
For this system, the input $u(t)$ is the voltage [V] applied to a pump, which generates an inflow to the upper tank. Water flows through a hole in the bottom of the upper tank and fills the lower tank. The output $y(t)$ of the two-tank system is the liquid level [m] of the lower tank.

The example uses measured data that contains 3000 input-output samples of a two-tank system generated using a sample time of 0.2 seconds. Load the data set and create an `iddata` object.

```
load twotankdata
z1f = iddata(y,u,0.2,'Name','Two-tank system');
```

This data set is also used in the following examples.

- "Two Tank System: C MEX-File Modeling of Time-Continuous SISO System" on page 13-135 — Nonlinear grey-box model estimation where physical equations describing the system are assumed
- "A Tutorial on Identification of Nonlinear ARX and Hammerstein-Wiener Models" on page 11-79 — Introductory example on black-box modeling using nonlinear ARX and Hammerstein-Wiener models.

This example is about black-box modeling using nonlinear ARX models with a regression tree ensemble model. Therefore, no knowledge of the actual physical laws is assumed.

Regression tree ensemble models are nonparametric models that are structured as a weighted combination of multiple regression trees. Each regression tree consists of decision nodes and leaf nodes. A decision node represents a value for an attribute (feature) being tested. A leaf node represents a numerical value associated with the decision [1]. Regression models are estimated recursively by splitting the data set into progressively smaller components. Combining multiple regression trees improves the predictive performance of regression tree ensemble models.

Preprocess Data

Before estimating a model, preprocess the data by first splitting it into estimation and validation data. Then, filter the data to remove high-frequency noise.

Split the data set into two subsets of equal size. You estimate the model using data set z1 and validate the model using data set z2.

```
z1 = z1f(1:1500);
z2 = z1f(1501:3000);
plot(z1,z2)
legend('Estimation','Validation');
```
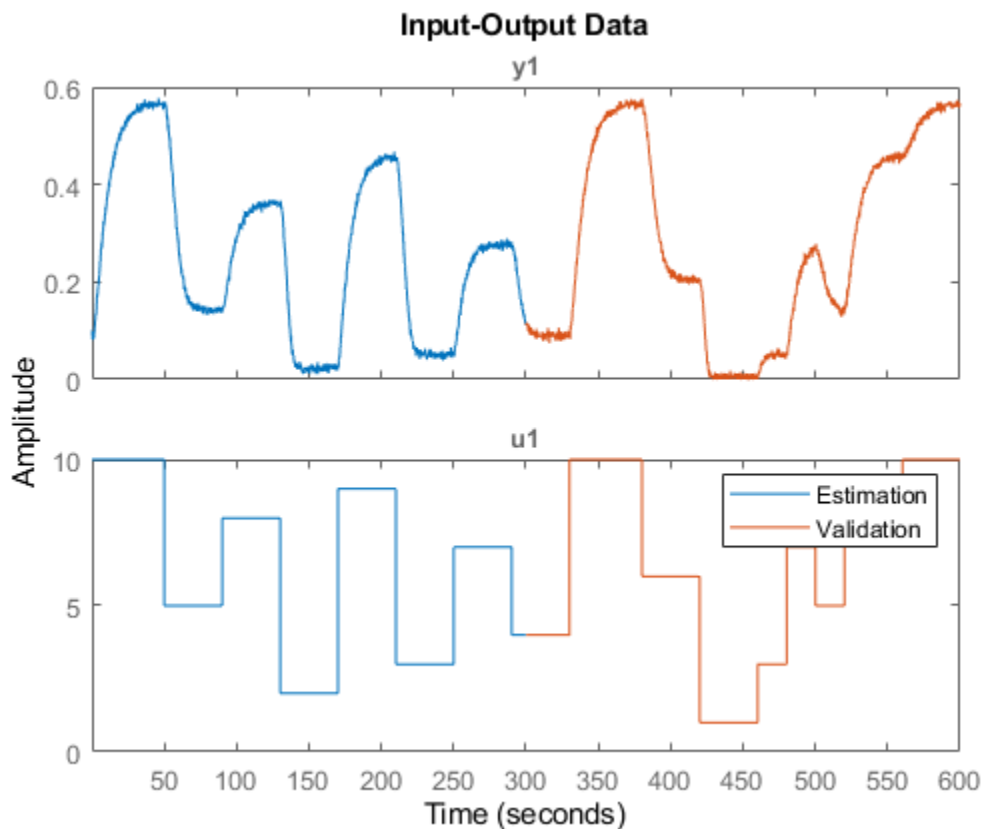

From the plot, you can see that the output data has high-frequency noise. Apply a low-pass filter to remove this noise.

```
z1f = idfilt(z1,3,0.066902);
z2f = idfilt(z2,3,0.066902);
```

Remove the effect of the filter initial conditions from the data by trimming the initial data points from each data set.

```
z1f = z1f(20:end);
z2f = z2f(20:end);
```

Plot the raw estimation data and the corresponding filtered data

```
plot(z1(20:end),z1f)
legend('Estimation (unfiltered)','Estimation (filtered)',...
    'Location','best')
```
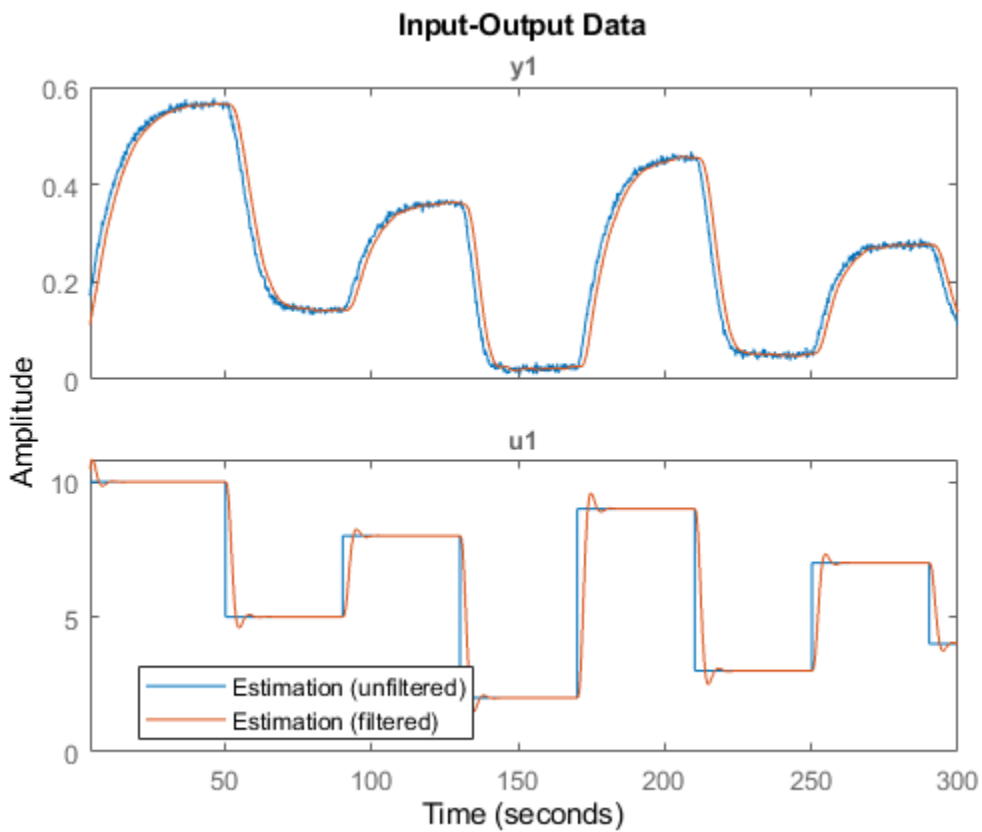

Plot the raw validation data and the corresponding filtered data

```
plot(z2(20:end),z2f)
legend('Validation (unfiltered)','Validation (filtered)',...
    'Location','best')
```
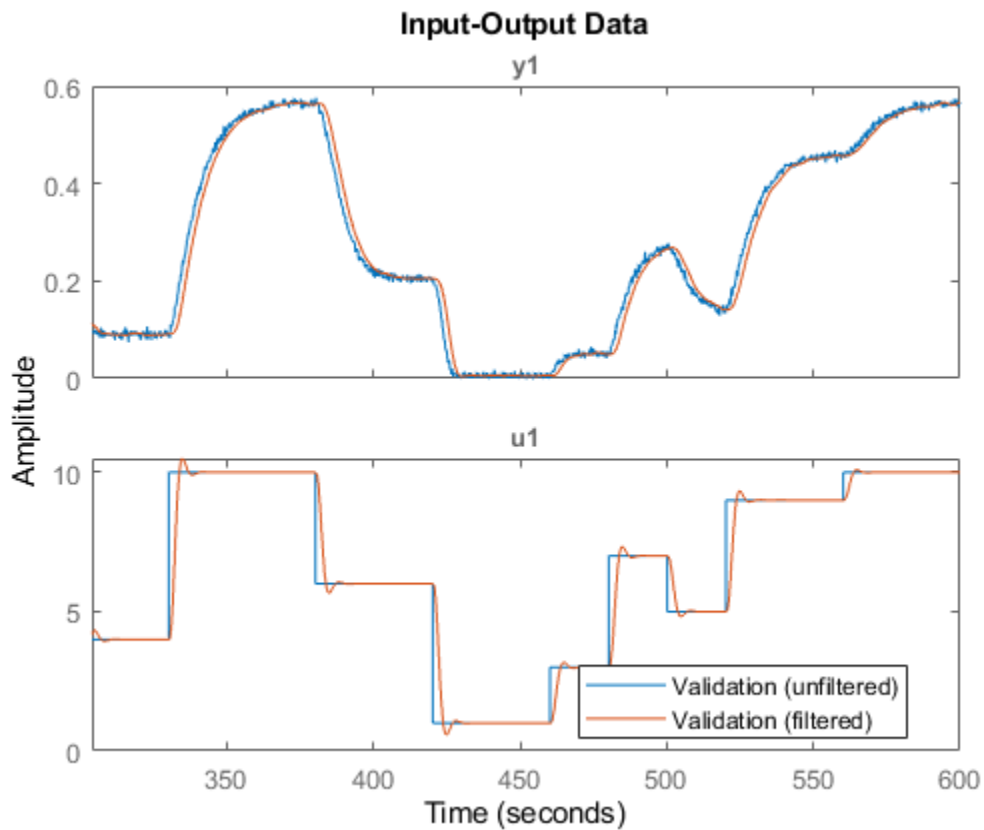
Estimate and Validate Model

Create a decision tree ensemble mapping object with default settings. You can estimate nonlinear ARX models that use this nonlinearity as an output function. Using this mapping object requires Statistics and Machine Learning Toolbox™ software.

```
ens = idTreeEnsemble;
```

Estimate a nonlinear ARX model with both input and output regressors.

```
mdl1 = nlarx(z1f,[30 30 0],ens);
```

Estimate a nonlinear ARX model with input regressors only.

```
mdl2 = nlarx(z1f,[0 150 0],ens);
```

Compare the outputs of the estimated models with the measured estimation data.

```
compare(z1f,mdl1,mdl2)
```

11-137
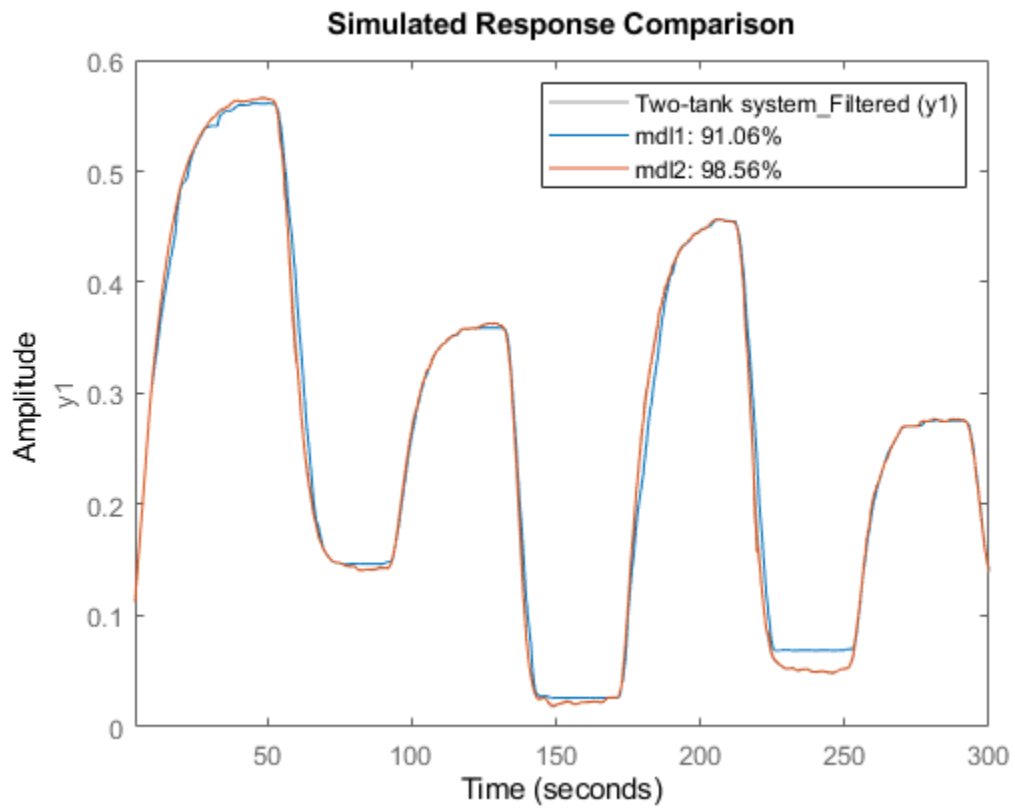
Compare the outputs of the estimated models with the measured validation data.

```
compare(z2f,mdl1,mdl2)
```
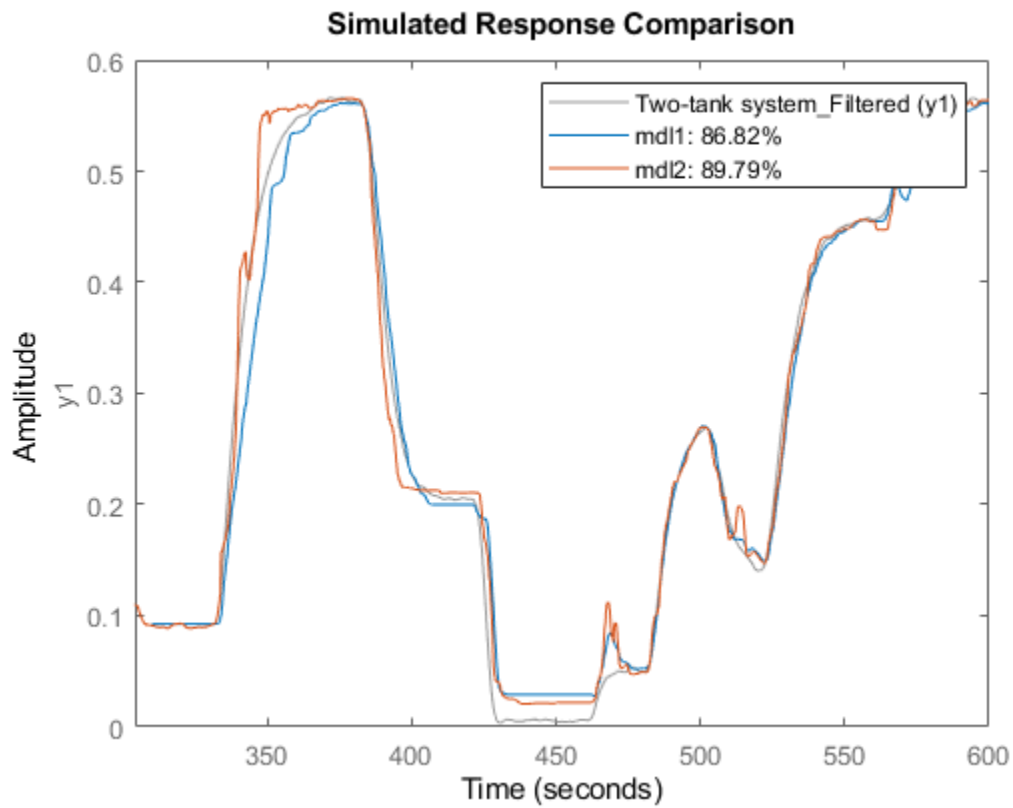
See Also

`nlarx` | `idTreeEnsemble`

Related Examples

- "Available Mapping Functions for Nonlinear ARX Models" on page 11-22
- "Identifying Nonlinear ARX Models" on page 11-17
- "A Tutorial on Identification of Nonlinear ARX and Hammerstein-Wiener Models" on page 11-79

Machine-Learning-Based Identification of Nonlinear Magneto-Rheological Fluid Damper

This example shows how to estimate a black-box dynamic model of a nonlinear magneto-rheological (MR) fluid damper. Magneto-rheological dampers are used to reduce structural vibrations by applying a controlled force, which depends on the input voltage and current of the device. A nonlinear ARX model with a regression tree ensemble mapping function is used to model the MR damper.

Regression tree ensemble models are nonparametric models that are structured as a weighted combination of multiple regression trees. Each regression tree consists of decision nodes and leaf nodes. A decision node represents a value for an attribute (feature) being tested. A leaf node represents a numerical value associated with the decision [1]. Regression models are estimated recursively by splitting the data set into progressively smaller components. Combining multiple regression trees improves the predictive performance of regression tree ensemble models.

The data used in this example was provided by Dr. Akira Sano (Keio University, Japan) and Dr. Jiandong Wang (Peking University, China) who performed the experiments in a laboratory of Keio University. For a more detailed description of the experimental system and some related studies see [2].

In the experiment, one end of an MR damper fixed to the ground. The other end is connected to a shaker table that generates vibrations. The data set contains 3,499 input-output samples. The input is the velocity V of the damper measured in cm/sec and the output is the damping force F measured in Newtons. Load the experimental data.

```
load('mrdamper.mat')
```

Construct an `iddata` object from F and V.

```
data = iddata(F,V,Ts);
```

Split the data into estimation data and validation data. To estimate a nonlinear ARX model with a regression tree ensemble mapping function, use the first 3,000 samples of the input-output data. Use the last 499 samples for validation.

```
% Estimation data
ze = data(1:3000);
% Validation data
zv = data(3001:end);
```

Plot the estimation and validation data

```
plot(ze)
hold on
plot(zv)
hold off
legend('Estimation data','Validation data')
```
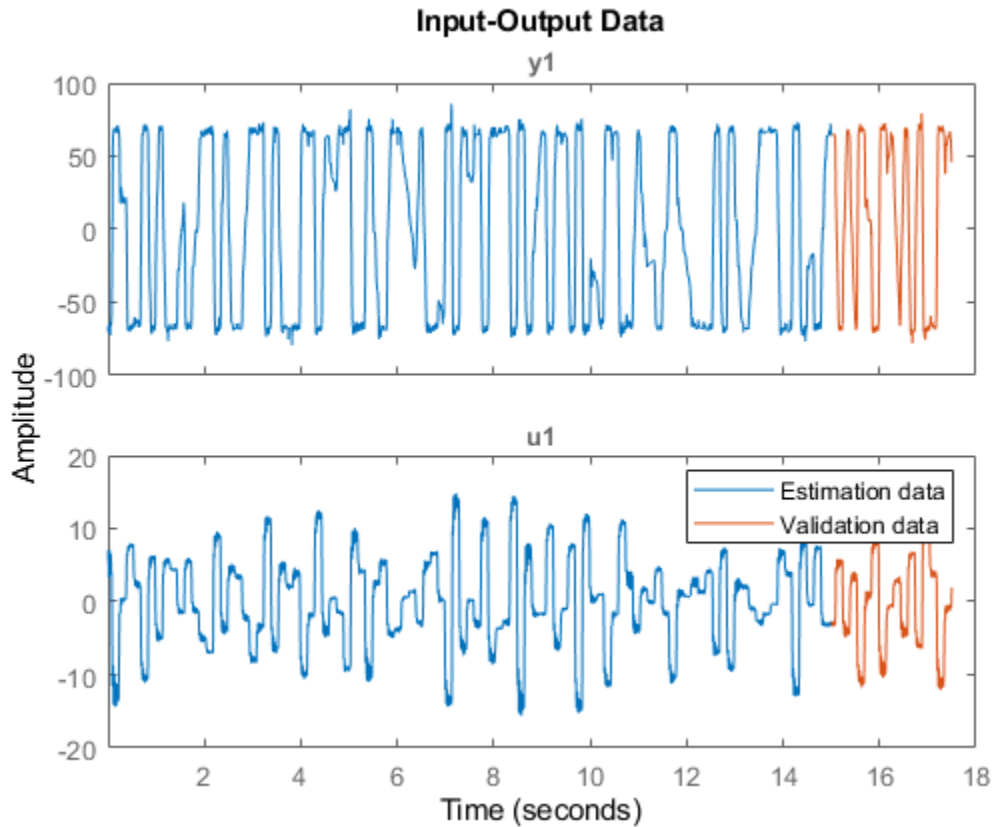
Estimate Model

Estimate a nonlinear ARX model using the tree ensemble nonlinearity. Set the nonlinear ARX model orders to [16 16 0]. Doing so configures the model to use 16 input regressors, 16 output regressors, and no input-output delays.

```
mdlTreeEnsDef = nlarx(ze,[16 16 0],idTreeEnsemble);
```

Plot the simulated response of the estimated model along with the estimation data.

```
compare(ze,mdlTreeEnsDef)
```
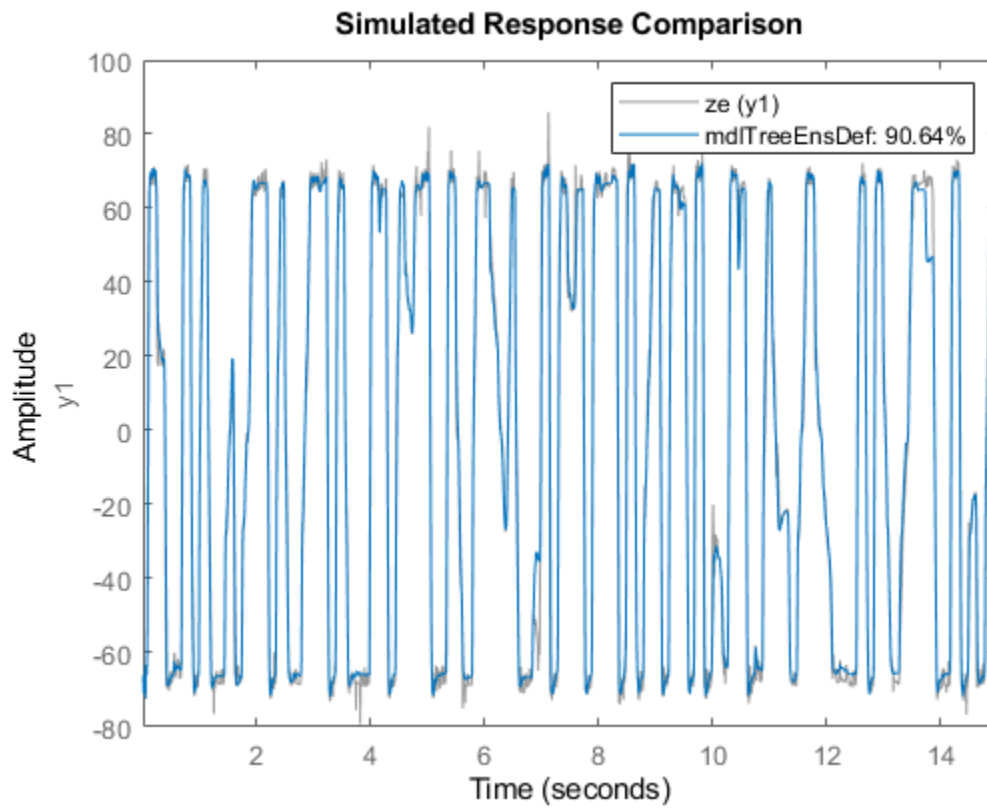
Plot the simulated response of the estimated model along with the validation data.

```
compare(zv,mdlTreeEnsDef)
```
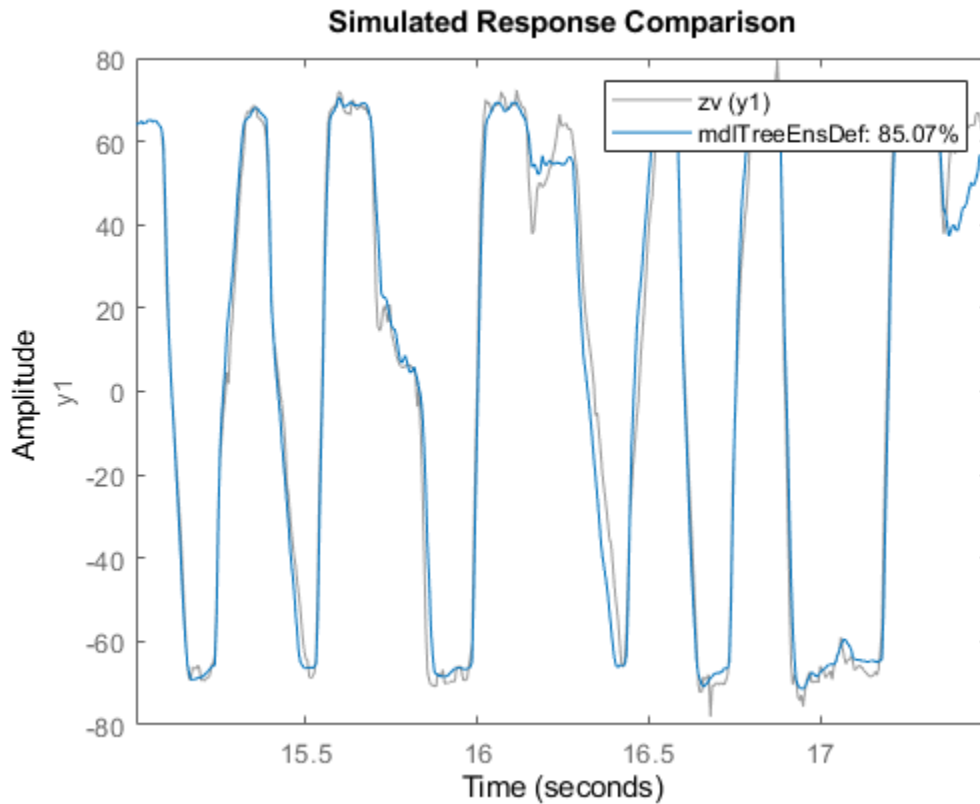
Explore Estimation Options

Use nondefault values for the estimation options. For this example, set the fit method to `'lsboost-resampled'`. Estimate a model and plot its simulated response.

```
ens = idTreeEnsemble;
ens.EstimationOptions.FitMethod = 'lsboost-reweighted';
ens.EstimationOptions.NumLearningCycles = 150;
mdlTreeEnsBoostReweight = nlarx(ze,[16 16 0],ens);
compare(zv,mdlTreeEnsBoostReweight)
```
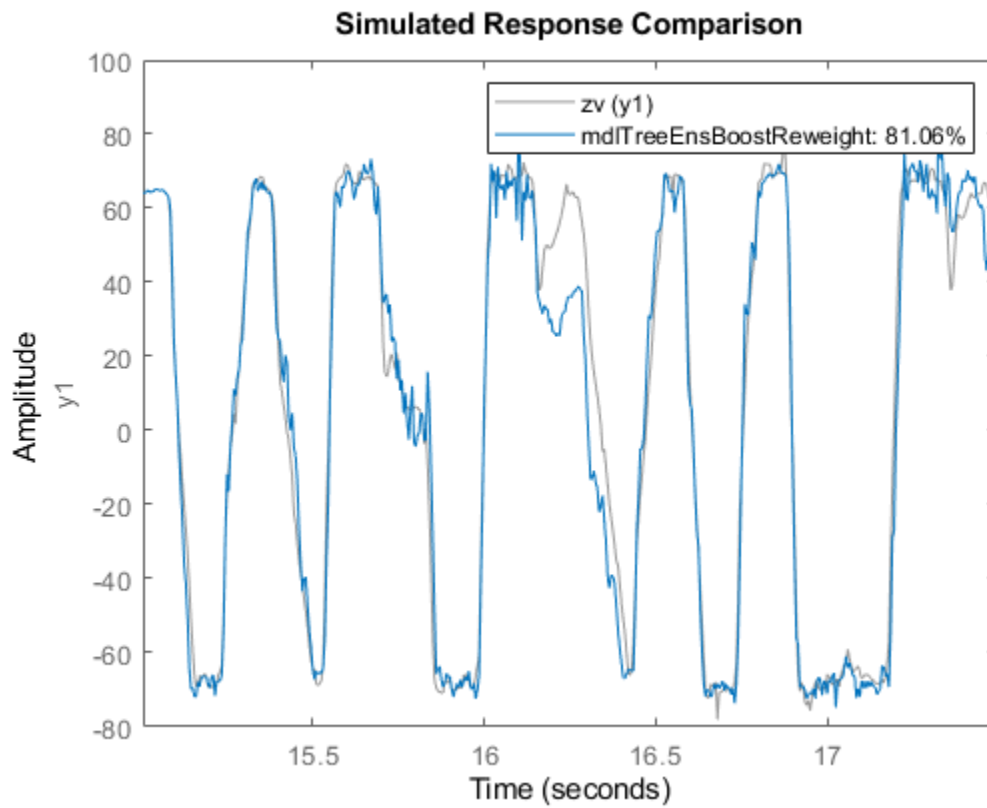
Reduce the estimation time and obtain and compact regression ensemble model by using the `Shrink` estimation option.

```
ens = idTreeEnsemble;
ens.EstimationOptions.Shrink = true;
mdlTreeEnsDefShrink = nlarx(ze,[16 16 0],ens);
compare(ze,mdlTreeEnsDefShrink)
```
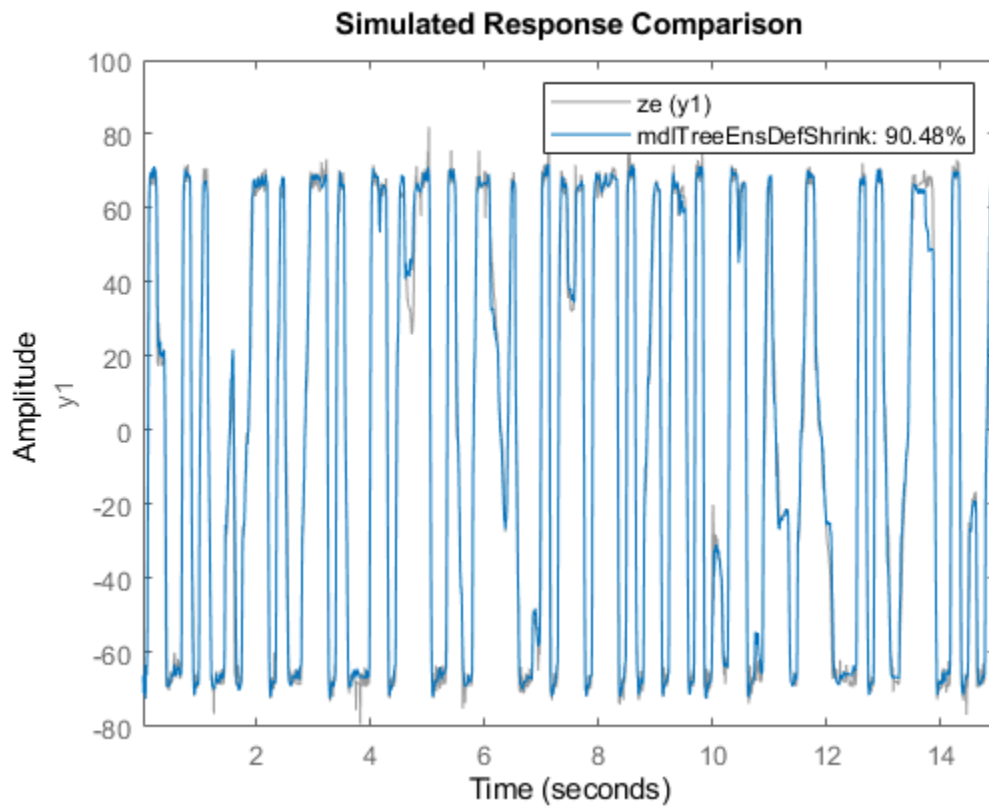
```
compare(zv,mdlTreeEnsDefShrink)
```
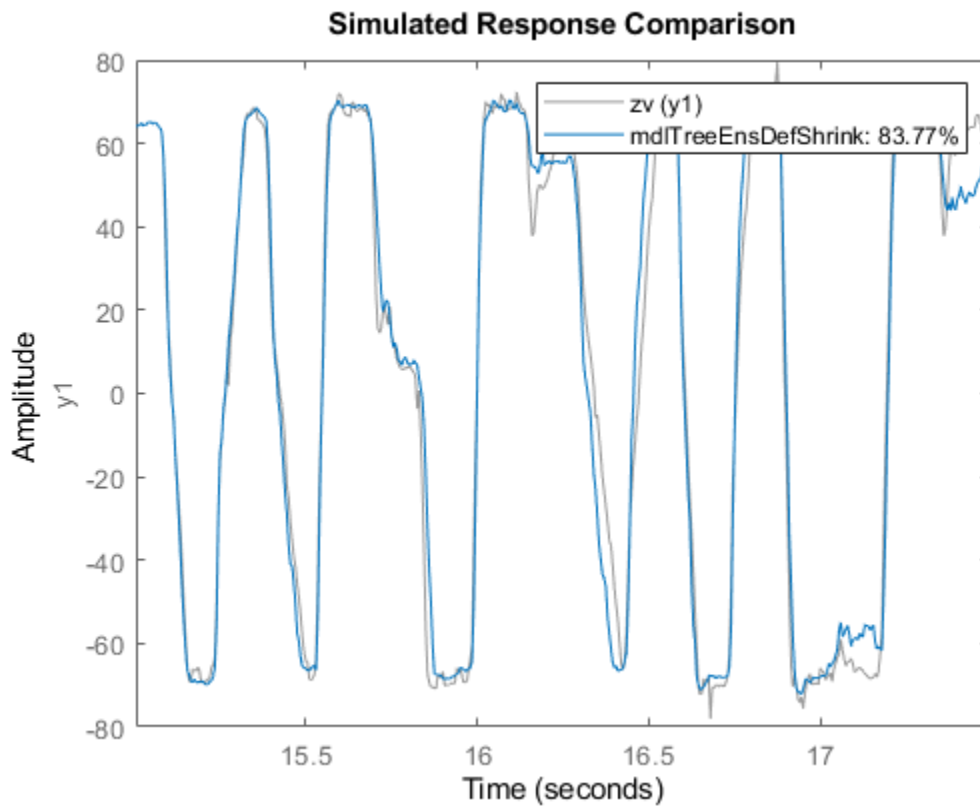
Compare Estimated Models with Other Nonlinearities

Use the training data to estimate nonlinear ARX models using wavelet network and sigmoid network nonlinearities.

```
mdlWaveletNet = nlarx(ze,[16 16 0],idWaveletNetwork);
mdlSigmoidNet = nlarx(ze,[16 16 0],idSigmoidNetwork);
```

Compare the identified nonlinear ARX models obtained in this example using the validation data.

```
compare(zv,mdlTreeEnsDef,mdlTreeEnsBoostReweight,...
    mdlTreeEnsDefShrink,mdlWaveletNet,mdlSigmoidNet)
```
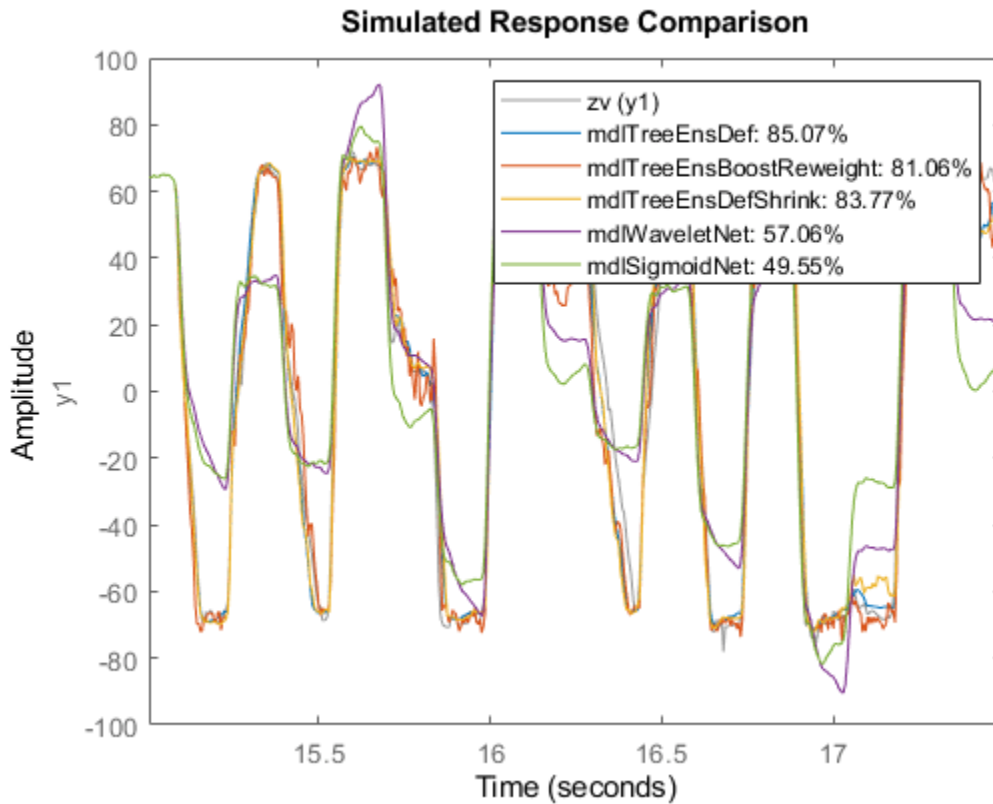
References

[1] Breiman, Leo, ed. *Classification and Regression Trees*. CRC Press repr. Boca Raton, Fla.: Chapman & Hall/CRC, 1984.

[2] Wang, Jiandong, Akira Sano, Tongwen Chen, and Biao Huang. 'Identification of Hammerstein Systems without Explicit Parameterisation of Non-Linearity'. *International Journal of Control* 82, no. 5 (May 2009): 937–52. https://doi.org/10.1080/00207170802382376.

See Also

nlarx | idTreeEnsemble

Related Examples

- "Available Mapping Functions for Nonlinear ARX Models" on page 11-22
- "Identifying Nonlinear ARX Models" on page 11-17
- "A Tutorial on Identification of Nonlinear ARX and Hammerstein-Wiener Models" on page 11-79

Surrogate Modeling Using Gaussian Process-Based NLARX Model

In this example, you replace a hydraulic cavitation cycle model in Simulink® with a surrogate nonlinear ARX (NLARX) model to facilitate faster simulation.

In the hydraulic cavitation cycle for this example, the fluid in a custom double-acting cylinder can cavitate and recover. During the repeating cycle of oscillating pressure, the absolute pressure does not drop below absolute zero as the bulk modulus of the homogeneous liquid-gas mixture decreases accordingly at low pressures. The input to this system is a velocity source, and the output observed is the absolute pressure in the cylinder chambers.

Often, such physical models are computationally expensive to simulate. You can reduce simulation time by replacing the model with a simpler surrogate model. In this example, you create a surrogate model for this physical system an estimated NLARX model with a Gaussian process nonlinear output function. Using this approach, you can guide the dynamics of the system by choosing how regressors (delayed forms of input-output data) interact with the nonlinear output function.

Obtain Input-Output Data

To estimate the model, you must first obtain input-output data by simulating the original model.

Open the model and set the amplitude of the velocity input signal.

```
modelName = 'HydraulicCavitationCycleFullModel';
velocityAmplitude = 0.05; %#ok<*NASGU>
open_system(modelName)
```
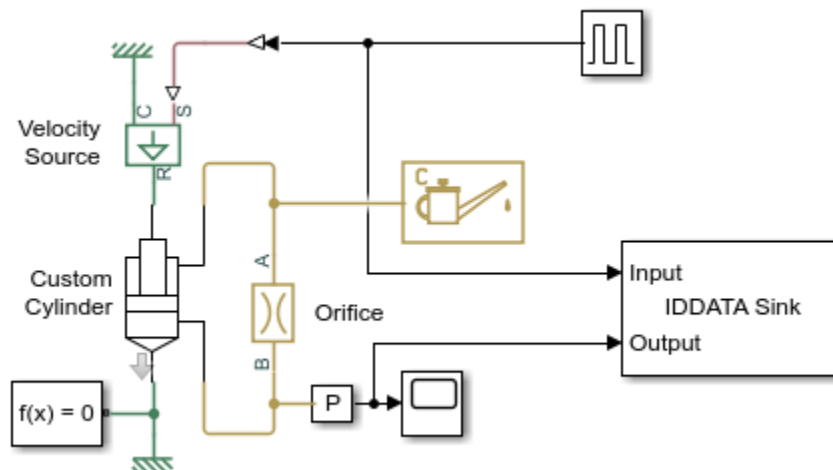

Model estimation requires an input signal that is rich enough to capture the dynamics of this nonlinear system. Therefore, the model applies a pulse train velocity signal to the custom cylinder.

This model includes an Iddata Sink block, which saves the simulated input-output data to the MATLAB® workspace as an `iddata` object.

Simulate the model.

```
sim(modelName);
```

Plot the data from the saved `iddata` object.

```
plot(cavitationCycleIOData)
```
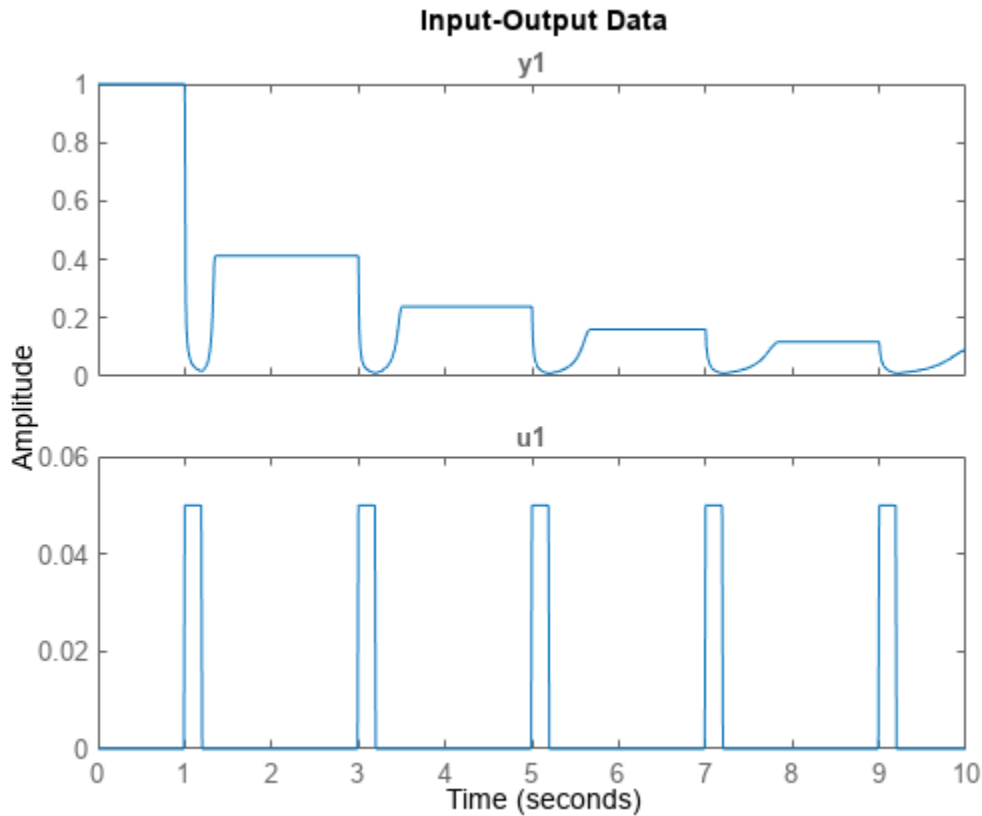

Here, y1 is the output signal, that is the absolute pressure in the cylinder chambers. `u1` is the input velocity applied to the custom cylinder.

This system is sensitive to changes in the input source. You must take this sensitivity into account when you train, validate, and use the NLARX model.

Estimate Model

Next, you estimate an NLARX model using this input-output data. The goal is to reasonably replicate the dynamics of the cavitation cycle.

An important aspect of NLARX estimation is the choice of nonlinear output function. For this example, you use a Gaussian process output function. The versatility in kernel function provided by this output function allows for variability in model structure.

Create a Gaussian process mapping object with default settings.

```
gpObject = idGaussianProcess;
```

Create a regressor set for the NLARX model.

```
vars = {'y1','u1'};
lags = {1:30,1:60};
regressors = linearRegressor(vars,lags);
```

Estimate a NLARX model using the Gaussian process output function and the defined linear regressors.

```
trainedModel = nlarx(cavitationCycleIOData,regressors,gpObject)

trainedModel =

Nonlinear ARX model with 1 output and 1 input
  Inputs: u1
  Outputs: y1

Regressors:
  Linear regressors in variables y1, u1

Output function: Gaussian process function using a SquaredExponential kernel
Sample time: 0.01 seconds

Status:
Estimated using NLARX on time domain data "cavitationCycleIOData".
Fit to estimation data: 99.96% (prediction focus)
FPE: 1.586e-08, MSE: 1.223e-08
More information in model's "Report" property.
```

Validate Model

To check the performance of the trained model, first obtain validation data from the original Simulink model. For this example, increase the velocity amplitude by 4% and simulate the model.

```
velocityAmplitude = 1.04*0.05;
sim(modelName);
```

Compare the output of the trained model to the simulation output using the validation data.

```
compare(cavitationCycleIOData,trainedModel)
```
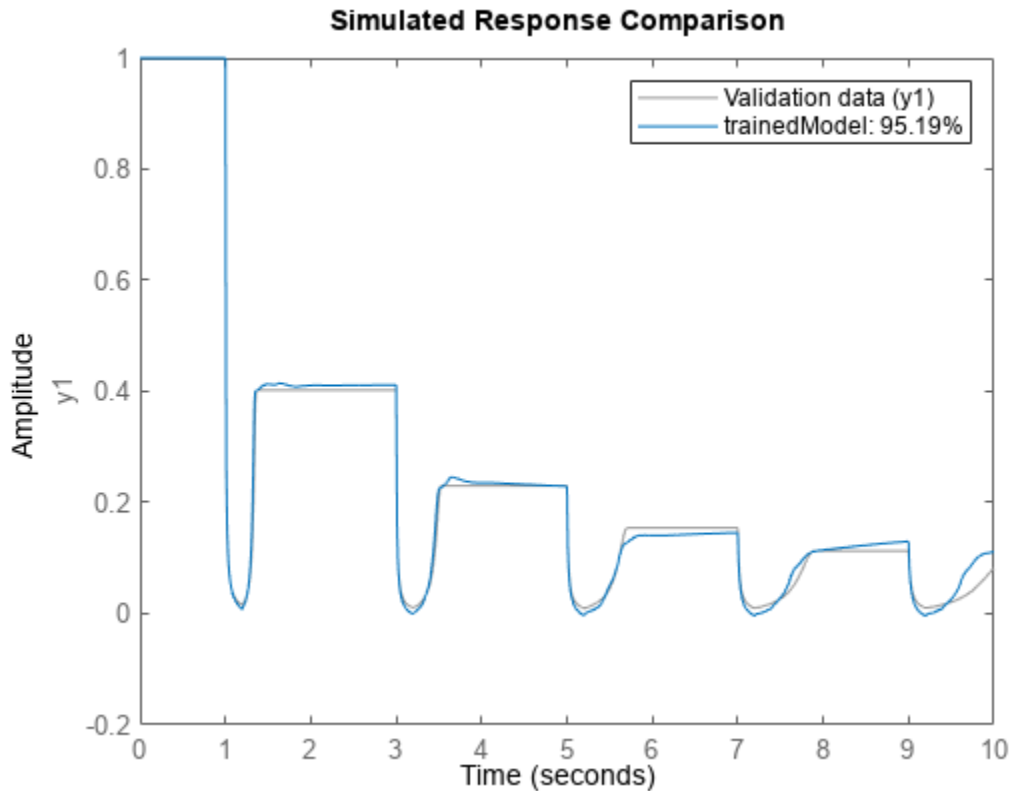
The model fit for the validation data is over 95%, which indicates that the trained model reasonably represents the dynamics of the system.

Test Model Sensitivity

The performance of the trained model is sensitive to the amplitude of the velocity input. To demonstrate this sensitivity, simulate the trained system with a larger velocity amplitude.

```
velocityAmplitude = 1.1*0.05;
sim(modelName);
```

Compare the performance of the trained model to the simulation using this testing data.

```
compare(cavitationCycleIOData,trainedModel)
```

Simulated Response Comparison
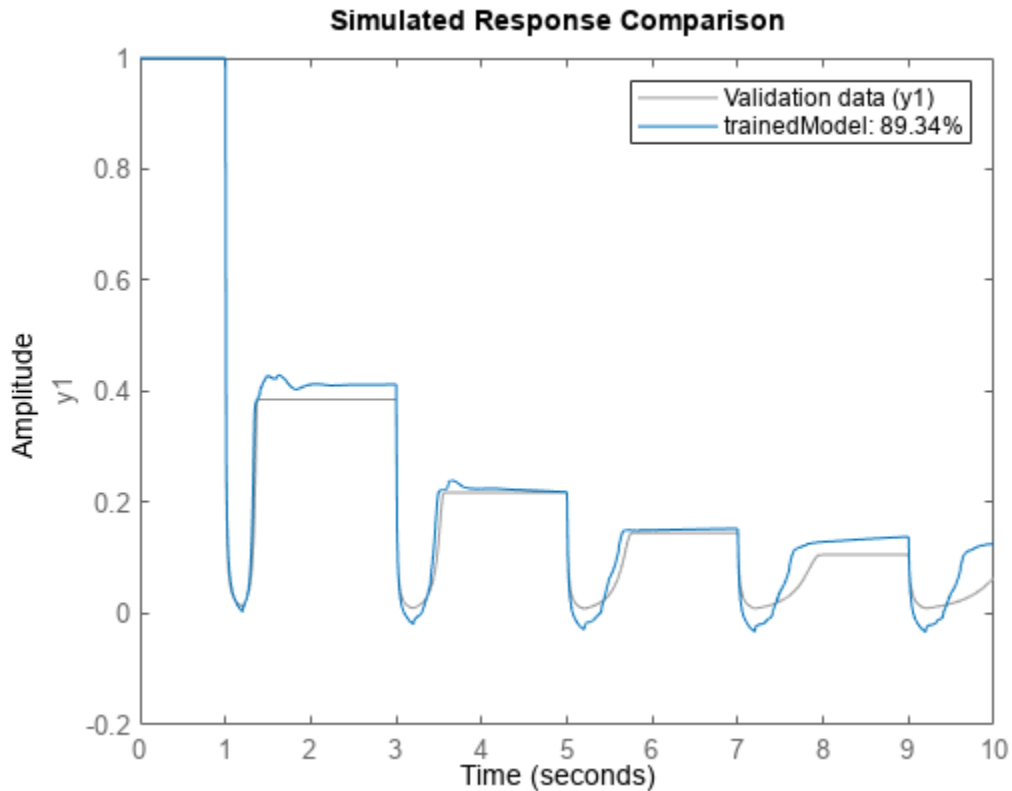
For this data, the fit for the trained model is below 90% and the output pressure differs significantly, particularly for lower pressure values.

Given the sensitivity of the trained model, you can consider training another surrogate model for larger amplitude operating conditions while retaining the existing regressor configuration with sufficiently good fit. Doing so demonstrates the flexibility of this modeling approach.

Substitute Surrogate Model

To substitute the surrogate system into the Simulink model, use a Nonlinear ARX Model block. As an example, open the `GPsurrogateModel` Simulink model.

```
surrogateModelName = 'GPSurrogateModel';
open_system(surrogateModelName)
```
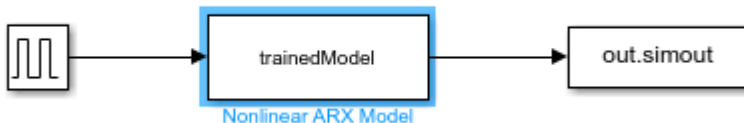

Simulate the surrogate model using the same velocity amplitude as the original estimation data.

```
velocityAmplitude = 0.05;
gpSurrogateModelData = sim(surrogateModelName);
```

Simulate the original model using the same velocity amplitude.

```
sim(modelName);
```

Plot the simulation output and compare it with the estimation data output.

```
plot(gpSurrogateModelData.tout,gpSurrogateModelData.simout, ...
    cavitationCycleIOData.SamplingInstants,cavitationCycleIOData.OutputData);
title('Surrogate Model Performance - Estimation Data')
xlabel('Time (s)')
ylabel('Absolute Pressure (atm)')
legend('Surrogate model','Original model')
```
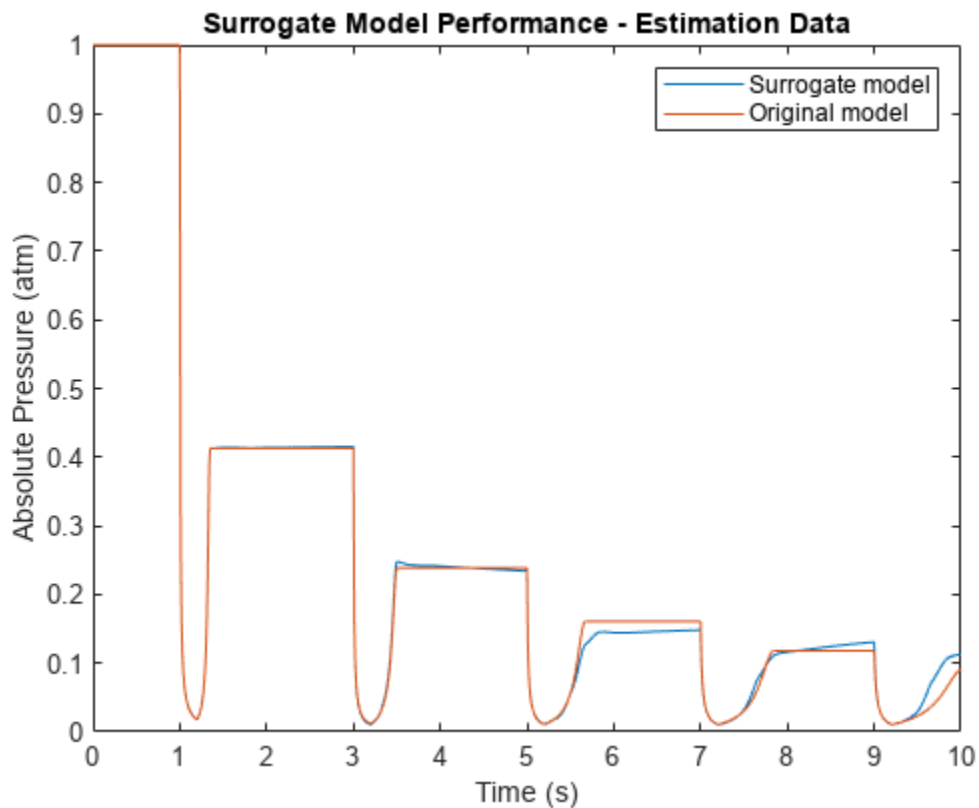

See Also

Functions
nlarx | idGaussianProcess

Blocks
Nonlinear ARX Model | Iddata Sink

Related Examples

- "Available Mapping Functions for Nonlinear ARX Models" on page 11-22
- "Identifying Nonlinear ARX Models" on page 11-17

Piezoelectric Actuator Model Identification Using Machine Learning

This example shows identification of a piezoelectric actuator using machine-learning-based techniques that include support vector machines and Gaussian process functions.

Experimental Setup

The data used in this example was provided by Prof. Micky Rakotondrabe (University of Toulouse, France) who performed the experiment in a laboratory in University of Franche-Comté at Besançon, France.

The experimental setup consists of a cantilevered piezoelectric actuator that is used in precise positioning applications. The actuator has two layers, namely, a piezoelectric layer that is made of lead zirconate titanate and a passive layer that is made of nickel. Piezoelectric actuators are used in several applications including high precision positioning and inkjet printing.

Identification of Piezoelectric Actuators

Piezoelectric actuators suffer from several problems, including hysteresis, creep, and vibration. Models based on physical principles must take these characteristics into account. The identification process for piezoelectric actuator models usually involves estimating a mathematical model that combines a nonlinear component that characterizes the behavior of the hysteresis with a linear model that characerizes the linear dynamics of the actuator. This example uses two approaches that are based on machine learning to identify a black box model of a piezoelectric actuator. The example assumes that there is no prior knowledge about the dynamics of the actuator or the hysteresis nonlinearity.

Input-Output Data

The input v is the voltage [V] applied to the actuator and the output d is the displacement of the actuator [μ m]. The data set contains 10000 input-output samples corresponding to a sampling rate of 0.0002 s. This data will be used for all estimation and validation tasks carried out in this example.

```
% load data
load idPiezoElectricData.mat
```

Estimation and Validation Data

The input voltage applied to the piezoelectric actuator in this data set is a summation of sinusoidal signals with different frequencies. The data set is split into two parts, `ze` and `zv`, where the first 5000 samples are used for estimation (`ze`), and the remaining 5000 samples are used for validation (`zv`)

```
ze = idPiezoElectricData(1:5000);   % estimation data
zv = idPiezoElectricData(5001:end); % validation data
```

Plot the two data sets together to visually verify their time ranges

```
figure; plot(ze, zv);
legend('ze','zv');
```

Input-Output Data
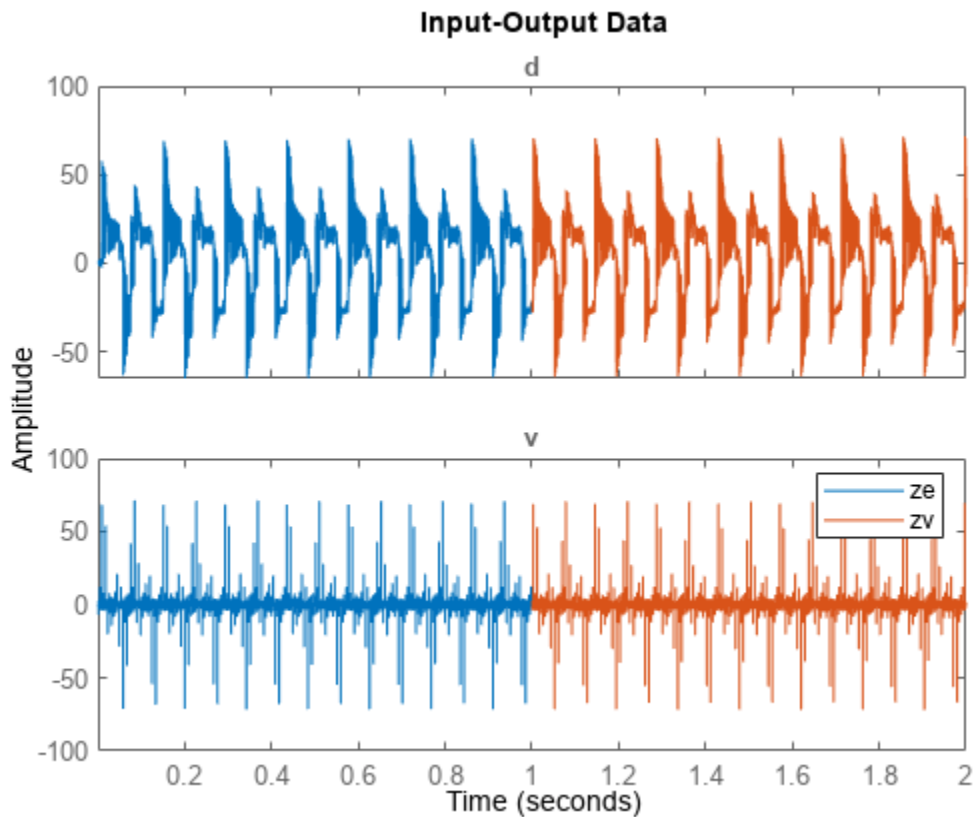
Identification Using Nonlinear ARX Model with Support Vector Machine Output Function

Configure a support vector machine (SVM) function to represent the nonlinearity in the system model.

Create the regressor set `Regressors` to use in the template `idnlarx` model.

```
Vars = {'d', 'v'};
Lags = {1:20, 0:20};

Regressors = linearRegressor(Vars, Lags);
```

Configure the `idSupportVectorMachine` nonlinearity object `NL`.

```
NL = idSupportVectorMachine;
NL.EpsilonMargin = 1e-3;
NL.KernelScale = 30;
```

Create a template `idnlarx` model `initialModel` that incorporates `Regressors` and `NL`. Then use `initialModel` to estimate an NLARX model `mdlSVM` from the data `ze`.

```
initialModel = idnlarx(ze.OutputName, ze.InputName, Regressors, NL);
mdlSVM = nlarx(ze, initialModel);
```

Visualize how the estimated model compares with the validation data.

```
compare(zv,mdlSVM);
```
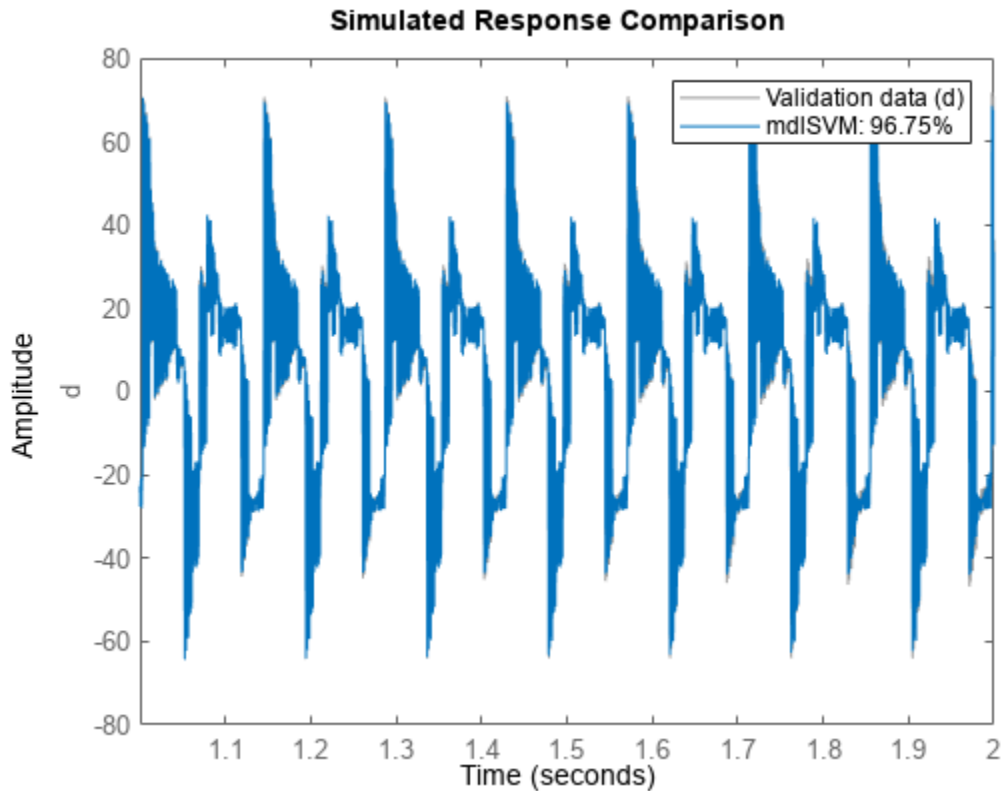
The legend in the figure shows a high fit accuracy for this model.

Identification using Nonlinear ARX Model with Gaussian Process Output Function

Configure a Gaussian process (GP) function to represent the nonlinearity in the system model.

Create the regressor set `Regressors` to use in the template `idnlarx` model.

```
% Create Regressors
Vars = {'d','v'};
Lags = {1:5,0:5};
Regressors = linearRegressor(Vars, Lags);
```

Configure the `idGaussianProcess` nonlinearity object `NL`.

```
NL = idGaussianProcess;
```

Create a template `idnlarx` model `initialModel` that incorporates `Regressors` and `NL`. Then use `initialModel` to estimate an `NLARX` model `mdlGP` from the data `ze`.

```
initialModel = idnlarx(ze.OutputName, ze.InputName, Regressors, NL);
mdlGP = nlarx(ze, initialModel);
```

Visualize how the estimated model compares with the validation data.

```
compare(zv,mdlGP);
```
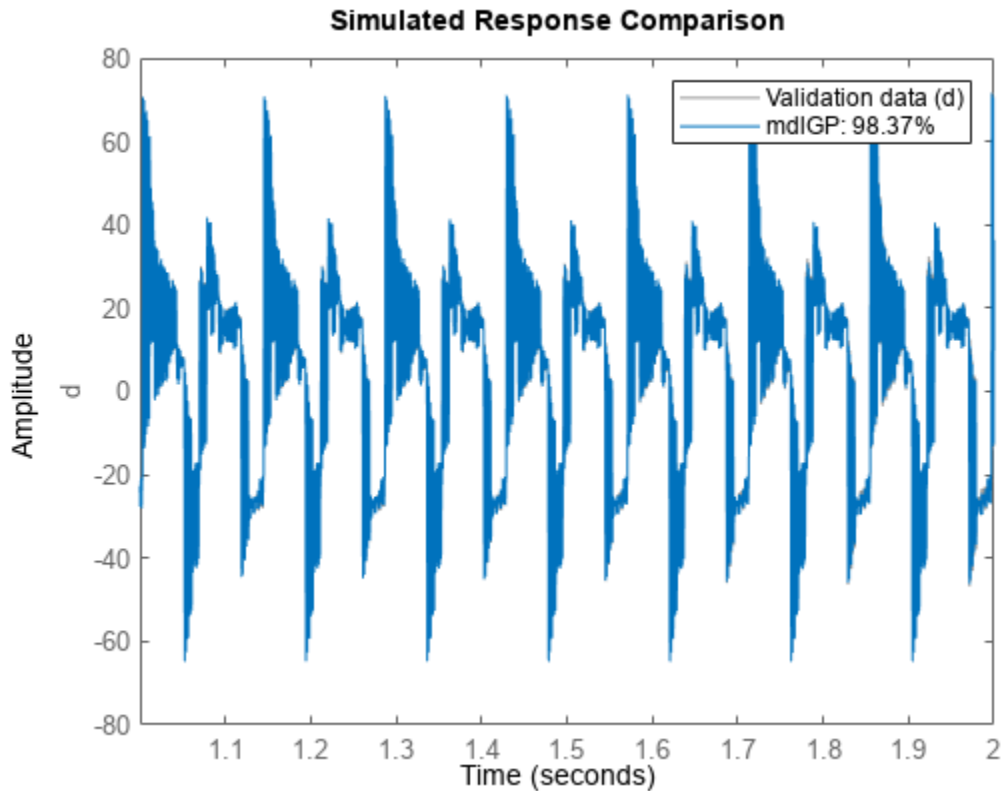
The legend in the figure shows a high fit accuracy for this model.

Compare Results for Nonlinear Models and Linear Model

Compare the identified models obtained using nonlinear ARX with support vector machines and Gaussian process output functions to an identified linear model obtained using the `ssest` command.

First set options for `ssest` using the option set `ssestOptions`. Then estimate a linear model `mdlLinear` using `ssest`.

```
opt = ssestOptions;
opt.EnforceStability = true;
opt.Focus = 'simulation';
mdlLinear = ssest(ze,6,opt);
```

Visualize how the three estimated models compare with the validation data.

```
figure
compare(zv,mdlSVM,mdlGP,mdlLinear);
```
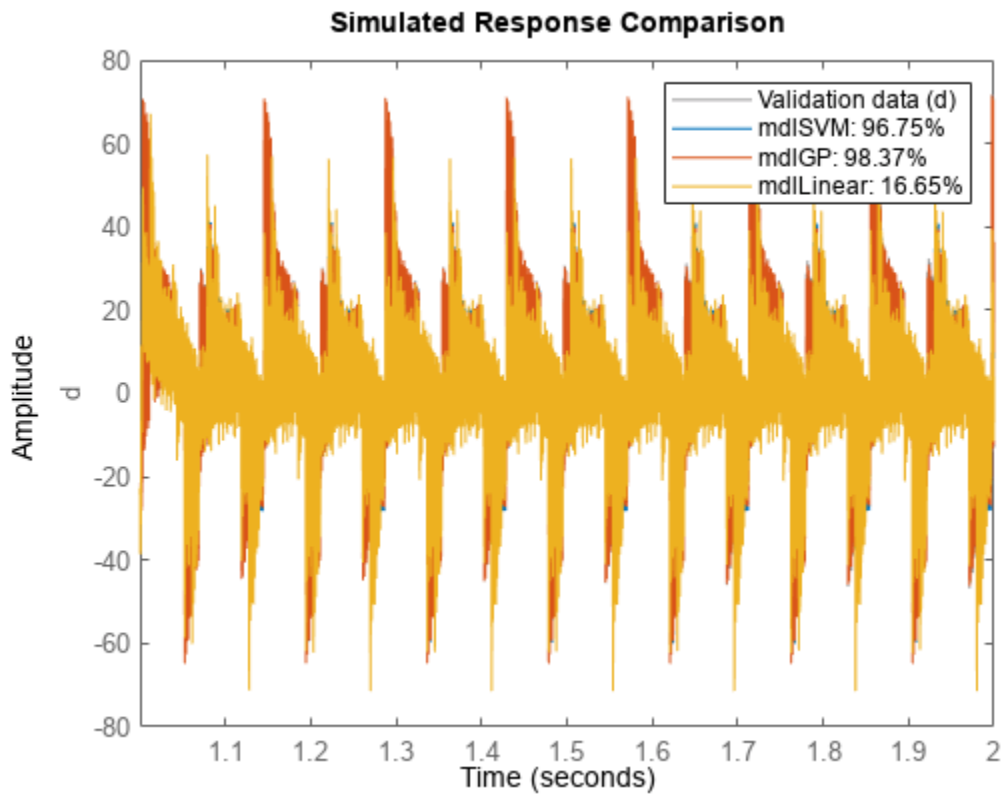
The legend in the figure shows that the fit accuracy obtained using a linear model is significantly lower than the fit accuracies for both nonlinear ARX models with output functions based on machine learning.

See Also

idnlarx | nlarx | idSupportVectorMachine | idGaussianProcess

Identify Hammerstein-Wiener Models

What are Hammerstein-Wiener Models?

When the output of a system depends nonlinearly on its inputs, sometimes it is possible to decompose the input-output relationship into two or more interconnected elements. In this case, you can represent the dynamics by a linear transfer function and capture the nonlinearities using nonlinear functions of inputs and outputs of the linear system. The Hammerstein-Wiener model achieves this configuration as a series connection of static nonlinear blocks with a dynamic linear block. Hammerstein-Wiener model applications span several areas, such as modeling electromechanical system and radio frequency components, audio and speech processing, and predictive control of chemical processes. These models have a convenient block representation, a transparent relationship to linear systems, and are easier to implement than heavy-duty nonlinear models such as neural networks and Volterra models.

You can use a Hammerstein-Wiener model as a black-box model structure because it provides a flexible parameterization for nonlinear models. For example, you can estimate a linear model and try to improve its fidelity by adding an input or output nonlinearity to this model. You can also use a Hammerstein-Wiener model as a grey-box structure to capture physical knowledge about process characteristics. For example, the input nonlinearity can represent typical physical transformations in actuators and the output nonlinearity can describe common sensor characteristics. For more information about when to fit nonlinear models, see "About Identified Nonlinear Models" on page 11-2.

Structure of Hammerstein-Wiener Models

Hammerstein-Wiener models describe dynamic systems using one or two static nonlinear blocks in series with a linear block. The linear block is a discrete transfer function that represents the dynamic component of the model.

This block diagram represents the structure of a Hammerstein-Wiener model:

Where,

- f is a nonlinear function that transforms input data $u(t)$ as $w(t) = f(u(t))$.

  $w(t)$, an internal variable, is the output of the Input Nonlinearity block and has the same dimension as $u(t)$.
- B/F is a linear transfer function that transforms $w(t)$ as $x(t) = (B/F)w(t)$.

  $x(t)$, an internal variable, is the output of the Linear block and has the same dimension as $y(t)$.

  B and F are similar to polynomials in a linear Output-Error model. For more information about Output-Error models, see "What Are Polynomial Models?" on page 6-2.

  For ny outputs and nu inputs, the linear block is a transfer function matrix containing entries:

  $$\frac{B_{j,i}(q)}{F_{j,i}(q)}$$

  where $j = 1, 2, \ldots, ny$ and $i = 1, 2, \ldots, nu$.
- h is a nonlinear function that maps the output of the linear block $x(t)$ to the system output $y(t)$ as $y(t) = h(x(t))$.

Because *f* acts on the input port of the linear block, this function is called the *input nonlinearity*. Similarly, because *h* acts on the output port of the linear block, this function is called the *output nonlinearity*. If your system contains several inputs and outputs, you must define the functions *f* and *h* for each input and output signal. You do not have to include both the input and the output nonlinearity in the model structure. When a model contains only the input nonlinearity *f*, it is called a Hammerstein model. Similarly, when the model contains only the output nonlinearity *h*, it is called a *Wiener* model.

The software computes the Hammerstein-Wiener model output *y* in three stages:

1   Compute $w(t) = f(u(t))$ from the input data.

     $w(t)$ is an input to the linear transfer function *B/F*.

     The input nonlinearity is a static (*memoryless*) function, where the value of the output a given time *t* depends only on the input value at time *t*.

     You can configure the input nonlinearity as a sigmoid network, wavelet network, saturation, dead zone, piecewise linear function, one-dimensional polynomial, or a custom network. You can also remove the input nonlinearity.

2   Compute the output of the linear block using $w(t)$ and initial conditions: $x(t) = (B/F)w(t)$.

     You can configure the linear block by specifying the orders of numerator *B* and denominator *F*.

3   Compute the model output by transforming the output of the linear block $x(t)$ using the nonlinear function *h* as $y(t) = h(x(t))$.

     Similar to the input nonlinearity, the output nonlinearity is a static function. You can configure the output nonlinearity in the same way as the input nonlinearity. You can also remove the output nonlinearity, such that $y(t) = x(t)$.

Resulting models are `idnlhw` objects that store all model data, including model parameters and nonlinearity estimators. For more information about these objects, see "Nonlinear Model Structures" on page 11-6.

You can estimate Hammerstein-Wiener models in the **System Identification** app or at the command line using the `nlhw` command. You can use uniformly sampled time-domain input-output data for estimating Hammerstein-Wiener models. Your data can have one or more input and output channels. You cannot use time series data (output only) or frequency-domain data for estimation. If you have time series data, to fit a nonlinear model, identify nonlinear ARX models or nonlinear grey-box models. For more information about these models, see "Identifying Nonlinear ARX Models" on page 11-17 and "Estimate Nonlinear Grey-Box Models" on page 13-25.

See Also
`idnlhw` | `nlhw`

More About
- "About Identified Nonlinear Models" on page 11-2
- "Identifying Hammerstein-Wiener Models" on page 12-5
- "Available Nonlinearity Estimators for Hammerstein-Wiener Models" on page 12-10
- "Using Hammerstein-Wiener Models" on page 12-9

Identifying 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. For more information about the structure of these models, see "What are Hammerstein-Wiener Models?" on page 12-2

You can estimate Hammerstein-Wiener models in the **System Identification** app or at the command line using the `nlhw` command. To estimate a Hammerstein-Wiener model, you first prepare the estimation data. You then configure the model structure and estimation algorithm, and then perform estimation. After estimation, you can validate the estimated model as described in "Validating Hammerstein-Wiener Models" on page 12-21.

Prepare Data for Identification

You can use only uniformly sampled time-domain input-output data for estimating Hammerstein-Wiener models. Your data can have one or more input and output channels. You cannot use time series data (output only) or frequency-domain data for estimation. Use nonlinear ARX on page 11-17 or nonlinear grey-box models on page 13-25 for time series data.

To prepare the data for model estimation, import your data into the MATLAB workspace, and do *one* of the following:

- **In the System Identification app** — Import data into the app, as described in "Represent Data".
- **At the command line** — Represent your data as an `iddata` object.

After importing the data, you can analyze data quality and preprocess data by interpolating missing values, filtering to emphasize a specific frequency range, or resampling using a different sample time. For more information, see "Ways to Prepare Data for System Identification" on page 2-5. For most applications, you do not need to remove offsets and linear trends from the data before nonlinear modeling. However, data detrending can be useful in some cases, such as before modeling the relationship between the change in input and output about an operating point.

After preparing your estimation data, you can configure your model structure, loss function, and estimation algorithm, and then estimate the model using the estimation data.

Configure Hammerstein-Wiener Model Structure

The Hammerstein-Wiener model structure consists of input and output nonlinear blocks in series with a linear block. The linear block is a discrete transfer function and represents the dynamic component of the model.

To configure the structure of a Hammerstein-Wiener model:

1   Configure the linear transfer function block.

Perform *one* of the following:

- Specify model order and input delay for the linear transfer function as:

- *nb* — Number of zeros plus one. *nb* is the length of the numerator (*B*) polynomial.
- *nf* — Number of poles. *nf* is the order of the transfer function denominator (*F* polynomial).
- *nk* — Delay from input to the output in terms of the number of samples.

For MIMO systems with *Ny* outputs and *Nu* inputs, *nb*, *nf*, and *nk* are *Ny*-by-*Nu* matrices.

- Initialize the linear block using a discrete-time linear model — You can initialize using linear models at the command line only. The initialization sets the transfer function of the linear block to that of the specified linear model. For more information, see "Initialize Hammerstein-Wiener Estimation Using Linear Model" on page 12-7.

2 Configure the input and output nonlinearities, *f* and *h* respectively.

The default input and output nonlinearity estimators are piecewise linear functions. See the `idPiecewiseLinear` reference page for more information. To configure the input and output nonlinearity estimators:

a Choose the type of input and output nonlinearity estimators, and configure their properties.

For a list of available nonlinearity estimators, see "Available Nonlinearity Estimators for Hammerstein-Wiener Models" on page 12-10.

b Exclude the input or output nonlinear block.

You do not have to include both the input and the output nonlinearity in the model structure. When a model contains only the input nonlinearity *f*, it is called a Hammerstein model. Similarly, when the model contains only the output nonlinearity *h*, it is called a *Wiener* model.

For information about how to configure the model structure at the command line and in the app, see "Estimate Hammerstein-Wiener Models at the Command Line" on page 12-15 and "Estimate Hammerstein-Wiener Models in the App" on page 12-12.

Specify Estimation Options for Hammerstein-Wiener Models

To configure the model estimation, specify the loss function to be minimized, and choose the estimation algorithm and other estimation options to perform the minimization.

Configure Loss Function

The loss function or cost function is a function of the error between the model output and the measured output. For more information about loss functions, see "Loss Function and Model Quality Metrics" on page 1-46.

At the command line, use the `nlhw` option set, `nlhwOptions` to configure your loss function. You can specify the following options:

- `OutputWeight` — Specify a weighting of the error in multi-output estimations.
- `Regularization` — Modify the loss function to add a penalty on the variance of the estimated parameters. For more information, see "Regularized Estimates of Model Parameters" on page 1-34.

For details about how to specify these options in the app, see "Estimate Hammerstein-Wiener Models in the App" on page 12-12.

Specify Estimation Algorithm

To estimate a Hammerstein-Wiener model, the software uses iterative search algorithms to minimize the loss function. At the command line, use `nlhwOptions` to specify the search algorithm and other estimation options. Some of the options you can specify are:

- `SearchMethod` — Search method for minimization of prediction or simulation errors, such as Gauss-Newton and Levenberg-Marquardt line search, and Trust-region reflective Newton approach.

- `SearchOptions` — Option set for the search algorithm, with fields that depend on the value of `SearchMethod`, such as:

  - `MaxIterations` — Maximum number of iterations to perform.
  - `Tolerance` — Condition for terminating iterative search when the expected improvement of the parameter values is less than a specified value.

- `InitialCondition` — By default, the software treats the initial states of the model as zero and does not estimate the states. You can choose to estimate initial states, which sometimes can improve parameter estimates.

To see a complete list of available estimation options, see `nlhwOptions`. For details about how to specify these estimation options in the app, see "Estimate Hammerstein-Wiener Models in the App" on page 12-12.

After preprocessing the estimation data and configuring the model structure, loss function, and estimation options, you can estimate the model in the **System Identification** app, or using `nlhw`. The resulting model is an `idnlhw` object that stores all model data, including model parameters and nonlinearity estimator. For more information about these model objects, see "Nonlinear Model Structures" on page 11-6. You can validate the estimated model as described in "Validating Hammerstein-Wiener Models" on page 12-21.

Initialize Hammerstein-Wiener Estimation Using Linear Model

At the command line, you can use one of the following linear models to initialize the linear block of a Hammerstein-Wiener model:

- Polynomial model of Output-Error (OE) structure (`idpoly`)
- State-space model with no disturbance component (`idss` model with K = 0)
- Transfer function (`idtf` model)

Typically, you use the `oe`, `n4sid`, or `tfest` commands to obtain the linear model. You can provide the linear model when constructing or estimating a Hammerstein-Wiener model. For example, use the following syntax to estimate a Hammerstein-Wiener model using estimation data and a linear model `LinModel`.

```
m = nlhw(data,LinModel)
```

Here `m` is an `idnlhw` object, and `data` is a time-domain `iddata` object. The software uses the linear model for initializing the Hammerstein-Wiener estimation by:

- Assigning the linear model orders as initial values of nonlinear model orders (`nb` and `nf` properties of the Hammerstein-Wiener (`idnlhw`) and delays (`nk` property).

- Setting the *B* and *F* polynomials of the linear transfer function in the Hammerstein-Wiener model structure on page 12-2.

During estimation, the estimation algorithm uses these values to adjust the nonlinear model to the data. By default, both the input and output nonlinearity estimators are piecewise linear functions (see `idPiecewiseLinear`).

You can also specify different input and output nonlinearity estimators. For example, a sigmoid network input nonlinearity estimator and a dead-zone output nonlinearity estimator.

```
m = nlhw(data,LinModel,'idSigmoidNetwork','idDeadZone')
```

For an example, see "Estimate Hammerstein-Wiener Models Initialized Using Linear OE Models" on page 12-28.

See Also

More About

- "What are Hammerstein-Wiener Models?" on page 12-2
- "Estimate Hammerstein-Wiener Models in the App" on page 12-12
- "Estimate Hammerstein-Wiener Models at the Command Line" on page 12-15
- "Validating Hammerstein-Wiener Models" on page 12-21
- "Using Hammerstein-Wiener Models" on page 12-9

Using Hammerstein-Wiener Models

After identifying a Hammerstein-Wiener model, you can use the model for the following tasks:

- **Simulation** — At the command line, use `sim` to simulate the model output. To compare models to measured output and to each other, use `compare`. Note that for Hammerstein-Wiener models, the simulated and predicted model output are equivalent because these models have a trivial noise component, that is disturbance in these models is white noise. For information about plotting simulated output in the app, see "Simulation and Prediction in the App" on page 17-15.

    You can also specify the initial conditions for simulation. The toolbox provides various options to facilitate how you specify initial states. For example, you can use `findstates` to automatically search for state values in simulation and prediction applications. You can also specify the states manually. See the `idnlhw` reference page for a definition of the Hammerstein-Wiener model states.

    To learn more about how `sim` computes the model output, see "How the Software Computes Hammerstein-Wiener Model Output" on page 12-25.

- **Linearization** — Compute linear approximation of Hammerstein-Wiener models using `linearize` or `linapp`.

    The `linearize` command provides a first-order Taylor series approximation of the system about an operating point. `linapp` computes a linear approximation of a nonlinear model for a given input data. For more information, see the "Linear Approximation of Nonlinear Black-Box Models" on page 11-50. You can compute the operating point for linearization using `findop`.

    After computing a linear approximation of a nonlinear model, you can perform linear analysis and control design on your model using Control System Toolbox commands. For more information, see "Using Identified Models for Control Design Applications" on page 19-2 and "Create and Plot Identified Models Using Control System Toolbox Software" on page 19-5.

- **Simulation and code generation using Simulink** — You can import the estimated Hammerstein-Wiener model into Simulink software using the Hammerstein-Wiener block (Hammerstein-Wiener Model) from the System Identification Toolbox block library. After you bring the `idnlhw` object from the workspace into Simulink, you can simulate the model output.

    The IDNLHW Model block supports code generation with Simulink Coder software, using both generic and embedded targets. Code generation does not work when the model contains `idCustomNetwork` as the input or output nonlinearity. For more information, see "Simulate Identified Model in Simulink" on page 20-5.

See Also

More About

- "What are Hammerstein-Wiener Models?" on page 12-2
- "Identifying Hammerstein-Wiener Models" on page 12-5
- "Linear Approximation of Nonlinear Black-Box Models" on page 11-50

Available Nonlinearity Estimators for Hammerstein-Wiener Models

A Hammerstein-Weiner Model consists of three sequential blocks—a static input nonlinearity block, a dynamic linear block, and a static output nonlinearity block. Each nonlinearity block contains one or more nonlinearity estimators, also known as mapping objects. Each nonlinear estimator object maps a scalar input signals into a scalar output signal, and typically includes a nonlinear function, a linear function, and an offset. For more information about Hammerstein-Wiener model structure, see "What are Hammerstein-Wiener Models?" on page 12-2.

The System Identification Toolbox software provides a set of nonlinearity estimators for Hammerstein-Wiener models. These nonlinearity estimators are available for both the input and output nonlinearities f and h, respectively. For more information about f and h, see "Structure of Hammerstein-Wiener Models" on page 12-2. When you estimate Hammerstein-Wiener models in the **System Identification** app or at the command line, the software creates and configures the nonlinear estimators that you specify. You can also create and configure nonlinear estimators independently at the command line and then specify these customized objects when you perform estimation.

Some nonlinear estimators represent simple nonlinear functions, such as a dead zone or a saturation limit. Others represent more complex nonlinear functions, such as a summed series of nonlinear units in a wavelet or sigmoid network. Another nonlinear estimator draws on a machine learning algorithm. One mapping object contains simply a unit gain, and serves to exclude an input or output nonlinearity from the model structure. The following table lists the available nonlinearity estimator objects. For a detailed description of an estimator object, see the corresponding reference page.

Nonlinearity	Object	Description	Comments
Piecewise linear	`idPiecewiseLinear`	A piecewise linear function parameterized by breakpoint locations.	Default
One layer sigmoid network	`idSigmoidNetwork`	Sum of dilated and translated sigmoid unit functions.	
Wavelet network	`idWaveletNetwork`	Sum of dilated and translated wavelets in a wavelet network.	
Saturation	`idSaturation`	Parameterize hard limits on the signal value as upper and lower saturation limits.	Use to model known saturation effects on signal amplitudes.
Dead zone	`idDeadZone`	Parameterize dead zones in signals as the duration of zero response.	Use to model known dead zones in signal amplitudes.
One-dimensional polynomial	`idPolynomial1D`	Single-variable polynomial of a degree that you specify.	
Unit gain	`idUnitGain`	Excludes the input or output nonlinearity from the model structure to achieve a Wiener or Hammerstein configuration, respectively. Note that 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.

Nonlinearity	Object	Description	Comments
Custom network	`idCustomNetwork`	Similar to sigmoid network but with a user-defined unit function.	User Defined. For advanced use.
Gaussian process (GP) regression model	`idGaussianProcess`	Kernel-based zero-mean Gaussian random process regression model. Use for output nonlinearities only.	Useful when measurement data is limited. Requires Statistics and Machine Learning Toolbox.

See Also

More About

- "What are Hammerstein-Wiener Models?" on page 12-2
- "Identifying Hammerstein-Wiener Models" on page 12-5

Estimate Hammerstein-Wiener Models in the App

You can estimate Hammerstein-Wiener models in the **System Identification** app after performing the following tasks:

- Import data into the System Identification app (see "Preparing Data for Nonlinear Identification" on page 11-13).
- (Optional) Choose a nonlinearity estimator in "Available Nonlinearity Estimators for Hammerstein-Wiener Models" on page 12-10.
- (Optional) Estimate or construct an OE or state-space linear model to use for initialization. See "Initialize Hammerstein-Wiener Estimation Using Linear Model" on page 12-7.

To estimate a Hammerstein-Wiener model using the imported estimation data, chosen nonlinearity estimators, and initial linear models:

1. In the **System Identification** app, select **Estimate > Nonlinear models** to open the Nonlinear Models dialog box.
2. In the **Configure** tab, select `Hammerstein-Wiener` from the **Model type** list.
3. (Optional) Edit the **Model name** by clicking the pencil icon. The name of the model should be unique to all Hammerstein-Wiener models in the System Identification app.
4. (Optional) If you want to refine a previously estimated model, click **Initialize** to select a previously estimated model from the **Initial Model** list.

   Note Refining a previously estimated model starts with the parameter values of the initial model and uses the same model structure. You can change these settings.

   The **Initial Model** list includes models that:

   - Exist in the System Identification app.
   - Have the same number of inputs and outputs as the dimensions of the estimation data (selected as **Working Data** in the System Identification app).

5. Keep the default settings in the Nonlinear Models dialog box that specify the model structure, or modify these settings:

   Note For more information about available options, click **Help** in the Nonlinear Models dialog box to open the app help.

What to Configure	Options in Nonlinear Models GUI	Comment
Input or output nonlinearity	In the **I/O Nonlinearity** tab, select the **Nonlinearity** and specify the **No. of Units**.	If you do not know which nonlinearity to try, use the (default) piecewise linear nonlinearity. When you estimate from binary input data, you cannot reliably estimate the input nonlinearity. In this case, set **Nonlinearity** for the input channel to None. For multiple-input and multiple-output systems, you can assign nonlinearities to specific input and output channels.
Model order and delay	In the **Linear Block** tab, specify **B Order**, **F Order**, and **Input Delay**. For MIMO systems, select the output channel and specify the orders and delays from each input channel.	If you do not know the input delay values, click **Infer Input Delay**. This action opens the Infer Input Delay dialog box which suggests possible delay values.
Estimation algorithm	In the **Estimate** tab, click **Estimation Options**.	You can specify to estimate initial states.

6 To obtain regularized estimates of model parameters, in the **Estimate** tab, click **Estimation Options**. Specify the regularization constants in the **Regularization_Tradeoff_Constant** and **Regularization_Weighting** fields. To learn more, see "Regularized Estimates of Model Parameters" on page 1-34.

7 Click **Estimate** to add this model to the System Identification app.

The **Estimate** tab displays the estimation progress and results.

8 Validate the model response by selecting the desired plot in the **Model Views** area of the System Identification app.

If you get a poor fit, try changing the model structure or algorithm configuration in step 5.

You can export the estimated model to the MATLAB workspace by dragging it to **To Workspace** in the System Identification app.

See Also

More About

- "What are Hammerstein-Wiener Models?" on page 12-2
- "Available Nonlinearity Estimators for Hammerstein-Wiener Models" on page 12-10

Estimate Hammerstein-Wiener Models at the Command Line

You can estimate Hammerstein-Wiener models after performing the following tasks:

- Prepare your data, as described in "Preparing Data for Nonlinear Identification" on page 11-13.
- (Optional) Choose a nonlinearity estimator in "Available Nonlinearity Estimators for Hammerstein-Wiener Models" on page 12-10.
- (Optional) Estimate or construct an input-output polynomial model of Output-Error (OE) structure (idpoly) or a state-space model with no disturbance component (idss with K=0) for initialization of Hammerstein-Wiener model. See "Initialize Hammerstein-Wiener Estimation Using Linear Model" on page 12-7.

Estimate Model Using nlhw

Use nlhw to both construct and estimate a Hammerstein-Wiener model. After each estimation, validate the model on page 12-21 by comparing it to other models and simulating or predicting the model response.

Basic Estimation

Start with the simplest estimation using m = nlhw(data,[nb nf nk]). For example:

```
load iddata3;
% nb = nf = 2 and nk = 1
m = nlhw(z3,[2 2 1])

m =

Hammerstein-Wiener model with 1 output and 1 input

Linear transfer function corresponding to the orders nb = 2, nf = 2, nk = 1

Input nonlinearity: Piecewise linear with 10 break-points
Output nonlinearity: Piecewise linear with 10 break-points
Sample time: 1 seconds

Status:
Termination condition: Maximum number of iterations reached..
Number of iterations: 20, Number of function evaluations: 185

Estimated using NLHW on time domain data "z3".
Fit to estimation data: 75.31%
FPE: 2.019, MSE: 1.472
More information in model's "Report" property.
```

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 idPiecewiseLinear reference page). m is an idnlhw object.

For MIMO systems, *nb*, *nf*, and *nk* are *ny*-by-*nu* matrices. See the nlhw reference page for more information about MIMO estimation.

Configure Nonlinearity Estimators

You can specify a different nonlinearity estimator than the default piecewise linear estimators.

```
m = nlhw(data,[nb,nf,nk],InputNL,OutputNL)
```

`InputNL` and `OutputNL` are nonlinearity estimator objects. If your input signal is binary, set `InputNL` to `idUnitGain`.

To use nonlinearity estimators with default settings, specify `InputNL` and `OutputNL` using constructors with no input arguments or their names as character vectors (such as 'idWaveletNetwork' for wavelet network or 'idSigmoidNetwork' for sigmoid network).

```
load iddata3;
m = nlhw(z3,[2 2 1],'idSigmoidNetwork','idDeadZone');
```

If you need to configure the properties of a nonlinearity estimator, use its object representation. For example, to estimate a Hammerstein-Wiener model that uses saturation as its input nonlinearity and one-dimensional polynomial of degree 3 as its output nonlinearity:

```
m = nlhw(z3,[2 2 1],'idSaturation',idPolynomial1D(3));
```

The third input `'idSaturation'` specifies the saturation nonlinearity with default property values. `idPolynomial1D(3)` creates a one-dimensional polynomial object of degree 3. Of course, you could have used the constructor idSaturation directly in place of the character vector 'idSaturation'.

For MIMO models, specify the nonlinearities using objects, or cell array of character vectors representing the nonlinearity names. If single name (character vector) used, the same value is applied to all the channels.

This table summarizes values that specify the nonlinearity estimators.

Nonlinearity	Value (Default Nonlinearity Configuration)	Class
Piecewise linear (default)	`'idPiecewiseLinear'`	`idPiecewiseLinear`
One layer sigmoid network	`'idSigmoidNetwork'`	`idSigmoidNetwork`
Wavelet network	`'idWaveletNetwork'`	`idWaveletNetwork`
Saturation	`'idSaturation'`	`idSaturation`
Dead zone	`'idDeadZone'`	`idDeadZone`
One-dimensional polynomial	`'idPolynomial1D'`	`idPolynomial1D`
Unit gain	`'idUnitGain'` or `[ ]`	`idUnitGain`

Additional available nonlinearities include custom networks that you create. Specify a custom network by defining a function called `gaussunit.m`, as described in the `idCustomNetwork` reference page. Define the custom network object `CNetw` as:

```
CNetw = idCustomNetwork(@gaussunit);
m = nlhw(z3,[2 2 1],'idSaturation',CNetw);
```

For more information, see "Available Nonlinearity Estimators for Hammerstein-Wiener Models" on page 12-10.

Exclude Input or Output Nonlinearity

Exclude a nonlinearity for a specific channel by specifying the `idUnitGain` value for the `InputNonlinearity` or `OutputNonlinearity` properties.

If the input signal is binary, set *InputNL* to `idUnitGain`.

For more information about model estimation and properties, see the `nlhw` and `idnlhw` reference pages.

For a description of each nonlinearity estimator, see "Available Nonlinearity Estimators for Hammerstein-Wiener Models" on page 12-10.

Iteratively Refine Model

Estimate a Hammerstein-Wiener model and then use `nlhw` command to iteratively refine the model.

```
load iddata3;
m1 = nlhw(z3,[2 2 1],idSigmoidNetwork, idWaveletNetwork);
m2 = nlhw(z3,m1);
```

Alternatively, use `pem` to refine the model.

```
m2 = pem(z3,m1);
```

Check the search termination criterion in `m.Report.Termination.WhyStop`. If `WhyStop` indicates that the estimation reached the maximum number of iterations, try repeating the estimation and possibly specifying a larger value for the `MaxIterations`.

Run 30 more iterations starting at model `m1`.

```
opt = nlhwOptions;
opt.SearchOptions.MaxIterations = 30;
m2 = nlhw(z3,m1,opt);
```

When the `m.Report.Termination.WhyStop` value is `Near (local) minimum, (norm(g) < tol` or `No improvement along the search direction with line search`, validate your model to see if this model adequately fits the data. If not, the solution might be stuck in a local minimum of the cost-function surface. Try adjusting the `SearchOptions.Tolerance` or the `SearchMethod` option of the `nlhw` option set, and repeat the estimation.

You can also try perturbing the parameters of the last model using `init`, and then refine the model using `nlhw` command.

Randomly perturb parameters of original model `m1` about nominal values.

```
m1p = init(m1);
```

Estimate the parameters of perturbed model.

```
M2 = nlhw(z3,m1p);
```

Note that using `init` does not guarantee a better solution on further refinement.

Improve Estimation Results Using Initial States

If your estimated Hammerstein-Wiener model provides a poor fit to measured data, you can repeat the estimation using the initial state values estimated from the data. By default, the initial states corresponding to the linear block of the Hammerstein-Wiener model are zero.

To specify estimating initial states during model estimation:

```
load iddata3;
opt = nlhwOptions('InitialCondition', 'estimate');
m = nlhw(z3,[2 2 1],idSigmoidNetwork,[],opt);
```

Troubleshoot Estimation

If you do not get a satisfactory model after many trials with various model structures and estimation options, it is possible that the data is poor. For example, your data might be missing important input or output variables and does not sufficiently cover all the operating points of the system.

Nonlinear black-box system identification usually requires more data than linear model identification to gain enough information about the system. See also "Troubleshooting Model Estimation" on page 17-96.

Estimate Multiple Hammerstein-Wiener Models

This example shows how to estimate and compare multiple Hammerstein-Wiener models using measured input-output data.

Load estimation and validation data.

```
load twotankdata
z = iddata(y,u,0.2);
ze = z(1:1000);
zv = z(1001:3000);
```

Estimate several models using the estimation data `ze` and different model orders, delays, and nonlinearity settings.

```
m1 = nlhw(ze,[2 3 1]);
m2 = nlhw(ze,[2 2 3]);
m3 = nlhw(ze,[2 2 3],idPiecewiseLinear('NumberofUnits',13),idPiecewiseLinear('NumberofUnits',10))
m4 = nlhw(ze,[2 2 3],idSigmoidNetwork('NumberofUnits',2),idPiecewiseLinear('NumberofUnits',10));
```

An alternative way to perform the estimation is to configure the model structure first using `idnlhw`, and then estimate the model.

```
m5 = idnlhw([2 2 3],idDeadZone,idSaturation);
m5 = nlhw(ze,m5);
```

Compare the resulting models by plotting the model outputs and the measured output in validation data `zv`.

```
compare(zv,m1,m2,m3,m4,m5)
```
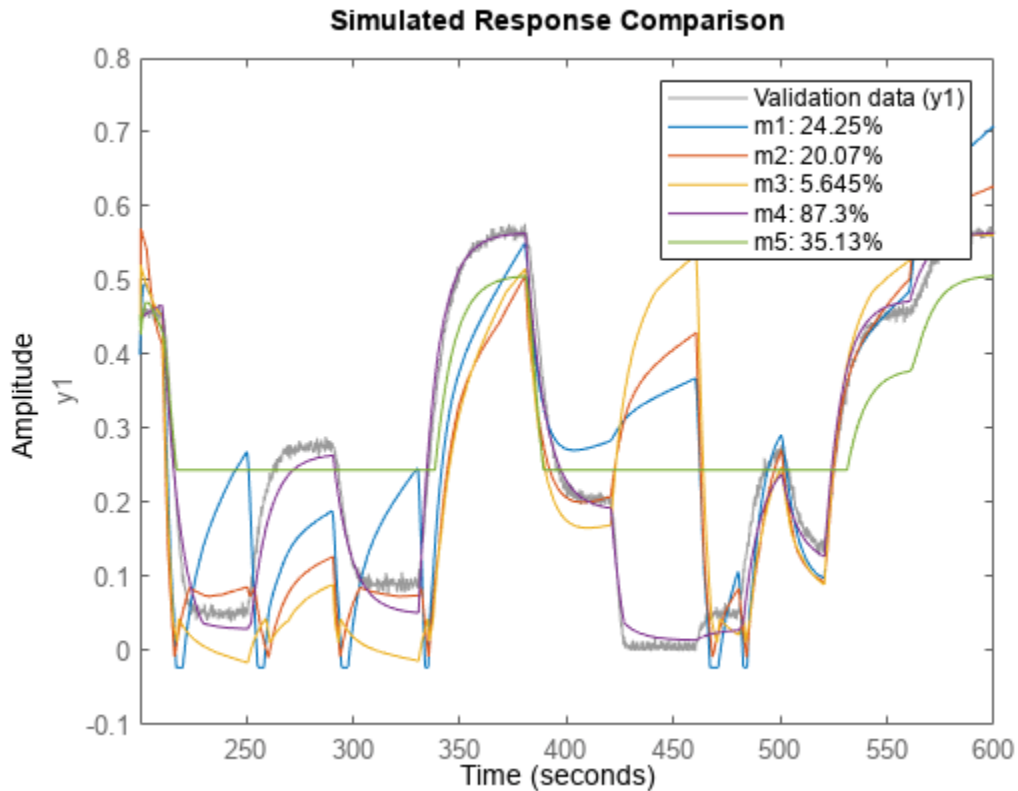
Improve a Linear Model Using Hammerstein-Wiener Structure

This example shows how to use the Hammerstein-Wiener model structure to improve a previously estimated linear model.

After estimating the linear model, insert it into the Hammerstein-Wiener structure that includes input or output nonlinearities.

Estimate a linear model.

```
load iddata1
LM = arx(z1,[2 2 1]);
```

Extract the transfer function coefficients from the linear model.

```
[Num,Den] = tfdata(LM);
```

Create a Hammerstein-Wiener model, where you initialize the linear block properties B and F using Num and Den, respectively.

```
nb = 1;        % In general, nb = ones(ny,nu)
               % ny is number of outputs and nu is number of inputs
nf = nb;
nk = 0;        % In general, nk = zeros(ny,nu)
               % ny is number of outputs and nu is number of inputs
```

```
M = idnlhw([nb nf nk],[],'idPiecewiseLinear');
M.B = Num;
M.F = Den;
```

Estimate the model coefficients, which refines the linear model coefficients in `Num` and `Den` .

```
M = nlhw(z1,M);
```

Compare responses of linear and nonlinear model against measured data.

```
compare(z1,LM,M);
```
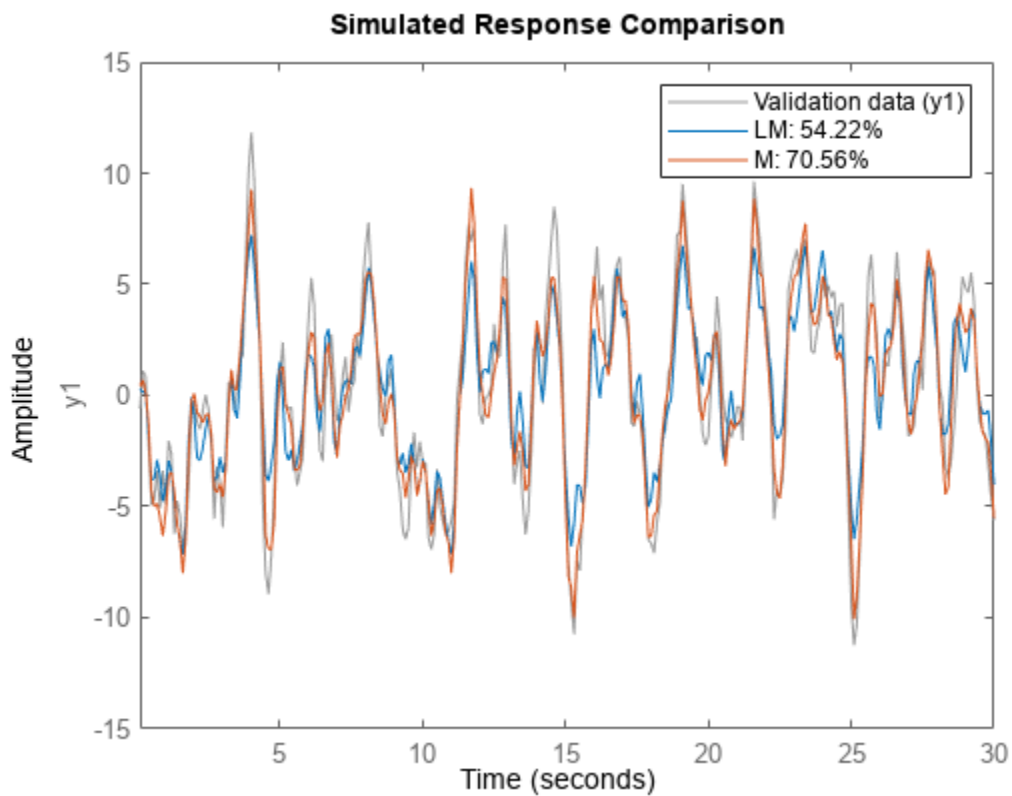

See Also

More About

- "What are Hammerstein-Wiener Models?" on page 12-2
- "Identifying Hammerstein-Wiener Models" on page 12-5
- "Validating Hammerstein-Wiener Models" on page 12-21
- "Estimate Hammerstein-Wiener Models Initialized Using Linear OE Models" on page 12-28
- "A Tutorial on Identification of Nonlinear ARX and Hammerstein-Wiener Models" on page 11-79

Validating Hammerstein-Wiener Models

After estimating a Hammerstein-Wiener model on page 12-5 for your system, you can validate whether it reproduces the system behavior within acceptable bounds. It is recommended that you use separate data sets for estimating and validating your model. If the validation indicates low confidence in the estimation, then see "Troubleshooting Model Estimation" on page 17-96 for next steps. For general information about validating models, see "Model Validation".

Compare Simulated Model Output to Measured Output

Plot simulated model output and measured output data for comparison, and compute best fit values. At the command line, use `compare` command. You can also use `sim` to simulate model response. Note that for Hammerstein-Wiener models, the simulated and predicted model output are equivalent because these models have a trivial noise-component, that is the additive disturbance in these models is white noise. For information about plotting simulated output in the app, see "Simulation and Prediction in the App" on page 17-15.

Check Iterative Search Termination Conditions

The estimation report that is generated after model estimation lists the reason the software terminated the estimation. For example, suppose that the report indicates that the estimation reached the maximum number of iterations. You can try repeating the estimation by specifying a larger value for the maximum number of iterations. For information about how to configure the maximum number of iterations and other estimation options, see "Specify Estimation Algorithm" on page 12-7.

To view the estimation report in the app, after model estimation is complete, view the **Estimation Report** area of the **Estimate** tab. At the command line, use `M.Report.Termination` to display the estimation termination conditions, where `M` is the estimated Hammerstein-Wiener model. For example, check the `M.Report.Termination.WhyStop` field that describes why the estimation was stopped.

For more information about the estimation report, see "Estimation Report" on page 1-21.

Check the Final Prediction Error and Loss Function Values

You can compare the performance of several estimated models by comparing the final prediction error and loss function values that are shown in the estimation report.

To view these values for an estimated model `M` at the command line, use the `M.Report.Fit.FPE` (final prediction error) and `M.Report.Fit.LossFcn` (value of loss function at estimation termination) properties. Smaller values typically indicate better performance. However, `M.Report.Fit.FPE` values can be unreliable when the model contains many parameters relative to the estimation data size. Use these indicators with other validation techniques to draw reliable conclusions.

Perform Residual Analysis

Residuals are differences between the model output and the measured output. Thus, residuals represent the portion of the output not explained by the model. You can analyze the residuals using

techniques such as the whiteness test and the independence test. For more information about these tests, see "What Is Residual Analysis?" on page 17-43

At the command line, use `resid` to compute, plot, and analyze the residuals. To plot residuals in the app, see "How to Plot Residuals in the App" on page 17-46.

Examine Hammerstein-Wiener Plots

A Hammerstein-Wiener plot displays the static input and output nonlinearities and linear responses of a Hammerstein-Wiener model.

Examining a Hammerstein-Wiener plot can help you determine whether you have selected a complicated nonlinearity for modeling your system. For example, suppose you use a piecewise-linear input nonlinearity to estimate your model, but the plot indicates saturation behavior. You can estimate a new model using the simpler saturation nonlinearity instead. For multivariable systems, you can use the Hammerstein-Wiener plot to determine whether to exclude nonlinearities for specific channels. If the nonlinearity for a specific input or output channel does not exhibit strong nonlinear behavior, you can estimate a new model after setting the nonlinearity at that channel to unit gain.

You can generate these plots in the **System Identification** app and at the command line. In the plot window, you can view the nonlinearities and linear responses by clicking one of the three blocks that represent the model:

- u_{NL} (*input nonlinearity*)— Click this block to view the static nonlinearity at the input to the `Linear Block`. The plot displays `evaluate(M.InputNonlinearity,u)` where `M` is the Hammerstein-Wiener model, and `u` is the input to the input nonlinearity block. For information about the blocks, see "Structure of Hammerstein-Wiener Models" on page 12-2.

- `Linear Block` — Click this block to view the Step, impulse, Bode, and pole-zero response plots of the embedded linear model (`M.LinearModel`). By default, a step plot of the linear model is displayed.

- y_{NL} (*output nonlinearity*) — Click this block to view the static nonlinearity at the output of the `Linear Block`. The plot displays `evaluate(M.OutputNonlinearity,x)`, where `x` is the output of the linear block.

Creating a Hammerstein-Wiener Plot

To create a Hammerstein-Wiener plot in the System Identification app, after you have estimated the model, select the **Hamm-Wiener** check box in the **Model Views** area. For general information about creating and working with plots in the app, see "Working with Plots" on page 21-8.

At the command line, after you have estimated a Hammerstein-Wiener model `M`, you can access the objects representing the input and output nonlinearity estimators using `M.InputNonlinearity` and `M.OutputNonlinearity`.

Use `plot` to view the shape of the nonlinearities and the properties of the linear block.

`plot(M)`

You can use additional `plot` arguments to specify the following information:

- Include several Hammerstein-Wiener models on the plot.

- Configure how to evaluate the nonlinearity at each input and output channel.
- Specify the time or frequency values for computing transient and frequency response plots of the linear block.

Configuring a Hammerstein-Wiener Plot

To configure the plots of the nonlinear blocks:

1 In the Hammerstein-Wiener Model Plot window, select the nonlinear block you want to plot.

- To plot the response of the input nonlinearity function, click the u_{NL} block.
- To plot the response of the output nonlinearity function, click the y_{NL} block.

The selected block is highlighted green.

Note The input to the output nonlinearity block y_{NL} is the output from the Linear Block and not the measured input data.

---

2 If your model contains multiple inputs or outputs, select the channel in the **Select nonlinearity at channel** list. Selecting the channel updates the plot and displays the nonlinearity values versus the corresponding input to this nonlinear block.

3 Change the range of the horizontal axis of the plot. This feature is available only for plots generated in the **System Identification** app.

In the plot window, select **Options > Set input range** to open the Range for Input to Nonlinearity dialog box. This feature is only available in the **System Identification** app.

Enter the range using the format `[MinValue MaxValue]`. Click **Apply** and then **Close** to update the plot.

To configure the linear block response plot:

1 In the Hammerstein-Wiener Model Plot window, click the `Linear Block`.

2 Select the input-output data pair for which you want to view the response in the **Select I/O pair** list.

3 Select the type of linear response plot. In the **Choose plot type** list, choose from the following options:

- `Step`
- `Impulse`
- `Bode`
- `Pole-Zero Map`

4 Set the time span for a step or impulse response plot. This feature is available only for plots generated in the **System Identification** app.

In the plot window, select **Options > Time span**. In the Time Range dialog box, specify the time span in the units of time you specified for the model. For a time span *T*, the resulting response is plotted from *-T/4* to *T*. Click **Apply** and then **Close**.

5 Set the frequency range for a Bode plot. This feature is available only for plots generated in the app.

The default frequency vector is 128 linearly distributed values, greater than zero and less than or equal to the Nyquist frequency. To change the range, select **Options > Frequency range**. In the Frequency Range dialog box, specify a new frequency vector in units of rad per model time units using one of following methods:

- MATLAB expression, such as `(1:100)*pi/100` or `logspace(-3,-1,200)`. The expression cannot contain variables in the MATLAB workspace.
- Row vector of values, such as `(1:0.1:100)`.

Click **Apply** and then **Close**.

See Also

More About
- "Identifying Hammerstein-Wiener Models" on page 12-5
- "Estimate Hammerstein-Wiener Models in the App" on page 12-12
- "Estimate Hammerstein-Wiener Models at the Command Line" on page 12-15
- "Using Hammerstein-Wiener Models" on page 12-9

How the Software Computes Hammerstein-Wiener Model Output

This topic describes how the software evaluates the output of nonlinearity estimators and uses this output to compute the response of a Hammerstein-Wiener model.

Evaluating Nonlinearities (SISO)

Evaluating the output of a nonlinearity for a input *u* requires that you first extract the input or output nonlinearity from the model:

```
F = M.InputNonlinearity;
H = M.OutputNonlinearity;
```

Evaluate *F*(*u*):

```
w = evaluate(F,u)
```

where `u` is a scalar representing the value of the input signal at a given time.

You can evaluate output at multiple time instants by evaluating *F* for several time values simultaneously using a column vector of input values:

```
w = evaluate(F,[u1;u2;u3])
```

Similarly, you can evaluate the value of the nonlinearity *H* using the output of the linear block *x*(*t*) as its input:

```
y = evaluate(H,x)
```

Evaluating Nonlinearities (MIMO)

For MIMO models, *F* and *H* are vectors of length *nu* and *ny*, respectively. *nu* is the number of inputs and *ny* is the number of outputs. In this case, you must evaluate the predicted output of each nonlinearity separately.

For example, suppose that you estimate a two-input model:

```
M = nlhw(data,[nb nf nk],[idWaveletNetwork;idPolynomial1D],'idSaturation')
```

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 an `idWaveletNetwork` object and `F2` is a `idPolynomial1D` object. `u1_value` is the first input signal and `u2_value` is the second input signal.

Evaluate F by evaluating F1 and F2 separately:

```
w1 = evaluate(F(1),u1_value);
w2 = evaluate(F(2),u2_value);
```

The total input to the linear block, w, is a vector of w1 and w2 (w = [w1 w2]).

Similarly, you can evaluate the value of the nonlinearity *H*:

```
H = M.OutputNonlinearity;
```

Simulation of Hammerstein-Wiener Model

This example shows how the software evaluates the simulated output by first computing the output of the input and output nonlinearity estimators.

Estimate a Hammerstein-Wiener model.

```
load twotankdata
estData = iddata(y,u,0.2);
M = nlhw(estData,[1 5 3],idPiecewiseLinear,idPolynomial1D);
```

Extract the input nonlinearity, linear model, and output nonlinearity as separate variables.

```
uNL = M.InputNonlinearity;
linModel = M.LinearModel;
yNL = M.OutputNonlinearity;
```

Simulate the output of the input nonlinearity estimator.

Input data for simulation

```
u = estData.u;
```

Compute output of input nonlinearity

```
w = evaluate(uNL,u);
```

Compute response of linear model to input w and zero initial conditions.

```
x = sim(linModel,w);
```

Compute the output of the Hammerstein-Wiener model M as the output of the output nonlinearity estimator to input x.

```
y = evaluate(yNL,x);
```

The previous set of commands are equivalent to directly simulating the output of M using the `sim` command.

```
ysim = sim(M,u);
```

Plot y and ysim, the manual and direct simulation results, respectively.

```
time = estData.SamplingInstants;
plot(time,y,'b',time,ysim,'--r');
xlabel('Time');
ylabel('Simulated Output')
```
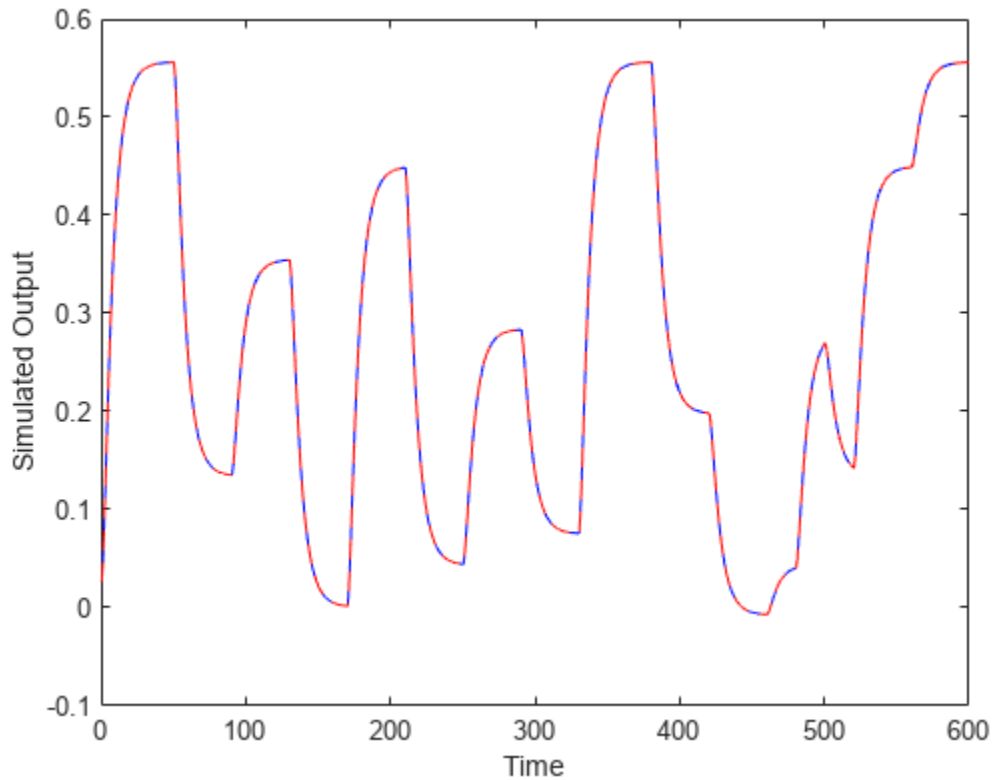
The plot indicates that `y` and `ysim` are the same.

See Also

`evaluate`

More About

*   "What are Hammerstein-Wiener Models?" on page 12-2
*   "Identifying Hammerstein-Wiener Models" on page 12-5

Estimate Hammerstein-Wiener Models Initialized Using Linear OE Models

This example shows how to estimate Hammerstein-Wiener models using linear OE models.

Load the estimation data.

```
load throttledata.mat
```

This command loads the data object `ThrottleData` into the workspace. The object contains input and output samples collected from an engine throttle system, sampled at a rate of 100Hz.

A DC motor controls the opening angle of the butterfly valve in the throttle system. A step signal (in volts) drives the DC motor. The output is the angular position (in degrees) of the valve.

Plot the data to view and analyze the data characteristics.

```
plot(ThrottleData)
```


In the normal operating range of 15-90 degrees, the input and output variables have a linear relationship. You use a linear model of low order to model this relationship.

In the throttle system, a hard stop limits the valve position to 90 degrees, and a spring brings the valve to 15 degrees when the DC motor is turned off. These physical components introduce nonlinearities that a linear model cannot capture.

Estimate a Hammerstein-Wiener model to model the linear behavior of this single-input single-output system in the normal operating range.

```
% Detrend the data because linear models cannot capture offsets.
Tr = getTrend(ThrottleData);
Tr.OutputOffset = 15;
DetrendedData = detrend(ThrottleData,Tr);

% Estimate a linear OE model with na=2, nb=1, nk=1.
opt = oeOptions('Focus','simulation');
LinearModel = oe(DetrendedData,[2 1 1],opt);
```

Compare the simulated model response with estimation data.

```
compare(DetrendedData, LinearModel)
```
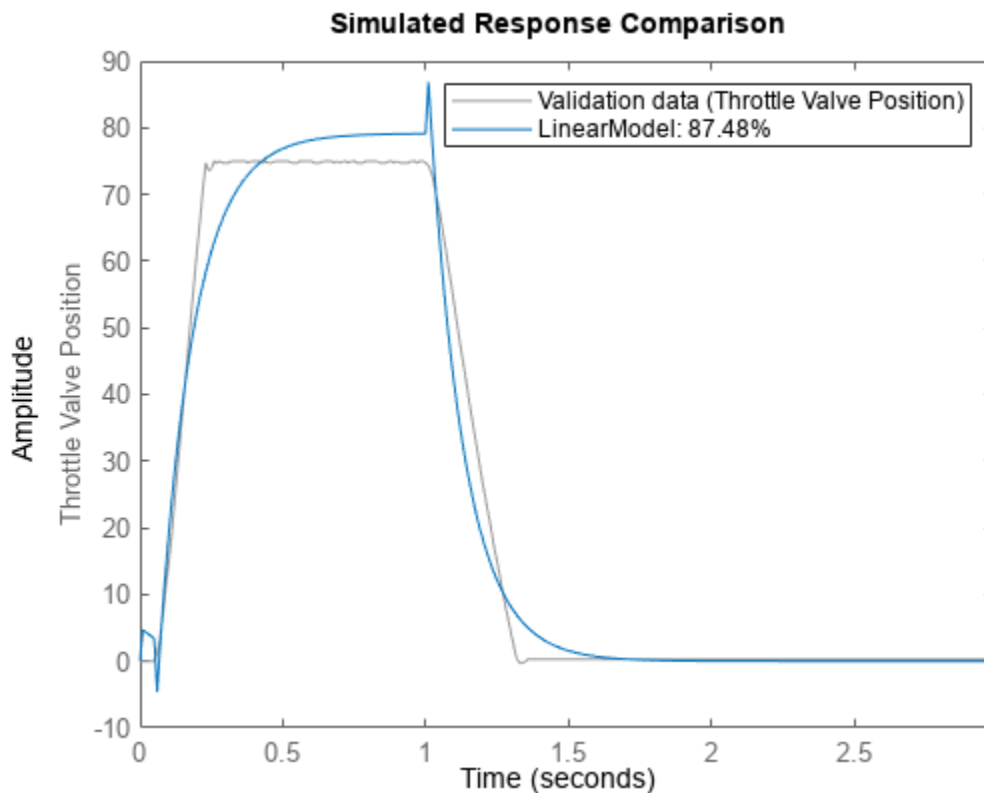

The linear model captures the rising and settling behavior in the linear operating range but does not account for output saturation at 90 degrees.

Estimate a Hammerstein-Wiener model to model the output saturation.

```
NonlinearModel = nlhw(ThrottleData, LinearModel, [], idSaturation);
```

The software uses the orders and delay of the linear model for the orders of the nonlinear model. In addition, the software uses the B and F polynomials of the linear transfer function.

Compare the nonlinear model with data.

```
compare(ThrottleData, NonlinearModel)
```
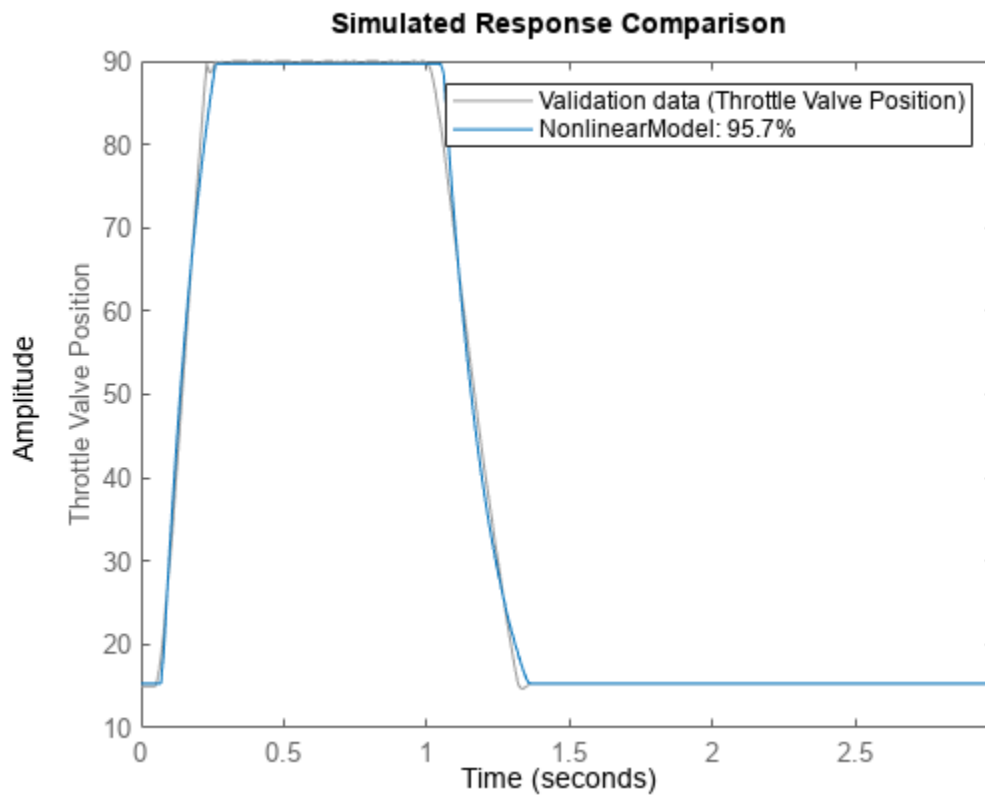

See Also

More About

- "What are Hammerstein-Wiener Models?" on page 12-2
- "Identifying Hammerstein-Wiener Models" on page 12-5
- "Estimate Hammerstein-Wiener Models at the Command Line" on page 12-15

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 "Estimate State-Space Models with Structured Parameterization" on page 7-32.

See Also
`idgrey` | `idnlgrey` | `idss`

Related Examples
- "Estimate Linear Grey-Box Models" on page 13-6
- "Estimate Nonlinear Grey-Box Models" on page 13-25

More About
- "Data Supported by Grey-Box Models" on page 13-3
- "Choosing idgrey or idnlgrey Model Object" on page 13-4

Data Supported by Grey-Box Models

You can estimate both continuous-time or discrete-time grey-box models for data with the following characteristics:

- Time-domain or frequency-domain data, including time-series data with no inputs.

  Note Nonlinear grey-box models support only time-domain data.

- Single-output or multiple-output data

You must first import your data into the MATLAB workspace. You must represent your data as an `iddata` or `idfrd` object. For more information about preparing data for identification, see "Data Preparation".

See Also

idgrey | idnlgrey | idss

Related Examples

- "Estimate Linear Grey-Box Models" on page 13-6
- "Estimate Nonlinear Grey-Box Models" on page 13-25

More About

- "Supported Grey-Box Models" on page 13-2
- "Choosing idgrey or idnlgrey Model Object" on page 13-4

Choosing idgrey or idnlgrey Model Object

Grey-box models require that you specify the structure of the ODE model in a file. You use this file to create the `idgrey` or `idnlgrey` model object. You can use both the `idgrey` and the `idnlgrey` objects to model linear systems. However, you can only represent nonlinear dynamics using the `idnlgrey` model object.

The `idgrey` object requires that you write a function to describe the linear dynamics in the state-space form, such that this file returns the state-space matrices as a function of your parameters. For more information, see "Specifying the Linear Grey-Box Model Structure" on page 13-6.

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 13-25.

The following table compares `idgrey` and `idnlgrey` model objects.

Comparison of idgrey and idnlgrey Objects

Settings and Operations	Supported by idgrey?	Supported by idnlgrey?
Set bounds on parameter values.	Yes	Yes
Handle initial states individually.	Yes	Yes
Perform linear analysis.	Yes For example, use the `bode` command.	No
Honor stability constraints.	Yes Specify constraints using the `Advanced.StabilityThreshold` estimation option. For more information, see `greyestOptions`.	No **Note** You can use parameter bounds to ensure stability of an `idnlgrey` model, if these bounds are known.
Estimate a disturbance model.	Yes The disturbance model is represented by K in state-space equations.	No
Optimize estimation results for simulation or prediction.	Yes Set the `Focus` estimation option to `'simulation'` or `'prediction'`. For more information, see `greyestOptions`.	No Because `idnlgrey` models are Output-Error models, there is no difference between simulation and prediction results.

See Also

idgrey | idnlgrey | idss

Related Examples

- "Estimate Linear Grey-Box Models" on page 13-6
- "Estimate Nonlinear Grey-Box Models" on page 13-25

More About

- "Supported Grey-Box Models" on page 13-2
- "Data Supported by Grey-Box Models" on page 13-3

Estimate Linear Grey-Box Models

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 nth-order differential equations.

The first step in grey-box modeling is to write a function that returns state-space matrices as a function of user-defined parameters and information about the model.

Use the following format to implement the linear grey-box model in the file:

`[A,B,C,D] = myfunc(par1,par2,...,parN,Ts,aux1,aux2,...)`

where the output arguments are the state-space matrices and `myfunc` is the name of the file. `par1,par2,...,parN` are the N parameters of the model. Each entry may be a scalar, vector or matrix.`Ts` is the sample time. `aux1,aux2,...` are the optional input arguments that `myfunc` uses to compute the state-space matrices in addition to the parameters and sample time. `aux` contains auxiliary variables in your system. You use auxiliary variables to vary system parameters at the input to the function, and avoid editing the file.

You can write the contents of `myfunc` to parameterize either a continuous-time, or a discrete-time state-space model, or both. When you create the linear grey-box model using `myfunc`, you can specify the nature of the output arguments of `myfunc`. The continuous-time state-space model has the form:

In continuous-time, the state-space description has the following form:

$$\dot{x}(t) = Ax(t) + Bu(t) + Ke(t)$$
$$y(t) = Cx(t) + Du(t) + e(t)$$
$$x(0) = x0$$

where, A,B,C and D are matrices that are parameterized by the parameters `par1,par2,...,parN`. The noise matrix K and initial state vector, $x0$, are not parameterized by `myfunc`. In some applications, you may want to express K and $x0$ as quantities that are parameterized by chosen parameters, just as the A, B, C and D matrices. To handle such cases, you can write the ODE file, `myfunc`, to return K and $x0$ as additional output arguments:

`[A,B,C,D,K,x0] = myfunc(par1,par2,...,parN,Ts,aux1,aux2,...)`

K and $x0$ are thus treated in the same way as the A, B, C and D matrices. They are all functions of the parameters `par1,par2,...,parN`. To configure the handling of initial states, $x0$, and the disturbance component, K, during estimation, use the `greyestOptions` option set.

In discrete-time, the state-space description has a similar form:

$$x(k + 1) = Ax(k) + Bu(k) + Ke(t)$$
$$y(k) = Cx(k) + Du(k) + e(t)$$
$$x(0) = x0$$

where, *A*, *B*, *C* and *D* are now the discrete-time matrices that are parameterized by the parameters `par1,par2,...,parN`. *K* and *x0* are not directly parameterized, but can be estimated if required by configuring the corresponding estimation options.

After creating the function or MEX-file with your model structure, you must define an `idgrey` model object.

Create Function to Represent a Grey-Box Model

This example shows how to represent the structure of the following continuous-time model:

$$\dot{x}(t) = \begin{bmatrix} 0 & 1 \\ 0 & \theta_1 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ \theta_2 \end{bmatrix} u(t)$$

$$y(t) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x(t) + e(t)$$

$$x(0) = \begin{bmatrix} \theta_3 \\ 0 \end{bmatrix}$$

This equation represents an electrical motor, where $y_1(t) = x_1(t)$ is the angular position of the motor shaft, and $y_2(t) = x_2(t)$ is the angular velocity. The parameter $-\theta_1$ is the inverse time constant of the motor, and $-\theta_2/\theta_1$ is the static gain from the input to the angular velocity.

The motor is at rest at `t` = 0, but its angular position θ_3 is unknown. Suppose that the approximate nominal values of the unknown parameters are $\theta_1 = -1$, $\theta_2 = 0.25$ and $\theta_3 = 0$. For more information about this example, see the section on state-space models in *System Identification: Theory for the User* , Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

The continuous-time state-space model structure is defined by the following equation:

$$\dot{x}(t) = Fx(t) + Gu(t) + \tilde{K}w(t)$$
$$y(t) = Hx(t) + Du(t) + w(t)$$
$$x(0) = x0$$

If you want to estimate the same model using a structured state-space representation, see "Estimate Structured Continuous-Time State-Space Models" on page 7-35.

To prepare this model for estimation:

• Create the following file to represent the model structure in this example:

```
function [A,B,C,D,K,x0] = myfunc(par,T)
A = [0 1; 0 par(1)];
B = [0;par(2)];
C = eye(2);
D = zeros(2,1);
K = zeros(2,2);
x0 = [par(3);0];
```

Save the file such that it is in the MATLAB® search path.

- Use the following syntax to define an `idgrey` model object based on the `myfunc` file:

```
par = [-1; 0.25; 0];
aux = {};
T = 0;
m = idgrey('myfunc',par,'c',aux,T);
```

where `par` represents a vector of all the user-defined parameters and contains their nominal (initial) values. In this example, all the scalar-valued parameters are grouped in the `par` vector. The scalar-valued parameters could also have been treated as independent input arguments to the ODE function `myfunc`. `'c'` specifies that the underlying parameterization is in continuous time. `aux` represents optional arguments. As `myfunc` does not have any optional arguments, use `aux = {}`. `T` specifies the sample time; `T = 0` indicates a continuous-time model.

Load the estimation data.

```
load(fullfile(matlabroot,'toolbox','ident','iddemos','data','dcmotordata'));
data = iddata(y,u,0.1);
```

Use `greyest` to estimate the grey-box parameter values:

```
m_est = greyest(data,m);
```

where `data` is the estimation data and `m` is an estimation initialization `idgrey` model. `m_est` is the estimated `idgrey` model.

See Also

`idgrey` | `greyest`

Related Examples

- "Estimate Continuous-Time Grey-Box Model for Heat Diffusion" on page 13-9
- "Estimate Discrete-Time Grey-Box Model with Parameterized Disturbance" on page 13-12
- "Estimate Coefficients of ODEs to Fit Given Solution" on page 13-14
- "Estimate Model Using Zero/Pole/Gain Parameters" on page 13-21
- "Estimate Nonlinear Grey-Box Models" on page 13-25

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:

$$k \cdot \Delta L \leq \xi < (k + 1)\Delta L$$

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.ExtraArguments = {20,1,22};
me = greyest(data,m);
```

This syntax uses the `ExtraArguments` 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.

See Also

`idgrey` | `greyest`

Related Examples

- "Estimate Linear Grey-Box Models" on page 13-6
- "Estimate Discrete-Time Grey-Box Model with Parameterized Disturbance" on page 13-12
- "Estimate Coefficients of ODEs to Fit Given Solution" on page 13-14
- "Estimate Model Using Zero/Pole/Gain Parameters" on page 13-21
- "Estimate Nonlinear Grey-Box Models" on page 13-25

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:

$$x(kT + T) = \begin{bmatrix} par1 & par2 \\ 1 & 0 \end{bmatrix} x(kT) + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u(kT) + w(kT)$$
$$y(kT) = [par3 \ par4]x(kT) + e(kT)$$
$$x(0) = x0$$

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 13-12 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 2-34 or "Representing Frequency-Response Data Using idfrd Objects" on page 2-61.

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);
```

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)
```

See Also
kalman | idgrey | greyest

Related Examples

- "Estimate Linear Grey-Box Models" on page 13-6
- "Estimate Continuous-Time Grey-Box Model for Heat Diffusion" on page 13-9
- "Estimate Coefficients of ODEs to Fit Given Solution" on page 13-14
- "Estimate Model Using Zero/Pole/Gain Parameters" on page 13-21
- "Estimate Nonlinear Grey-Box Models" on page 13-25

More About

- "Identifying State-Space Models with Separate Process and Measurement Noise Descriptions" on page 13-55

Estimate Coefficients of ODEs to Fit Given Solution

This example shows how to estimate model parameters 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 `greyest`.

- For nonlinear dynamics, represent the model using a nonlinear grey-box model (`idnlgrey`). Estimate the model coefficients using `nlgreyest`.

In this example, you estimate the value of the friction coefficient of a simple pendulum using its oscillation data. The equation of motion of a simple pendulum is:

$$ml^2\ddot{\theta} + b\dot{\theta} + mgl\sin\theta = 0$$

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 is estimated to fit the given angular displacement data. There is no external driving force that is contributing to the pendulum motion.

Load measured data.

```
load(fullfile(matlabroot,'toolbox','ident', ...
    'iddemos','data','pendulumdata'));
data = iddata(y,[],0.1,'Name','Pendulum');
data.OutputName = 'Pendulum position';
data.OutputUnit = 'rad';
data.Tstart = 0;
data.TimeUnit = 's';
```

The measured angular displacement data is loaded and saved as `data`, an `iddata` object with a sample 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:

$$X(t) = AX(t) + Bu(t)$$
$$y(t) = CX(t) + Du(t)$$

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)
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. Ts is the sample time. Save this function as `LinearPendulum.m`. The function `LinearPendulum` must be on the MATLAB® path. Alternatively, you can specify the full path name for this function.

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 `'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. Also force the estimation algorithm to return a stable model. This option is available only for linear model (idgrey) estimation.

```
opt = greyestOptions('InitialState','estimate','Display','on');
opt.EnforceStability = true;
```

5. Estimate the viscous friction coefficient.

```
linear_model = greyest(data,linear_model,opt);
```

```
Estimation data: Time domain data Pendulum
Data has 1 outputs, 0 inputs and 1001 samples.
ODE Function: LinearPendulum
Function type: 'c'
Number of parameters: 4
```

ESTIMATION PROGRESS

```
Algorithm: Nonlinear least squares with automatically chosen line search method
------------------------------------------------------------------------------
-------
                        Norm of     First-order    Improvement (%)
Iteration      Cost      step        optimality    Expected  Achieved   Bisections
------------------------------------------------------------------------------
-------
    0       0.0494819      -            450           70.8        -          -
    1       0.0144883    2.62        1.29e+03         70.8       70.7        0
    2       0.0132907    0.499       1.42e+03          11        8.27        0
    3       0.0125849    0.433         739            8.68       5.31        0
    4       0.0124825    0.236         418            1.77       0.814       0
    5       0.0124306    0.139         281           0.803      0.415        0
    6       0.0124187    0.0783        137           0.217      0.0963       0
    7       0.0124141    0.0447        92.9          0.0788     0.037        0
```

RESULT

```
Termination condition: Near (local) minimum, (norm(g) < tol)..
Number of iterations: 9, Number of function evaluations: 19

Status: Estimated using GREYEST
Fit to estimation data: 49.92%, FPE: 0.012487
```

■ Stop   Close

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')
```

```
linear_b_est =

    0.1178
```

```
dlinear_b_est =

    0.0088
```

getpvec returns, `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 returned in `linear_b_est`.

6. Compare the response of the linear grey-box model to the measured data.

```
compare(data,linear_model)
```
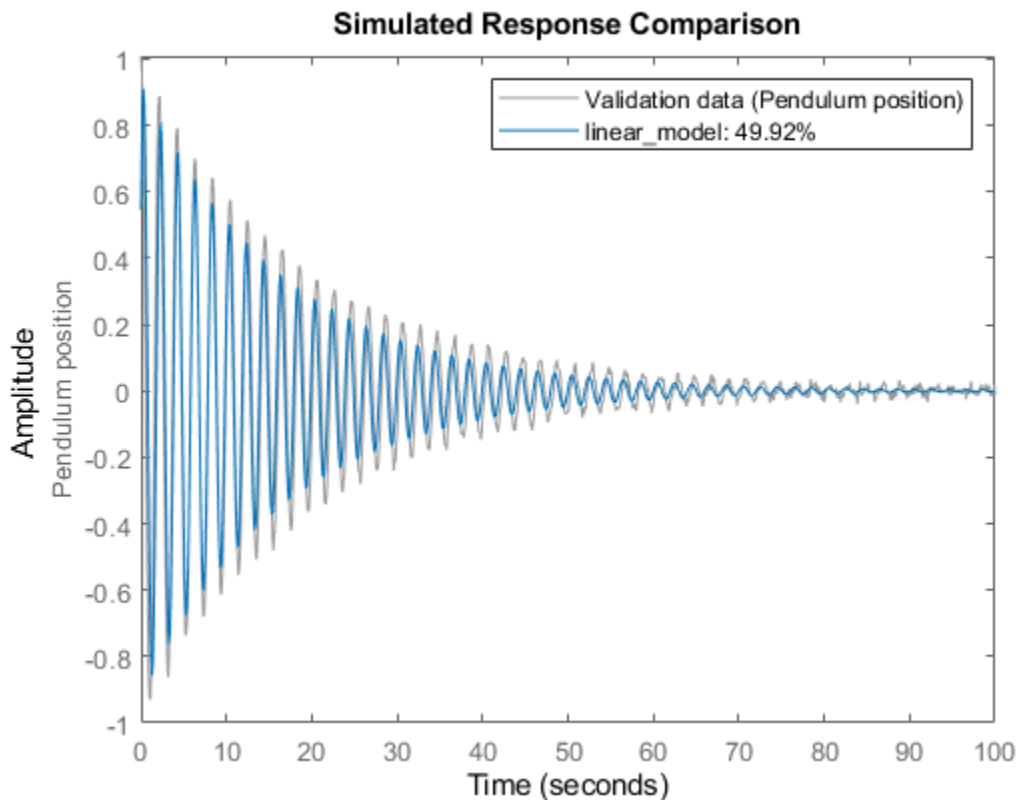

The linear grey-box estimation model provides a 49.9% fit to measured data. The poor fit is due to the assumption that the pendulum undergoes small angular displacements, whereas the measured data shows large oscillations.

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)

% 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` on the MATLAB® path. Alternatively, you can specify the full path name for this function.

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 = nlgreyest(data,nonlinear_model,'Display','Full');
```

The `nlgreyest` command updates the parameter of `nonlinear_model`.

```
b_est = nonlinear_model.Parameters(4).Value;
[nonlinear_b_est, dnonlinear_b_est] = getpvec(nonlinear_model,'free')


nonlinear_b_est =
```

```
    0.1002


dnonlinear_b_est =

    0.0149
```

`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 returned in `nonlinear_b_est`.

5. Compare the response of the linear and nonlinear grey-box models to the measured data.

```
compare(data,linear_model,nonlinear_model)
```
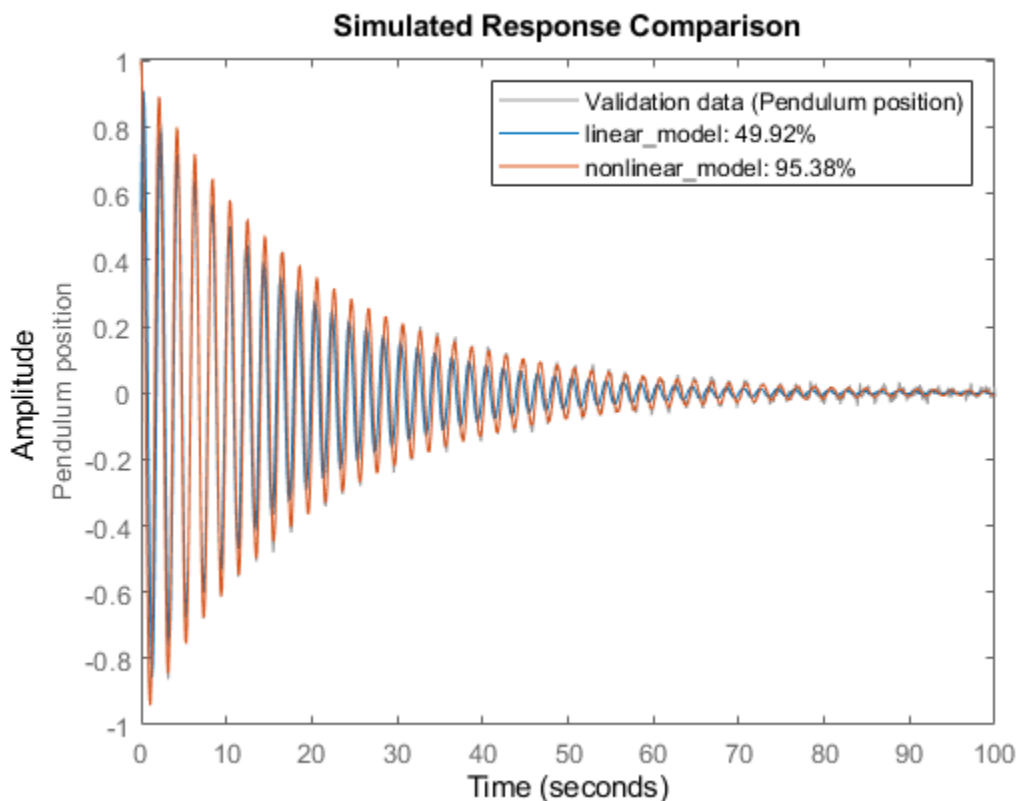

The nonlinear grey-box model estimation provides a closer fit to the measured data.

See Also
idgrey | idnlgrey | greyest | nlgreyest

Related Examples
- "Estimate Linear Grey-Box Models" on page 13-6

Estimate Model Using Zero/Pole/Gain Parameters

This example shows how to estimate a model that is parameterized by poles, zeros, and gains. The example requires Control System Toolbox™ software.

You 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.

Load the measured data.

```
load zpkestdata zd;
```

The variable zd, which contains measured data, is loaded into the MATLAB® workspace.

```
plot(zd);
```
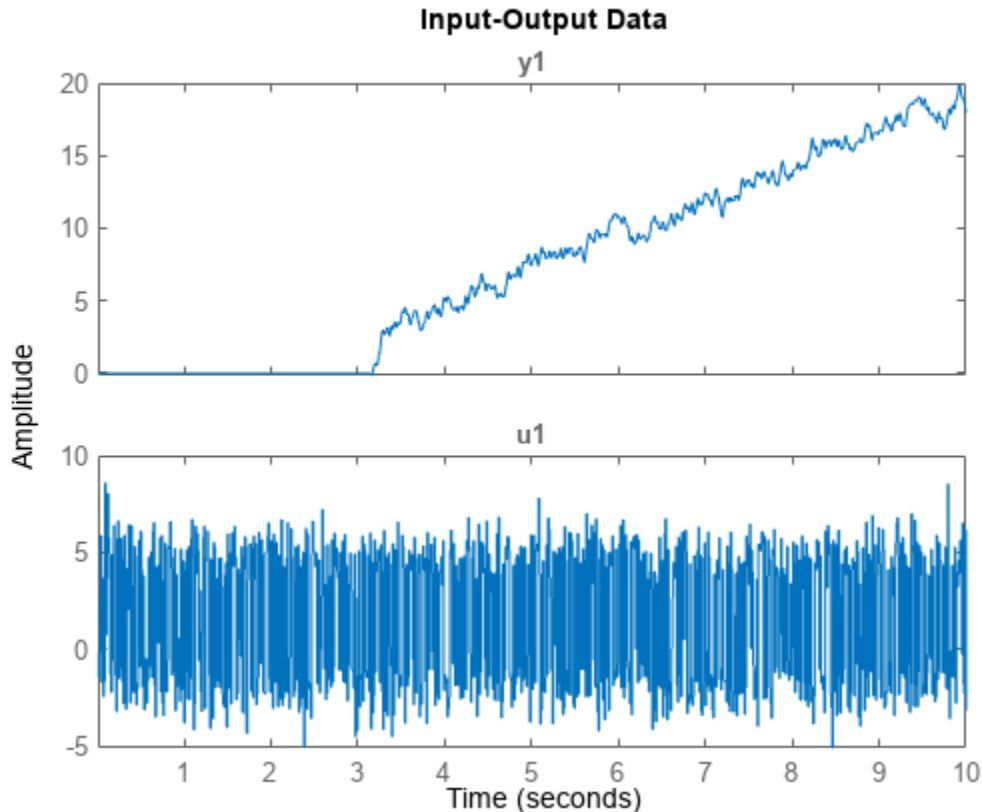

The output shows an input delay of approximately 3.14 seconds.

Estimate the model using the zero-pole-gain (zpk) form using the `zpkestODE` function. To view this function, enter

```
type zpkestODE
```

```
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.

%   Copyright 2011 The MathWorks, Inc.

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
```

Create a linear grey-box model associated with the ODE function.

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.

```
compareOpt = compareOptions('InitialCondition','zero');
compare(zd,init_sys,compareOpt);
```

Simulated Response Comparison
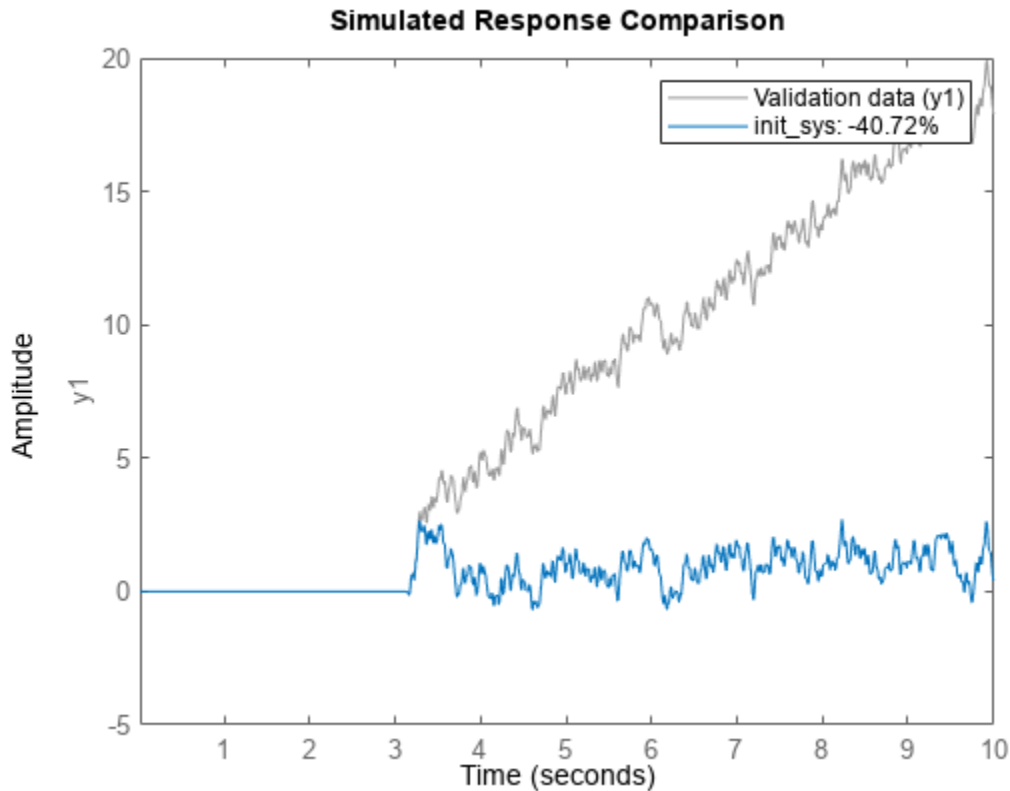
The initial model provides a poor fit.

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)]
```

ans = 10×2 complex

```
  -0.5000 + 1.0000i   -1.6158 + 1.6173i
  -0.5000 - 1.0000i   -1.6158 - 1.6173i
  -0.5000 + 0.0000i   -0.9416 + 0.0000i
  -1.0000 + 0.0000i   -1.4100 + 0.0000i
  -1.1100 + 2.0000i   -2.4050 + 1.4340i
  -1.1100 - 2.0000i   -2.4050 - 1.4340i
  -3.0100 + 0.0000i   -2.3388 + 0.0000i
  -4.0100 + 0.0000i   -2.3393 + 0.0000i
  -0.0200 + 0.0000i   -0.0082 + 0.0000i
```

```
    10.1000 + 0.0000i    9.7881 + 0.0000i
```

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,compareOpt);
```
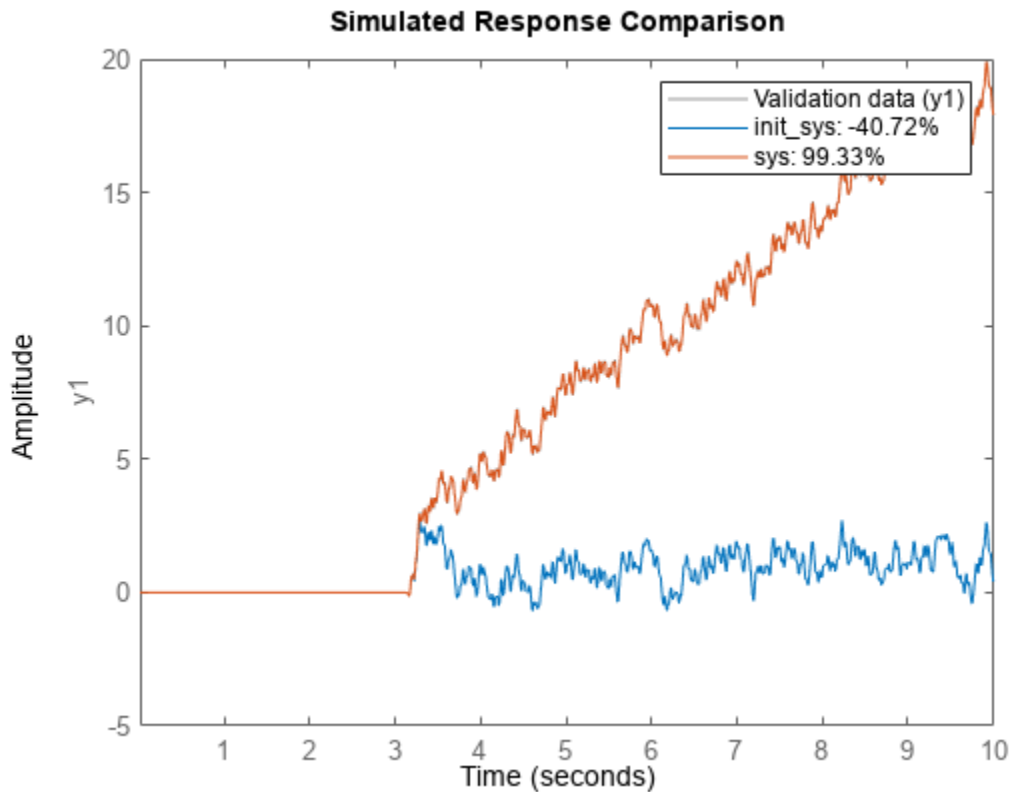

`sys` provides a closer fit (98.35%) to the measured data.

See Also
`idgrey` | `greyest` | `getpvec` | `ssdata` | `c2d`

Related Examples
- "Estimate Linear Grey-Box Models" on page 13-6
- "Estimate Coefficients of ODEs to Fit Given Solution" on page 13-14
- "Estimate Continuous-Time Grey-Box Model for Heat Diffusion" on page 13-9
- "Estimate Discrete-Time Grey-Box Model with Parameterized Disturbance" on page 13-12
- "Estimate Nonlinear Grey-Box Models" on page 13-25

Estimate Nonlinear Grey-Box Models

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 sample time. 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. For more information about MEX-files, see "Write C Functions Callable from MATLAB (MEX Files)". This file is called an ODE file or a model file.

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. Also check that for the expected input and parameter value ranges, the model output and derivatives remain finite.

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, define an `idnlgrey` object. This object shares many of the properties of the linear `idgrey` model object.

Use the following general 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.

You can also specify additional properties of the `idnlgrey` model, including simulation method and related options. For detailed information about this object and its properties, see the `idnlgrey` reference page.

Use `nlgreyest` or `pem` to estimate your grey-box model. Before estimating, it is advisable to simulate the model to verify that the model file has been coded correctly. For example, compute the model response to estimation data's input signal using `sim`:

```
y = sim(model,data)
```

where, `model` is the `idnlgrey` object, and `data` is the estimation data (`iddata` object).

Using nlgreyest to Estimate Nonlinear Grey-Box Models

You can use the `nlgreyest` 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:

```
m2 = nlgreyest(data,m)
```

where `data` is the estimation data and `m` is the `idnlgrey` model object you constructed. The output `m2` is an `idnlgrey` model of same configuration as `m`, with parameters and initial states updated to fit the data. More information on estimation can be retrieved from the `Report` property. For more information on `Report` and how to use it, see "Output Arguments" in the `nlgreyest` reference page, or type `m2.Report` on the command line.

You can specify additional estimation options using the `nlgreyestOptions` option set, including `SearchMethod` and `SearchOption`.

For information about validating your models, see "Model Validation".

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" on page 13-60.

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', 'dcmotordata'));
```

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.

```
z.InputName = 'Voltage';
z.InputUnit =  'V';
z.OutputName = {'Angular position', 'Angular velocity'};
z.OutputUnit = {'rad', 'rad/s'};
z.Tstart = 0;
z.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 position).
figure('Name', [z.Name ': Voltage input -> Angular velocity output']);
plot(z(:, 2, 1));   % Plot second input-output pair (Voltage -> Angular velocity).
```

Input-Output Data

Angular velocity

Voltage

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" on page 13-45.

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 velocity.
      -(1/tau)*x(2)+(k/tau)*u(1)  ... % Angular acceleration.
     ];
```

The file must always be structured to return the following:

Output arguments:

- **dx** is the vector of state derivatives in continuous-time case, and state update values in the discrete-time case.

- **y** is the output equation

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.

```
nlgr = setinit(nlgr, 'Name', {'Angular position' 'Angular velocity'});
nlgr = setinit(nlgr, 'Unit', {'rad' 'rad/s'});
nlgr = setpar(nlgr, 'Name', {'Time-constant' 'Static gain'});
nlgr = setpar(nlgr, 'Unit', {'s' 'rad/(V*s)'});
```

You can also use **setinit** and **setpar** to assign values, minima, maxima, and estimation status for all initial states or parameters simultaneously.

5. View the initial model.

a. Get basic information about the model.

The DC-motor has 2 (initial) states and 2 model parameters.

```
size(nlgr)
```

```
Nonlinear grey-box model with 2 outputs, 1 inputs, 2 states and 2 parameters (2 free).
```

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 `help idnlgrey.InitialStates` and `help idnlgrey.Parameters` for more information.

```
nlgr.InitialStates(1)
nlgr.Parameters(2)


ans =

  struct with fields:

       Name: 'Angular position'
       Unit: 'rad'
      Value: 0
    Minimum: -Inf
    Maximum: Inf
      Fixed: 1


ans =

  struct with fields:

       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 =

  2x1 cell array

    {[1]}
    {[1]}


ans =

  2x1 cell array

    {[-Inf]}
```

```
    {[-Inf]}
```

d. Obtain basic information about the object:

`nlgr`

```
nlgr =

Continuous-time nonlinear grey-box model defined by 'dcmotor_m' (MATLAB file):

   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(s), 2 state(s), 2 output(s), and 2 free parameter(s) (out of 2).

Name: DC-motor

Status:
Created by direct construction or transformation. Not estimated.
```

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: {}
    SimulationOptions: [1x1 struct]
        TimeVariable: 't'
       NoiseVariance: [2x2 double]
           InputName: {'Voltage'}
           InputUnit: {'V'}
          InputGroup: [1x1 struct]
          OutputName: {2x1 cell}
          OutputUnit: {2x1 cell}
         OutputGroup: [1x1 struct]
               Notes: [0x1 string]
            UserData: []
                Name: 'DC-motor'
                  Ts: 0
            TimeUnit: 'seconds'
              Report: [1x1 idresults.nlgreyest]
```

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). The simulation options are specified using the "SimulationOptions" model property.

1. Set the absolute and relative error tolerances to small values (`1e-6` and `1e-5`, respectively).

```
nlgr.SimulationOptions.AbsTol = 1e-6;
nlgr.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.

```
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 `nlgreyest`, which is a prediction error minimization method for nonlinear grey box models. The estimation options, such as the choice of estimation progress display, are specified using the "nlgreyestOptions" option set.

```
nlgr = setinit(nlgr, 'Fixed', {false false}); % Estimate the initial states.
opt = nlgreyestOptions('Display', 'on');
nlgr = nlgreyest(z, nlgr, opt);
```

```
Nonlinear Grey Box Model Estimation
Data has 2 outputs, 1 inputs and 400 samples.
ODE Function: dcmotor_m
Number of parameters: 2
```

ESTIMATION PROGRESS

```
Algorithm: Trust-Region Reflective Newton
---------------------------------------------------------------
                             Norm of      First-order
Iteration      Cost           step         optimality
---------------------------------------------------------------
     0         1.39351          -               -
     1         0.111682       0.827          1.89e+03
     2         0.0596697      0.106            44.4
     3         0.0593534      0.0124           0.661
     4         0.0593527      0.000834         0.0804
     5         0.0593527      5.65e-05         0.00014
---------------------------------------------------------------
```

RESULT

```
Termination condition: Change in cost was less than the specified tolerance..
Number of iterations: 5, Number of function evaluations: 6

Status: Estimated using NLGREYEST
Fit to estimation data: [98.34;84.47]%, FPE: 0.0010963
```

■ Stop | Close |

Performance Evaluation of the Estimated DC-Motor Model

1. Review the information about the estimation process.

This information is stored in the `Report` 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.Report
fprintf('\n\nThe search termination condition:\n')
nlgr.Report.Termination
```

```
ans =

          Status: 'Estimated using NLGREYEST'
          Method: 'Solver: ode45; Search: lsqnonlin'
             Fit: [1x1 struct]
      Parameters: [1x1 struct]
     OptionsUsed: [1x1 idoptions.nlgreyest]
       RandState: []
        DataUsed: [1x1 struct]
     Termination: [1x1 struct]


The search termination condition:

ans =

  struct with fields:

                 WhyStop: 'Change in cost was less than the specified tolerance.'
              Iterations: 5
      FirstOrderOptimality: 1.4013e-04
                FcnCount: 6
               Algorithm: 'trust-region-reflective'
```

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.

```
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]);
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).

```
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. Use the `resid` command to view the correlations among the residuals. The first column of plots shows the autocorrelations of the residuals for the two outputs. The second column shows the cross-correlation of these residuals with the input "Voltage". The correlations are within acceptable bounds (blue region).

```
figure('Name',[nlgr.Name ': residuals of estimated model']);
resid(z,nlgr);
```

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.

```
getcov(nlgr)
nlgr.NoiseVariance
```

```
ans =

   1.0e-04 *

    0.1573    0.0021
    0.0021    0.0008


ans =

    0.0010   -0.0000
```

```
   -0.0000    0.0110
```

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);
```

```
nlgr =

Continuous-time nonlinear grey-box model defined by 'dcmotor_m' (MATLAB file):

   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(s), 2 state(s), 2 output(s), and 2 free parameter(s) (out of 2).

 Inputs:
    u(1)  Voltage(t) [V]
 States:                              Initial value
    x(1)  Angular position(t) [rad]     xinit@exp1    0.0302675   (estimated) in [-Inf, Inf]
    x(2)  Angular velocity(t) [rad/s]   xinit@exp1   -0.133777    (estimated) in [-Inf, Inf]
 Outputs:
    y(1)  Angular position(t) [rad]
    y(2)  Angular velocity(t) [rad/s]
 Parameters:                          Value Standard Deviation
    p1   Time-constant [s]               0.243649       0.00396671   (estimated) in [-Inf, Inf]
    p2   Static gain [rad/(V*s)]         0.249644       0.000284486  (estimated) in [-Inf, Inf]

Name: DC-motor

Status:
Termination condition: Change in cost was less than the specified tolerance..
Number of iterations: 5, Number of function evaluations: 6

Estimated using Solver: ode45; Search: lsqnonlin on time domain data "DC-motor".
Fit to estimation data: [98.34;84.47]%
FPE: 0.001096, MSE: 0.1187
More information in model's "Report" property.

<a href="matlab:if exist('nlgr','var'), if isa(nlgr,'idnlgrey'), disp(' '); disp(get(nlgr)), else
```

Conclusions

This example illustrates the basic tools for performing nonlinear grey-box modeling. See the other nonlinear grey-box examples to learn about:

*   Using nonlinear grey-box models in more advanced modeling situations, such as building nonlinear continuous- and discrete-time, time-series and static models.

*   Writing and using C MEX model-files.

*   Handling nonscalar parameters.

*   Impact of certain algorithm choices.

For more information on identification of dynamic systems with System Identification Toolbox, visit the System Identification Toolbox product information page.

Nonlinear Grey-Box Model Properties and Estimation Options

`idnlgrey` creates a nonlinear grey-box model based on the model structure and properties. The parameters and initial states of the created `idnlgrey` object are estimated using `nlgreyest`.

The following model properties and estimation options affect the model creation and estimation results.

Simulation Method

You specify the simulation method using the `SimulationOptions` (`struct`) property of `idnlgrey`.

System Identification Toolbox software provides several variable-step and fixed-step solvers for simulating `idnlgrey` models.

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.

To view a list of available solvers and their properties, see the `SimulationOptions` model property in `idnlgrey` reference page.

Search Method

You specify the search method for estimating model parameters using the `SearchMethod` option of the `nlgreyestOptions` option set. 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` cannot handle automatic weighting by the inverse of estimated noise variance in multi-output cases. For more information, see `OutputWeight` estimation option in the `nlgreyestOptions` reference page.

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.

For detailed information about this and other `nlgreyest` estimation options, see `nlgreyestOptions`.

Gradient Options

You specify the method for calculating gradients using the `GradientOptions` option of the `nlgreyestOptions` option set. *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.

For detailed information about this and other `nlgreyest` estimation options, see `nlgreyestOptions`.

See Also
`idnlgrey` | `nlgreyest`

Related Examples
- "Creating IDNLGREY Model Files" on page 13-45
- "Estimate Linear Grey-Box Models" on page 13-6

More About
- "Supported Grey-Box Models" on page 13-2
- "Data Supported by Grey-Box Models" on page 13-3

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 specifies 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 sample 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.

% Output equation.
y = x(2);                                          % Water level, lower tank.

% State equations.
dx = [1/A1*(k*u(1)-a1*sqrt(2*g*x(1)));             ... % Water level, upper tank.
      1/A2*(a1*sqrt(2*g*x(1))-a2*sqrt(2*g*x(2)))   ... % Water level, lower tank.
     ];
```

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_TEMPLATE.c')
```

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 2005-2015 The MathWorks, Inc. */
/*   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;
    size_t  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.

1   High-precision variables (all inputs, states, outputs and parameters of an IDNLGREY object) should be defined to be of the data type "double".

2   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_dx 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.

3   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.

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.

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".

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 values of initial states.
nlgr_m      = idnlgrey('twotanks_m', Order, Parameters, InitialStates, 0)

nlgr_m =

Continuous-time nonlinear grey-box model defined by 'twotanks_m' (MATLAB 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(s), 2 state(s), 1 output(s), and 6 free parameter(s) (out of 6).

Status:
Created by direct construction or transformation. Not estimated.

nlgr_cmex = idnlgrey('twotanks_c', Order, Parameters, InitialStates, 0)

nlgr_cmex =

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(s), 2 state(s), 1 output(s), and 6 free parameter(s) (out of 6).

Status:
Created by direct construction or transformation. Not estimated.
```

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.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')
```

See Also

```
idgrey | idnlgrey | idss
```

Related Examples

Identifying State-Space Models with Separate Process and Measurement Noise Descriptions

General Model Structure

An identified linear model is used to simulate and predict system outputs for given input and noise signals. The input signals are measured while the noise signals are only known via their statistical mean and variance. The general form of the state-space model, often associated with Kalman filtering, is an example of such a model, and is defined as:

$$x(t + 1) = A(\theta)x(t) + B(\theta)u(t) + w(t)$$
$$y(t) = C(\theta)x(t) + D(\theta)u(t) + v(t),$$

$$(13\text{-}1)$$

where, at time t:

- $x(t)$ is the vector of model states.
- $u(t)$ is the measured input data.
- $y(t)$ is the measured output data.
- $w(t)$ is the process noise.
- $v(t)$ is the measurement noise.

The noise disturbances are independent random variables with zero mean and covariances:

$$E[w(t)w^\mathsf{T}(t)] = R_1(\theta)$$

$$E[v(t)v^\mathsf{T}(t)] = R_2(\theta)$$

$$E[w(t)v^\mathsf{T}(t)] = R_{12}(\theta)$$

The vector θ parameterizes the model, including the coefficients of the system matrices and the noise covariances. However, all elements of the model are not necessarily free. If you have physical insight into the states of the system and sources of noise, the model can have a specific structure with few parameters in the vector θ.

Innovations Form and One-Step Ahead Predictor

For a given value of θ, you want to predict the best estimates of $x(t)$ and $y(t)$ in the presence of any disturbances. The required predictor model equations are derived from the Kalman filtering technique:

$$\widehat{x}(t + 1, \theta) = A(\theta)x(t) + B(\theta)u(t) + K(\theta)\big[y(t) - C(\theta)\widehat{x}(t, \theta) - D(\theta)u(t)\big]$$
$$\widehat{y}(t, \theta) = C(\theta)\widehat{x}(t) + D(\theta)u(t),$$

$$(13\text{-}2)$$

where $\widehat{x}(t, \theta)$ is the predicted value of the state vector $x(t)$ at time instant t, and $\widehat{y}(t, \theta)$ is the predicted value of output $y(t)$. The variables $u(t)$ and $y(t)$ in the above equation represent the measured input and output values at time t. The *Kalman Gain* matrix, $K(\theta)$, is derived from the system matrices and noise covariances as follows:

$$K(\theta) = [A(\theta)\Gamma(\theta)C^\mathsf{T}(\theta) + R_{12}(\theta)][C(\theta)\Gamma(\theta)C^\mathsf{T}(\theta) + R_2(\theta)]^{-1},$$

where $\Gamma(\theta)$ is the covariance of the state estimate error:

$$\Gamma(\theta) = \bar{E}\left[[x(t) - \widehat{x}(t,\theta)][x(t) - \widehat{x}(t,\theta)]^{\mathsf{T}}\right].$$

$\Gamma(\theta)$ is the solution of an algebraic Riccati equation. For more information, see `dare` and [1]

Denoting the output prediction error as $e(t) = y(t) - \widehat{y}(t,\theta)$, you can write the general state-space model in a simpler form:

$$x(t+1,\theta) = A(\theta)x(t) + B(\theta)u(t) + K(\theta)e(t)$$
$$y(t) = C(\theta)x(t) + D(\theta)u(t) + e(t).$$

(13-3)

This simpler representation is the innovations form of the state-space model, and has only one unique disturbance source, $e(t)$. This form corresponds to choosing $R_2=I$, $R_{12}=K$, and $R_1=KK^T$ for the general model structure. System Identification Toolbox software uses the innovations form as its primary representation of state-space models.

Both the general and innovations form of the model lead to the same predictor model as shown in "Equation 13-2". Use the `predict` command to compute the predicted model response and to generate this predictor system.

Model Identification

The identification task is to use input and output measurement data to determine the parameterization vector, θ. The approach to take depends on the amount of prior information available regarding the system and the noise disturbances.

Black Box Identification

When only input-output data measurements are available, and you have no knowledge of the noise structure, you can only estimate the model in the innovations form. To do so, we use the one-step ahead prediction error minimization approach (PEM) to compute the best output predictor. For this approach, the matrix K is parameterized independently of the other system matrices, and no prior information about the system states or output covariances is considered for estimation. The estimated model can be cast back into the general model structure in many nonunique ways, one of which is to assume $R_2=I$, $R_{12}=K$, and $R_1=KK^T$. The innovations form is a system representation of the predictor in which $e(t)$ does not necessarily represent the actual measurement noise.

Estimate state-space models in the innovations form using the `n4sid`, `ssest`, and `ssregest` commands. The system matrices A, B, C, D, and K are parameterized independently and the identification minimizes the weighted norm of the prediction error, $e(t)$. For more information, see "Estimating State-Space Models Using ssest, ssregest and n4sid" on page 7-25 and the estimation examples in `ssest`.

Note In this case, the estimation algorithm chooses the model states arbitrarily. As a result, it is difficult to imagine physically meaningful descriptions of the states and the sources for the disturbances affecting them.

Structured Identification

In some situations, in addition to the input-output data, you know something about the state and measurement disturbances. To make the notion of state disturbances meaningful, it is necessary that

the states be well-defined, such as the positions and velocities in a mechanical lumped-mass system. Well-defined states and known noise sources result in a *structured* state-space model, which you can then parameterize using the general model structure of "Equation 13-1".

To identify such models, use a grey-box modeling approach, which lets you use any prior knowledge regarding the system parameters and noise covariances. For example, you may know that only the first element of R_1 is nonzero, or that all the off-diagonal terms of R_2 are zero. When using grey-box modeling, provide initial guess values for the parameterization vector, θ. If the model states are physically meaningful, it should be possible to determine initial estimates for the parameters in θ.

To estimate a grey-box model with parameterized disturbances:

- Create a MATLAB function, called the ODE file, that:

  - Computes the parameterized state-space matrices, *A*, *B*, *C*, and *D*, using the parameter vector θ, which is supplied as an input argument.

  - Computes the noise covariance matrices R_1, R_2, and R_{12}. Each of these matrices can be completely or partially unknown. Any unknown matrix elements are defined in terms of parameters in θ.

  - Uses the system matrices *A* and *C*, and the noise covariances with the `kalman` command to find the Kalman gain matrix, *K*.

    ```
    [~,K] = kalman(ss(A,eye(nx),C,zeros(ny,nx),Ts),R1,R2,R12);
    ```

    Here, `nx` is the number of model states, `ny` is the number of model outputs, and `Ts` is the sample time. The `kalman` command requires Control System Toolbox software.

  - Returns *A*, *B*, *C*, *D*, and *K* as output arguments.
- Create an `idgrey` model that uses the ODE function and an initial guess value for the parameter vector, θ.
- Configure any estimation options using the `greyestOptions` command.
- Estimate θ using `greyest` command.

For an example of using parameterized disturbances with grey-box modeling, see "Estimate Discrete-Time Grey-Box Model with Parameterized Disturbance" on page 13-12.

Summary

Use the innovations form if all you have is measured input-output data. It is worthwhile to use the general form only if you can define a system parameterization with meaningful states, and you have nontrivial knowledge about the noise covariances. In this case, use grey-box estimation to identify the state-space model.

Both the general form and the innovations form lead to the same predictor. So, if your end goal is to deploy the model for predicting future outputs or to perform simulations, it is more convenient to use the innovations form of the model.

References

[1] Ljung, L. "State-Space Models." Section 4.3 in *System Identification: Theory for the User*. 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1999, pp. 93–102.

See Also
`idgrey` | `greyest` | `predict` | `ssest` | `ssregest` | `n4sid`

Related Examples

- "Estimate Discrete-Time Grey-Box Model with Parameterized Disturbance" on page 13-12
- "Estimate Linear Grey-Box Models" on page 13-6

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 17-2.

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 4-29. 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 `FunctionType ='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 "Simulate Identified Model in Simulink" on page 20-5.

See Also
`idgrey` | `idnlgrey`

Related Examples
- "Estimate Linear Grey-Box Models" on page 13-6
- "Estimate Nonlinear Grey-Box Models" on page 13-25

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 y u
```

The matrix y contains the two outputs: y1 is the angular position of the motor shaft and y2 is the angular velocity. There are 400 data samples and the sample time is 0.1 seconds. The input is contained in the vector u. It is the input voltage to the motor.

```
z = iddata(y,u,0.1); % The IDDATA object
z.InputName = 'Voltage';
z.OutputName = {'Angle';'AngVel'};
plot(z(:,1,:))
```


Figure: Measurement Data: Voltage to Angle

```
plot(z(:,2,:))
```

Figure: Measurement Data: Voltage to Angular Velocity

Model Structure Selection

```
d/dt x = A x + B u + K e
    y   = C x + D u + e
```

We shall build a model of the dc-motor. The dynamics of the motor is well known. If we choose x1 as the angular position and x2 as the angular velocity it is easy to set up a state-space model of the following character neglecting disturbances: (see Example 4.1 in Ljung(1999):

```
           | 0      1  |        | 0   |
d/dt x =   |           | x  +   |     | u
           | 0    -th1 |        | th2 |

         | 1  0 |
    y =  |      | x
         | 0  1 |
```

The parameter `th1` is here the inverse time-constant of the motor and `th2` is such that `th2/th1` is the static gain from input to the angular velocity. (See Ljung(1987) for how `th1` and `th2` relate to the physical parameters of the motor). We shall estimate these two parameters from the observed data. The model structure (parameterized state space) described above can be represented in MATLAB® using IDSS and IDGREY models. These models let you perform estimation of parameters using experimental data.

Specification of a Nominal (Initial) Model

If we guess that th1=1 and th2 = 0.28 we obtain the nominal or initial model

```
A = [0 1; 0 -1]; % initial guess for A(2,2) is -1
B = [0; 0.28]; % initial guess for B(2) is 0.28
C = eye(2);
D = zeros(2,1);
```

and we package this into an IDSS model object:

```
ms = idss(A,B,C,D);
```

The model is characterized by its matrices, their values, which elements are free (to be estimated) and upper and lower limits of those:

```
ms.Structure.a
```

```
ans =

       Name: 'A'
      Value: [2x2 double]
    Minimum: [2x2 double]
    Maximum: [2x2 double]
       Free: [2x2 logical]
      Scale: [2x2 double]
       Info: [2x2 struct]


1x1 param.Continuous


ms.Structure.a.Value
ms.Structure.a.Free


ans =

     0     1
     0    -1


ans =

  2x2 logical array

   1   1
   1   1
```

Specification of Free (Independent) Parameters Using IDSS Models

So we should now mark that it is only A(2,2) and B(2,1) that are free parameters to be estimated.

```
ms.Structure.a.Free = [0 0; 0 1];
ms.Structure.b.Free = [0; 1];
ms.Structure.c.Free = 0; % scalar expansion used
```

```matlab
ms.Structure.d.Free = 0;
ms.Ts = 0;  % This defines the model to be continuous
```

The Initial Model

```matlab
ms % Initial model
```

```
ms =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
        x1   x2
   x1    0    1
   x2    0   -1

  B =
         u1
   x1     0
   x2  0.28

  C =
        x1   x2
   y1    1    0
   y2    0    1

  D =
         u1
   y1    0
   y2    0

  K =
        y1   y2
   x1    0    0
   x2    0    0

Parameterization:
   STRUCTURED form (some fixed coefficients in  A, B, C).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 2
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

Estimation of Free Parameters of the IDSS Model

The prediction error (maximum likelihood) estimate of the parameters is now computed by:

```matlab
dcmodel = ssest(z,ms,ssestOptions('Display','on'));
dcmodel
```

```
dcmodel =
  Continuous-time identified state-space model:
```

```
        dx/dt = A x(t) + B u(t) + K e(t)
         y(t) = C x(t) + D u(t) + e(t)

  A =
           x1        x2
   x1        0         1
   x2        0  -4.013

  B =
      Voltage
   x1       0
   x2   1.002

  C =
          x1  x2
   Angle    1   0
   AngVel   0   1

  D =
          Voltage
   Angle        0
   AngVel       0

  K =
       Angle   AngVel
   x1       0        0
   x2       0        0

Parameterization:
   STRUCTURED form (some fixed coefficients in  A, B, C).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 2
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on time domain data "z".
Fit to estimation data: [98.35;84.42]%
FPE: 0.001071, MSE: 0.1192
```

```
Estimation data: Time domain data z
Data has 2 outputs, 1 inputs and 400 samples.
Number of states: 2
```

ESTIMATION PROGRESS

```
Algorithm: Nonlinear least squares with automatically chosen line search method
-------------------------------------------------------------------------------
-------
                        Norm of      First-order    Improvement (%)
Iteration       Cost      step        optimality     Expected  Achieved   Bisections
-------------------------------------------------------------------------------
-------
      0      1.89282         -       3.09e+03        145          -          -
      1     0.139814      0.862      2.13e+03        145        92.6         0
      2    0.00639664      1.15      1.73e+03        158        95.4         0
      3     0.0011775      0.848          385        109        81.6         0
      4    0.00106111      0.227          7.9        10.1       9.88         0
      5    0.00106082     0.0126        0.158       0.027      0.0272        0
      6    0.00106082    0.000154      0.00106    4.03e-06    4.07e-06       0
-------------------------------------------------------------------------------
```

RESULT

```
Termination condition: Near (local) minimum, (norm(g) < tol)..
Number of iterations: 6, Number of function evaluations: 13

Status: Estimated using SSEST
Fit to estimation data: [98.35;84.42]%, FPE: 0.00107149
```

■ Stop   Close

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);
```

Simulated Response Comparison

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)
```

Pole-Zero Map

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);
```

The resulting model is

```
dcmodel2
```

```
dcmodel2 =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
            x1       x2
    x1       0   0.9975
    x2       0   -4.011

  B =
        Voltage
    x1        0
    x2    1.004
```

```
 C =
          x1  x2
   Angle    1   0
   AngVel   0   1

 D =
          Voltage
   Angle        0
   AngVel       0

 K =
        Angle  AngVel
   x1       0       0
   x2       0       0

Parameterization:
   STRUCTURED form (some fixed coefficients in  A, B, C).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 3
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PEM on time domain data "z".
Fit to estimation data: [98.35;84.42]%
FPE: 0.001077, MSE: 0.1192
```

We find that the estimated `A(1,2)` is close to 1. To compare the two model we use the `compare` command:

`compare(z,dcmodel,dcmodel2)`

Specification of Models with Coupled Parameters Using IDGREY Objects

Suppose that we accurately know the static gain of the dc-motor (from input voltage to angular velocity, e.g. from a previous step-response experiment. If the static gain is G, and the time constant of the motor is t, then the state-space model becomes

```
            |0     1|     |  0  |
d/dt x =    |       |x + |     | u
            |0   -1/t|     | G/t |


            |1    0|
    y    =  |      | x
            |0    1|
```

With G known, there is a dependence between the entries in the different matrices. In order to describe that, the earlier used way with "Free" parameters will not be sufficient. We thus have to write a MATLAB file which produces the A, B, C, and D, and optionally also the K and X0 matrices as outputs, for each given parameter vector as input. It also takes auxiliary arguments as inputs, so that the user can change certain things in the model structure, without having to edit the file. In this case we let the known static gain G be entered as such an argument. The file that has been written has the name 'motorDynamics.m'.

```
type motorDynamics
```

```
function [A,B,C,D,K,X0] = motorDynamics(par,ts,aux)
%MOTORDYNAMICS ODE file representing the dynamics of a motor.
```

```
%
%   [A,B,C,D,K,X0] = motorDynamics(Tau,Ts,G)
%   returns the State Space matrices of the DC-motor with
%   time-constant Tau (Tau = par) and known static gain G. The sample
%   time is Ts.
%
%   This file returns continuous-time representation if input argument Ts
%   is zero. If Ts>0, a discrete-time representation is returned. To make
%   the IDGREY model that uses this file aware of this flexibility, set the
%   value of Structure.FcnType property to 'cd'. This flexibility is useful
%   for conversion between continuous and discrete domains required for
%   estimation and simulation.
%
%   See also IDGREY, IDDEMO7.

%   L. Ljung
%   Copyright 1986-2015 The MathWorks, Inc.

t = par(1);
G = aux(1);

A = [0 1;0 -1/t];
B = [0;G/t];
C = eye(2);
D = [0;0];
K = zeros(2);
X0 = [0;0];
if ts>0 % Sample the model with sample time Ts
   s = expm([[A B]*ts; zeros(1,3)]);
   A = s(1:2,1:2);
   B = s(1:2,3);
end
```

We now create an IDGREY model object corresponding to this model structure: The assumed time constant will be

```
par_guess = 1;
```

We also give the value 0.25 to the auxiliary variable G (gain) and sample time.

```
aux = 0.25;
dcmm = idgrey('motorDynamics',par_guess,'cd',aux,0);
```

The time constant is now estimated by

```
dcmm = greyest(z,dcmm,greyestOptions('Display','on'));
```

We have thus now estimated the time constant of the motor directly. Its value is in good agreement with the previous estimate.

```
dcmm
```

```
dcmm =
  Continuous-time linear grey box model defined by "motorDynamics" function:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)
```

```
A =
          x1        x2
  x1        0         1
  x2        0   -4.006

B =
      Voltage
  x1        0
  x2    1.001

C =
          x1  x2
  Angle    1   0
  AngVel   0   1

D =
          Voltage
  Angle         0
  AngVel        0

K =
      Angle  AngVel
  x1      0       0
  x2      0       0

 Model parameters:
  Par1 = 0.2496

Parameterization:
   ODE Function: motorDynamics
   (parametrizes both continuous- and discrete-time equations)
   Disturbance component: parameterized by the ODE function
   Initial state: parameterized by the ODE function
   Number of free coefficients: 1
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using GREYEST on time domain data "z".
Fit to estimation data: [98.35;84.42]%
FPE: 0.00107, MSE: 0.1193
```

With this model we can now proceed to test various aspects as before. The syntax of all the commands is identical to the previous case. For example, we can compare the idgrey model with the other state-space model:

```
compare(z,dcmm,dcmodel)
```

Simulated Response Comparison

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:

$$a_{kl}(q) = 1 + a_1 q^{-1} + \ldots + a_{nakl} q^{-nakl} q$$

It is thus a polynomial in `1/q` of degree `nakl`.

Similarly B(q) is a ny | nu matrix, whose kj-element is:

$$b_{kj}(q) = b_0 q^{-nkk} + b_1 q^{-nkkj-1} + \ldots + b_{nbkj} q^{-nkkj-nbkj}$$

There is thus a delay of `nkkj` from input number `j` to output number `k`. The most common way to create those would be to use the ARX-command. The orders are specified as: `nn = [na nb nk]` with `na` being a `ny-by-ny` matrix whose `kj`-entry is `nakj`; `nb` and `nk` are defined similarly.

Let's test some ARX-models on the dc-data. First we could simply build a general second order model:

```
dcarx1 = arx(z,'na',[2,2;2,2],'nb',[2;2],'nk',[1;1])
```

```
dcarx1 =
Discrete-time ARX model:
  Model for output "Angle": A(z)y_1(t) = - A_i(z)y_i(t) + B(z)u(t) + e_1(t)
    A(z) = 1 - 0.5545 z^-1 - 0.4454 z^-2

    A_2(z) = -0.03548 z^-1 - 0.06405 z^-2

    B(z) = 0.004243 z^-1 + 0.006589 z^-2

  Model for output "AngVel": A(z)y_2(t) = - A_i(z)y_i(t) + B(z)u(t) + e_2(t)
    A(z) = 1 - 0.2005 z^-1 - 0.2924 z^-2

    A_1(z) = 0.01849 z^-1 - 0.01937 z^-2

    B(z) = 0.08642 z^-1 + 0.03877 z^-2

Sample time: 0.1 seconds

Parameterization:
   Polynomial orders:   na=[2 2;2 2]   nb=[2;2]   nk=[1;1]
   Number of free coefficients: 12
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using ARX on time domain data "z".
Fit to estimation data: [97.87;83.44]% (prediction focus)
FPE: 0.002157, MSE: 0.1398
```

The result, `dcarx1`, is stored as an IDPOLY model, and all previous commands apply. We could for example explicitly list the ARX-polynomials by:

```
dcarx1.a
```

```
ans =

  2x2 cell array

    {[1 -0.5545 -0.4454]}    {[0 -0.0355 -0.0640]}
    {[ 0 0.0185 -0.0194]}    {[1 -0.2005 -0.2924]}
```

as cell arrays where e.g. the {1,2} element of dcarx1.a is the polynomial A(1,2) described earlier, relating y2 to y1.

We could also test a structure, where we know that y1 is obtained by filtering y2 through a first order filter. (The angle is the integral of the angular velocity). We could then also postulate a first order dynamics from input to output number 2:

```
na = [1 1; 0 1];
nb = [0 ; 1];
nk = [1 ; 1];
dcarx2 = arx(z,[na nb nk])


dcarx2 =
Discrete-time ARX model:
  Model for output "Angle": A(z)y_1(t) = - A_i(z)y_i(t) + B(z)u(t) + e_1(t)
    A(z) = 1 - 0.9992 z^-1

    A_2(z) = -0.09595 z^-1

    B(z) = 0

  Model for output "AngVel": A(z)y_2(t) = B(z)u(t) + e_2(t)
    A(z) = 1 - 0.6254 z^-1

    B(z) = 0.08973 z^-1

Sample time: 0.1 seconds

Parameterization:
   Polynomial orders:   na=[1 1;0 1]   nb=[0;1]   nk=[1;1]
   Number of free coefficients: 4
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using ARX on time domain data "z".
Fit to estimation data: [97.52;81.46]% (prediction focus)
FPE: 0.003452, MSE: 0.177
```

To compare the different models obtained we use

```
compare(z,dcmodel,dcmm,dcarx2)
```

Finally, we could compare the bodeplots obtained from the input to output one for the different models by using `bode`: First output:

```
dcmm2 = idss(dcmm); % convert to IDSS for subreferencing
bode(dcmodel(1,1),'r',dcmm2(1,1),'b',dcarx2(1,1),'g')
```

Second output:

```
bode(dcmodel(2,1),'r',dcmm2(2,1),'b',dcarx2(2,1),'g')
```

The two first models are more or less in exact agreement. The ARX-models are not so good, due to the bias caused by the non-white equation error noise. (We had white measurement noise in the simulations).

Conclusions

Estimation of models with pre-selected structures can be performed using System Identification toolbox. In state-space form, parameters may be fixed to their known values or constrained to lie within a prescribed range. If relationship between parameters or other constraints need to be specified, IDGREY objects may be used. IDGREY models evaluate a user-specified MATLAB file for estimating state-space system parameters. Multi-variate ARX models offer another option for quickly estimating multi-output models with user-specified structure.

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" on page 13-60.

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', 'dcmotordata'));
```

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.

```
z.InputName = 'Voltage';
z.InputUnit =  'V';
z.OutputName = {'Angular position', 'Angular velocity'};
z.OutputUnit = {'rad', 'rad/s'};
z.Tstart = 0;
z.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 position).
figure('Name', [z.Name ': Voltage input -> Angular velocity output']);
plot(z(:, 2, 1));    % Plot second input-output pair (Voltage -> Angular velocity).
```

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" on page 13-45.

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 velocity.
      -(1/tau)*x(2)+(k/tau)*u(1)    ... % Angular acceleration.
     ];
```

The file must always be structured to return the following:

Output arguments:

- **dx** is the vector of state derivatives in continuous-time case, and state update values in the discrete-time case.

- **y** is the output equation

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.

```
nlgr = setinit(nlgr, 'Name', {'Angular position' 'Angular velocity'});
nlgr = setinit(nlgr, 'Unit', {'rad' 'rad/s'});
nlgr = setpar(nlgr, 'Name', {'Time-constant' 'Static gain'});
nlgr = setpar(nlgr, 'Unit', {'s' 'rad/(V*s)'});
```

You can also use `setinit` and `setpar` to assign values, minima, maxima, and estimation status for all initial states or parameters simultaneously.

5. View the initial model.

a. Get basic information about the model.

The DC-motor has 2 (initial) states and 2 model parameters.

```
size(nlgr)
```

```
Nonlinear grey-box model with 2 outputs, 1 inputs, 2 states and 2 parameters (2 free).
```

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 `help idnlgrey.InitialStates` and `help idnlgrey.Parameters` for more information.

```
nlgr.InitialStates(1)
nlgr.Parameters(2)


ans =

  struct with fields:

       Name: 'Angular position'
       Unit: 'rad'
      Value: 0
    Minimum: -Inf
    Maximum: Inf
      Fixed: 1


ans =

  struct with fields:

       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 =

  2x1 cell array

    {[1]}
    {[1]}


ans =

  2x1 cell array

    {[-Inf]}
```

```
   {[-Inf]}
```

d. Obtain basic information about the object:

```
nlgr
```

```
nlgr =

Continuous-time nonlinear grey-box model defined by 'dcmotor_m' (MATLAB file):

   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(s), 2 state(s), 2 output(s), and 2 free parameter(s) (out of 2).

Name: DC-motor

Status:
Created by direct construction or transformation. Not estimated.
```

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: {}
   SimulationOptions: [1x1 struct]
        TimeVariable: 't'
       NoiseVariance: [2x2 double]
           InputName: {'Voltage'}
           InputUnit: {'V'}
          InputGroup: [1x1 struct]
          OutputName: {2x1 cell}
          OutputUnit: {2x1 cell}
         OutputGroup: [1x1 struct]
               Notes: [0x1 string]
            UserData: []
                Name: 'DC-motor'
                  Ts: 0
            TimeUnit: 'seconds'
              Report: [1x1 idresults.nlgreyest]
```

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). The simulation options are specified using the "SimulationOptions" model property.

1. Set the absolute and relative error tolerances to small values (`1e-6` and `1e-5`, respectively).

```
nlgr.SimulationOptions.AbsTol = 1e-6;
nlgr.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.

```
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 `nlgreyest`, which is a prediction error minimization method for nonlinear grey box models. The estimation options, such as the choice of estimation progress display, are specified using the "nlgreyestOptions" option set.

```
nlgr = setinit(nlgr, 'Fixed', {false false}); % Estimate the initial states.
opt = nlgreyestOptions('Display', 'on');
nlgr = nlgreyest(z, nlgr, opt);
```

```
Nonlinear Grey Box Model Estimation
Data has 2 outputs, 1 inputs and 400 samples.
ODE Function: dcmotor_m
Number of parameters: 2
```

ESTIMATION PROGRESS

```
Algorithm: Trust-Region Reflective Newton
-----------------------------------------------------------------
                            Norm of      First-order
Iteration      Cost          step        optimality
-----------------------------------------------------------------
    0         1.39351          -              -
    1         0.111682        0.827        1.89e+03
    2         0.0596697       0.106          44.4
    3         0.0593534       0.0124         0.661
    4         0.0593527       0.000834       0.0804
    5         0.0593527       5.65e-05       0.00014
-----------------------------------------------------------------
```

RESULT

```
Termination condition: Change in cost was less than the specified tolerance..
Number of iterations: 5, Number of function evaluations: 6

Status: Estimated using NLGREYEST
Fit to estimation data: [98.34;84.47]%, FPE: 0.0010963
```

■ Stop   Close

Performance Evaluation of the Estimated DC-Motor Model

1. Review the information about the estimation process.

This information is stored in the `Report` 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.Report
fprintf('\n\nThe search termination condition:\n')
nlgr.Report.Termination
```

```
ans =

        Status: 'Estimated using NLGREYEST'
        Method: 'Solver: ode45; Search: lsqnonlin'
           Fit: [1x1 struct]
    Parameters: [1x1 struct]
   OptionsUsed: [1x1 idoptions.nlgreyest]
     RandState: []
      DataUsed: [1x1 struct]
   Termination: [1x1 struct]


The search termination condition:

ans =

  struct with fields:

              WhyStop: 'Change in cost was less than the specified tolerance.'
           Iterations: 5
    FirstOrderOptimality: 1.4013e-04
             FcnCount: 6
            Algorithm: 'trust-region-reflective'
```

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.

```
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]);
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).

```
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. Use the `resid` command to view the correlations among the residuals. The first column of plots shows the autocorrelations of the residuals for the two outputs. The second column shows the cross-correlation of these residuals with the input "Voltage". The correlations are within acceptable bounds (blue region).

```
figure('Name',[nlgr.Name ': residuals of estimated model']);
resid(z,nlgr);
```

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.

```
getcov(nlgr)
nlgr.NoiseVariance
```

```
ans =

    1.0e-04 *

    0.1573    0.0021
    0.0021    0.0008


ans =

    0.0010   -0.0000
```

```
     -0.0000      0.0110
```

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);
```

```
nlgr =

Continuous-time nonlinear grey-box model defined by 'dcmotor_m' (MATLAB file):

   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(s), 2 state(s), 2 output(s), and 2 free parameter(s) (out of 2).

 Inputs:
    u(1)  Voltage(t) [V]
 States:                             Initial value
    x(1)  Angular position(t) [rad]     xinit@exp1   0.0302675   (estimated) in [-Inf, Inf]
    x(2)  Angular velocity(t) [rad/s]   xinit@exp1   -0.133777   (estimated) in [-Inf, Inf]
 Outputs:
    y(1)  Angular position(t) [rad]
    y(2)  Angular velocity(t) [rad/s]
 Parameters:                         Value Standard Deviation
    p1   Time-constant [s]              0.243649      0.00396671   (estimated) in [-Inf, Inf]
    p2   Static gain [rad/(V*s)]        0.249644      0.000284486  (estimated) in [-Inf, Inf]

Name: DC-motor

Status:
Termination condition: Change in cost was less than the specified tolerance..
Number of iterations: 5, Number of function evaluations: 6

Estimated using Solver: ode45; Search: lsqnonlin on time domain data "DC-motor".
Fit to estimation data: [98.34;84.47]%
FPE: 0.001096, MSE: 0.1187
More information in model's "Report" property.

<a href="matlab:if exist('nlgr','var'), if isa(nlgr,'idnlgrey'), disp(' '); disp(get(nlgr)), else
```

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.

Modeling a Vehicle Dynamics System

This example shows nonlinear grey-box modeling of vehicle dynamics. Many new vehicle features (like Electronic Stability Programs (ESP), indirect Tire Pressure Monitoring Systems (TPMS), road-tire friction monitoring systems, and so forth) rely on models of the underlying vehicle dynamics. The so-called bicycle vehicle model is a rather simple model structure that is frequently being used in the vehicle dynamics literature. In this example we will start-off with this model structure and try to estimate the longitudinal and the lateral stiffness of a tire. The actual modeling work was originally carried out by Erik Narby in his MSc work at NIRA Dynamics AB, Sweden.

Vehicle Dynamics Modeling

The following figure illustrates the vehicle modeling situation to be considered.
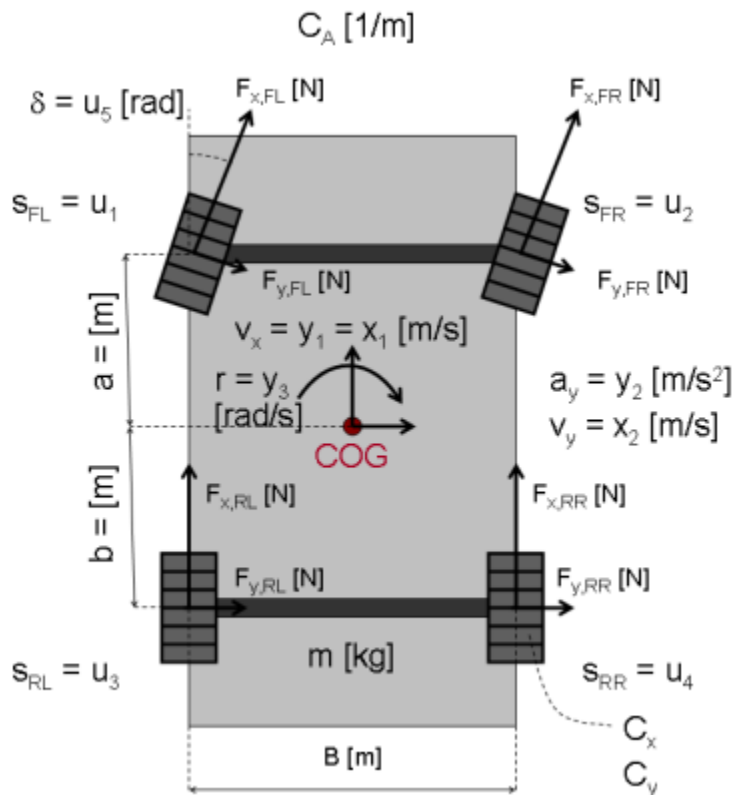
Figure 1: Schematic view of a vehicle dynamics system.

By the use of Newton's law of motion and some basic geometric relationships, the longitudinal velocity v_x(t), the lateral velocity v_y(t) and the yaw rate r(t) measured around the Center Of Gravity (COG) of the vehicle can be described by the following three differential equations:

```
d
-- v_x(t) = v_y(t)*r(t) + 1/m*(  (F_x,FL(t)+F_x,FR(t))*cos(delta(t))
dt                             - (F_y,FL(t)+F_y,FR(t))*sin(delta(t))
                               + F_x,RL(t)+F_x,RR(t)
                               - C_A*v_x(t)^2)
d
```

```
-- v_y(t) = -v_x(t)*r(t) + 1/m*(  (F_x,FL(t)+F_x,FR(t))*sin(delta(t))
dt                                + (F_y,FL(t)+F_y,FR(t))*cos(delta(t))
                                  + F_y,RL(t)+F_y,RR(t))

d
-- r(t)   = 1/J*(  a*(  (F_x,FL(t)+F_x,FR(t))*sin(delta(t))
dt                    + (F_y,FL(t)+F_y,FR(t))*cos(delta(t)))
                 - b*(F_y,RL(t)+F_y,RR(t)))
```

where subscript x is used to denote that a force F acts in the longitudinal direction and y that it acts in the lateral direction. The abbreviations FL, FR, RL and RR label the tires: Front Left, Front Right, Rear Left and Rear Right, respectively. The first equation describing the longitudinal acceleration also contains an air resistance term that is assumed to be a quadratic function of the longitudinal vehicle velocity v_x(t). In addition, delta(t) (an input) is the steering angle, J a moment of inertia, and a and b the distances from the center of gravity to the front and rear axles, respectively.

Let us assume that the tire forces can be modeled through the following linear approximations:

```
F_x,i(t) = C_x*s_i(t)
F_y,i(t) = C_y*alpha_i(t)      for i = {FL, FR, RL, RR}
```

where C_x and C_y are the longitudinal and lateral tire stiffness, respectively. Here we have assumed that these stiffness parameters are the same for all 4 tires. s_i(t) is the so-called (longitudinal) slip of tire i and alpha_i(t) a tire slip angle. For a front-wheel driven vehicle (as considered here), the slips s_FL(t) and s_FR(t) are derived from the individual wheel speeds (measured) by assuming that the rear wheels do not show any slip (i.e., s_RL(t) = s_RR(t) = 0). Hence the slips are inputs to our model structure. For the front wheels, the tire slip angles alpha_Fj(t) can be approximated by (when v_x(t) > 0)

```
alpha_Fj(t) = delta(t) - arctan((v_y(t) + a*r(t))/v_x(t))
            ~ delta(t) - (v_y(t) + a*r(t))/v_x(t)          for j = {L, R}
```

For the rear wheels, the tire slip angles alpha_Rj(t) are similarly derived and computed as

```
alpha_Rj(t) = - arctan((v_y(t) - b*r(t))/v_x(t))
            ~ - (v_y(t) - b*r(t))/v_x(t)                   for j = {L, R}
```

With J = 1/((0.5*(a+b))^2*m) we can next set up a state-space structure describing the vehicle dynamics. Introduce the states:

```
x1(t) = v_x(t)      Longitudinal velocity [m/s].
x2(t) = v_y(t)      Lateral velocity [m/s].
x3(t) = r(t)        Yaw rate [rad/s].
```

the five measured or derived input signals

```
u1(t) = s_FL(t)     Slip of Front Left tire [ratio].
u2(t) = s_FR(t)     Slip of Front Right tire [ratio].
u3(t) = s_RL(t)     Slip of Rear Left tire [ratio].
u4(t) = s_RR(t)     Slip of Rear Right tire [ratio].
u5(t) = delta(t)    Steering angle [rad].
```

and the model parameters:

```
m                   Mass of the vehicle [kg].
a                   Distance from front axle to COG [m].
b                   Distance from rear axle to COG [m].
Cx                  Longitudinal tire stiffness [N].
```

```
   Cy                      Lateral tire stiffness [N/rad].
   CA                      Air resistance coefficient [1/m].
```

The outputs of the system are the longitudinal vehicle velocity y1(t) = x1(t), the lateral vehicle acceleration (measured by an accelerometer):

```
y2(t) = a_y(t) = 1/m*(  (F_x,FL(t) + F_x,FR(t))*sin(delta(t))
                      + (F_y,FL(t) + F_y,FR(t))*cos(delta(t))
                      + F_y,RL(t) + F_y,RR(t))
```

and the yaw rate y3(t) = r(t) (measured by a gyro).

Put together, we arrive at the following state-space model structure:

```
   d
   -- x1(t) = x2(t)*x3(t) + 1/m*(  Cx*(u1(t)+u2(t))*cos(u5(t))
   dt                            - 2*Cy*(u5(t)-(x2(t)+a*x3(t))/x1(t))*sin(u5(t))
                                 + Cx*(u3(t)+u4(t))
                                 - CA*x1(t)^2)

   d
   -- x2(t) = -x1(t)*x3(t) + 1/m*(  Cx*(u1(t)+u2(t))*sin(u5(t))
   dt                             + 2*Cy*(u5(t)-(x2(t)+a*x3(t))/x1(t))*cos(u5(t))
                                  + 2*Cy*(b*x3(t)-x2(t))/x1(t))

   d
   -- x3(t) = 1/((0.5*(a+b))^2)*m)*(  a*(  Cx*(u1(t)+u2(t)*sin(u5(t))
   dt                                    + 2*Cy*(u5(t) - (x2(t)+a*x3(t))/x1(t))*cos(u5(t)))
                                      - 2*b*Cy*(b*x3(t)-x2(t))/x1(t))

       y1(t) = x1(t)
       y2(t) = 1/m*(  Cx*(u1(t)+u2(t))*sin(u5(t))
                    + 2*Cy*(u5(t)-(x2(t)+a*x3(t))/x1(t))*cos(u5(t))
                    + 2*Cy*(b*x3(t)-x2(t))/x1(t))
       y3(t) = x3(t)
```

IDNLGREY Vehicle Model

As a basis for our vehicle identification experiments we first need to create an IDNLGREY model file describing these vehicle equations. Here we rely on C-MEX modeling and create a vehicle_c.c model file, in which NY is set to 3. The state and output update functions of vehicle_c.c, compute_dx and compute_y, are somewhat involved and includes several standard C-defined mathematical functions, like cos(.) and sin(.) as well as pow(.) for computing the power of its argument.

The state update function compute_dx returns dx (argument 1) and uses 3 input arguments: the state vector x, the input vector u, and the six scalar parameters encoded in p (t and auxvar of the template C-MEX model file have been removed here):

```
  /* State equations. */
  void compute_dx(double *dx, double *x, double *u, double **p)
  {
      /* Retrieve model parameters. */
      double *m, *a, *b, *Cx, *Cy, *CA;
      m  = p[0];   /* Vehicle mass.                    */
      a  = p[1];   /* Distance from front axle to COG. */
      b  = p[2];   /* Distance from rear axle to COG.  */
      Cx = p[3];   /* Longitudinal tire stiffness.     */
      Cy = p[4];   /* Lateral tire stiffness.          */
      CA = p[5];   /* Air resistance coefficient.      */
```

```
    /* x[0]: Longitudinal vehicle velocity. */
    /* x[1]: Lateral vehicle velocity. */
    /* x[2]: Yaw rate. */
    dx[0] = x[1]*x[2]+1/m[0]*(Cx[0]*(u[0]+u[1])*cos(u[4])
            -2*Cy[0]*(u[4]-(x[1]+a[0]*x[2])/x[0])*sin(u[4])
            +Cx[0]*(u[2]+u[3])-CA[0]*pow(x[0],2));
    dx[1] = -x[0]*x[2]+1/m[0]*(Cx[0]*(u[0]+u[1])*sin(u[4])
            +2*Cy[0]*(u[4]-(x[1]+a[0]*x[2])/x[0])*cos(u[4])
            +2*Cy[0]*(b[0]*x[2]-x[1])/x[0]);
    dx[2] = 1/(pow(((a[0]+b[0])/2),2)*m[0])
            *(a[0]*(Cx[0]*(u[0]+u[1])*sin(u[4])
            +2*Cy[0]*(u[4]-(x[1]+a[0]*x[2])/x[0])*cos(u[4]))
            -2*b[0]*Cy[0]*(b[0]*x[2]-x[1])/x[0]);
}
```

The output update function compute_y returns y (argument 1) and uses 3 input arguments: the state vector x, the input vector u, and five of the six parameters (the air resistance CA is not needed) encoded in p:

```
/* Output equations. */
void compute_y(double *y, double *x, double *u, double **p)
{
    /* Retrieve model parameters. */
    double *m  = p[0];   /* Vehicle mass.                   */
    double *a  = p[1];   /* Distance from front axle to COG. */
    double *b  = p[2];   /* Distance from rear axle to COG.  */
    double *Cx = p[3];   /* Longitudinal tire stiffness.    */
    double *Cy = p[4];   /* Lateral tire stiffness.         */

    /* y[0]: Longitudinal vehicle velocity. */
    /* y[1]: Lateral vehicle acceleration. */
    /* y[2]: Yaw rate. */
    y[0] = x[0];
    y[1] = 1/m[0]*(Cx[0]*(u[0]+u[1])*sin(u[4])
            +2*Cy[0]*(u[4]-(x[1]+a[0]*x[2])/x[0])*cos(u[4])
            +2*Cy[0]*(b[0]*x[2]-x[1])/x[0]);
    y[2] = x[2];
}
```

Having a proper model structure file, the next step is to create an IDNLGREY object reflecting the modeling situation. For ease of bookkeeping, we also specify the names and units of the inputs and outputs.

```
FileName      = 'vehicle_c';                      % File describing the model structure.
Order         = [3 5 3];                          % Model orders [ny nx nu].
Parameters    = [1700; 1.5; 1.5; 1.5e5; 4e4; 0.5]; % Initial parameters.
InitialStates = [1; 0; 0];                        % Initial value of initial states.
Ts            = 0;                                % Time-continuous system.
nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts, ...
             'Name', 'Bicycle vehicle model', 'TimeUnit', 's');
     nlgr.InputName =  {'Slip on front left tire';          ...   % u(1).
                        'Slip on front right tire';         ...   % u(2).
                        'Slip on rear left tire';           ...   % u(3).
                        'Slip on rear right tire';          ...   % u(4).
                        'Steering angle'};                  ...   % u(5).
       nlgr.InputUnit =  {'ratio'; 'ratio'; 'ratio'; 'ratio'; 'rad'};

       nlgr.OutputName = {'Long. velocity';  ...   % y(1); Longitudinal vehicle velocity
```

```
                                  'Lat. accel.';    ...      % y(2); Lateral vehicle acceleration
                                  'Yaw rate'};                                ...   % y(3).
                nlgr.OutputUnit = {'m/s'; 'm/s^2'; 'rad/s'};
```

The names and the units of the (initial) states and the model parameters are specified via SETINIT.
We also use this command to specify that the first initial state (the longitudinal velocity) ought to be
strictly positive for the model to be valid and to specify that all model parameters should be strictly
positive. These constraints will subsequently be honored when performing initial state and/or model
parameter estimation.

```
nlgr = setinit(nlgr, 'Name', {'Longitudinal vehicle velocity'       ... % x(1).
                              'Lateral vehicle velocity'       ... % x(2).
                              'Yaw rate'});                    ... % x(3).
nlgr = setinit(nlgr, 'Unit', {'m/s'; 'm/s'; 'rad/s'});
nlgr.InitialStates(1).Minimum = eps(0);   % Longitudinal velocity > 0 for the model to be valid.
nlgr = setpar(nlgr, 'Name', {'Vehicle mass';                        ... % m.
                             'Distance from front axle to COG';      ... % a
                             'Distance from rear axle to COG';       ... % b.
                             'Longitudinal tire stiffness';          ... % Cx.
                             'Lateral tire stiffness';               ... % Cy.
                             'Air resistance coefficient'});         ... % CA.
nlgr = setpar(nlgr, 'Unit', {'kg'; 'm'; 'm'; 'N'; 'N/rad'; '1/m'});
nlgr = setpar(nlgr, 'Minimum', num2cell(eps(0)*ones(6, 1)));   % All parameters > 0!
```

Four of the six parameters of this model structure can readily be obtained through the data sheet of
the vehicle in question:

```
  m  = 1700 kg
  a  = 1.5 m
  b  = 1.5 m
  CA = 0.5 or 0.7 1/m (see below)
```

Hence we will not estimate these parameters:

```
nlgr.Parameters(1).Fixed = true;
nlgr.Parameters(2).Fixed = true;
nlgr.Parameters(3).Fixed = true;
nlgr.Parameters(6).Fixed = true;
```

With this, a textual summary of the entered IDNLGREY model structure is obtained through
PRESENT as follows.

```
present(nlgr);
```

```
nlgr =

Continuous-time nonlinear grey-box model defined by 'vehicle_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 5 input(s), 3 state(s), 3 output(s), and 2 free parameter(s) (out of 6).

 Inputs:
    u(1)  Slip on front left tire(t) [ratio]
    u(2)  Slip on front right tire(t) [ratio]
    u(3)  Slip on rear left tire(t) [ratio]
```

```
    u(4)  Slip on rear right tire(t) [ratio]
    u(5)  Steering angle(t) [rad]
  States:                                          Initial value
    x(1)  Longitudinal vehicle velocity(t) [m/s]   xinit@exp1   1   (fixed) in ]0, Inf]
    x(2)  Lateral vehicle velocity(t) [m/s]        xinit@exp1   0   (fixed) in [-Inf, Inf]
    x(3)  Yaw rate(t) [rad/s]                       xinit@exp1   0   (fixed) in [-Inf, Inf]
  Outputs:
    y(1)  Long. velocity(t) [m/s]
    y(2)  Lat. accel.(t) [m/s^2]
    y(3)  Yaw rate(t) [rad/s]
  Parameters:                                Value
    p1   Vehicle mass [kg]                    1700      (fixed) in ]0, Inf]
    p2   Distance from front axle to COG [m]    1.5      (fixed) in ]0, Inf]
    p3   Distance from rear axle to COG [m]     1.5      (fixed) in ]0, Inf]
    p4   Longitudinal tire stiffness [N]      150000    (estimated) in ]0, Inf]
    p5   Lateral tire stiffness [N/rad]        40000    (estimated) in ]0, Inf]
    p6   Air resistance coefficient [1/m]        0.5      (fixed) in ]0, Inf]

Name: Bicycle vehicle model

Status:
Created by direct construction or transformation. Not estimated.
More information in model's "Report" property.

<a href="matlab:if exist('nlgr','var'), if isa(nlgr,'idnlgrey'), disp(' '); disp(get(nlgr)), else
```

Input-Output Data

At this point, we load the available input-output data. This file contains data from three different experiments:

```
  A. Simulated data with high stiffness tires [y1 u1].
  B. Simulated data with low stiffness tires  [y2 u2].
  C. Measured data from a Volvo V70           [y3 u3].
```

In all cases, the sample time Ts = 0.1 seconds.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'vehicledata'));
```

A. System Identification Using Simulated High Tire Stiffness Data

In our first vehicle identification experiment we consider simulated high tire stiffness data. A copy of the model structure nlgr and an IDDATA object z1 reflecting this particular modeling situation is first created. The 5 input signals are stored in u1 and the 3 output signals in y1. The slip inputs (generated from the wheel speed signals) for the front wheels were chosen to be sinusoidal with a constant offset; the yaw rate was also sinusoidal but with a different amplitude and frequency. In reality, this is a somewhat artificial situation, because one rarely excites the vehicle so much in the lateral direction.

```
nlgr1 = nlgr;
nlgr1.Name = 'Bicycle vehicle model with high tire stiffness';
z1 = iddata(y1, u1, 0.1, 'Name', 'Simulated high tire stiffness vehicle data');
z1.InputName = nlgr1.InputName;
z1.InputUnit = nlgr1.InputUnit;
z1.OutputName = nlgr1.OutputName;
z1.OutputUnit = nlgr1.OutputUnit;
z1.Tstart = 0;
z1.TimeUnit = 's';
```

The inputs and outputs are shown in two plot figures.

```
h_gcf = gcf;
set(h_gcf,'DefaultLegendLocation','southeast');
h_gcf.Position = [100 100 795 634];
for i = 1:z1.Nu
    subplot(z1.Nu, 1, i);
    plot(z1.SamplingInstants, z1.InputData(:,i));
    title(['Input #' num2str(i) ': ' z1.InputName{i}]);
    xlabel('');
    axis tight;
end
xlabel([z1.Domain ' (' z1.TimeUnit ')']);
```
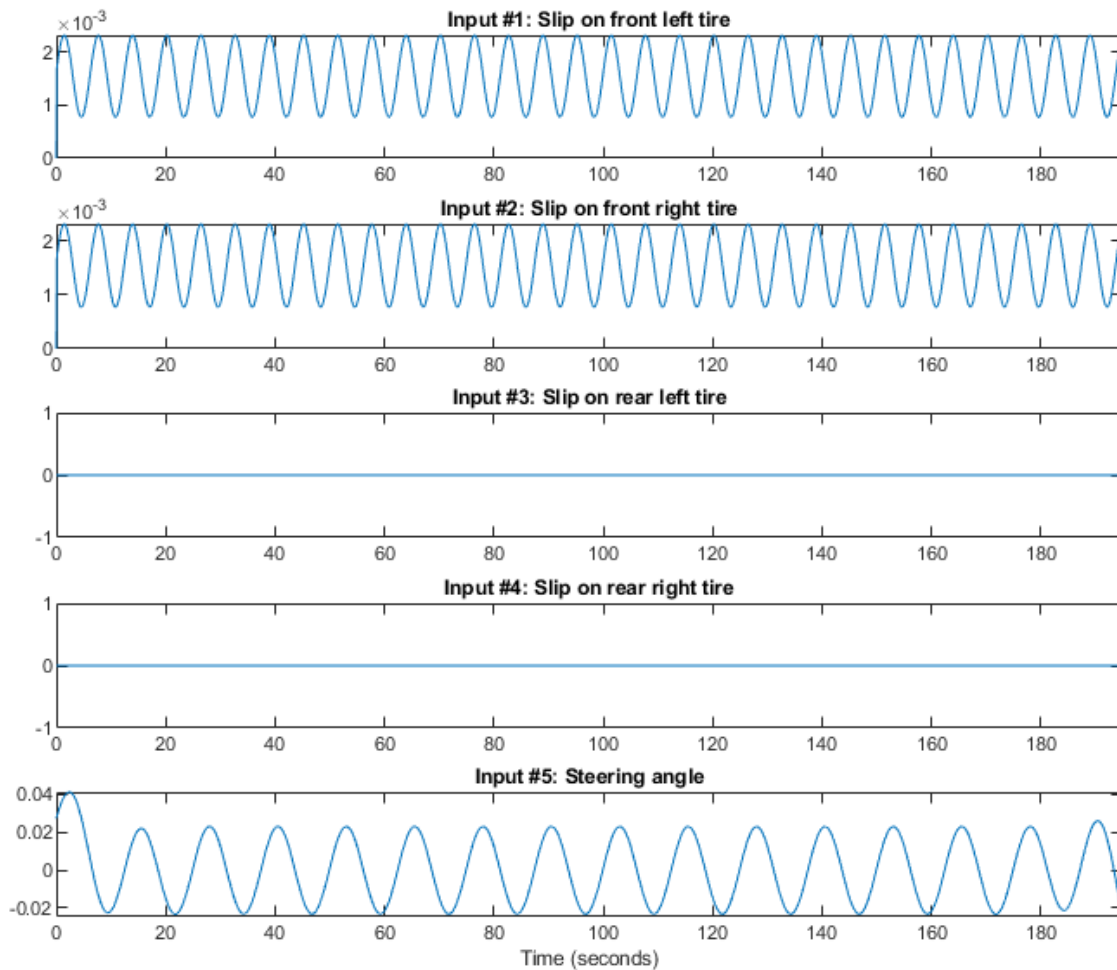

Figure 2: Inputs to a vehicle system with high tire stiffness.

```
for i = 1:z1.Ny
    subplot(z1.Ny, 1, i);
    plot(z1.SamplingInstants, z1.OutputData(:,i));
```

```
    title(['Output #' num2str(i) ': ' z1.OutputName{i}]);
    xlabel('');
    axis tight;
end
xlabel([z1.Domain ' (' z1.TimeUnit ')']);
```
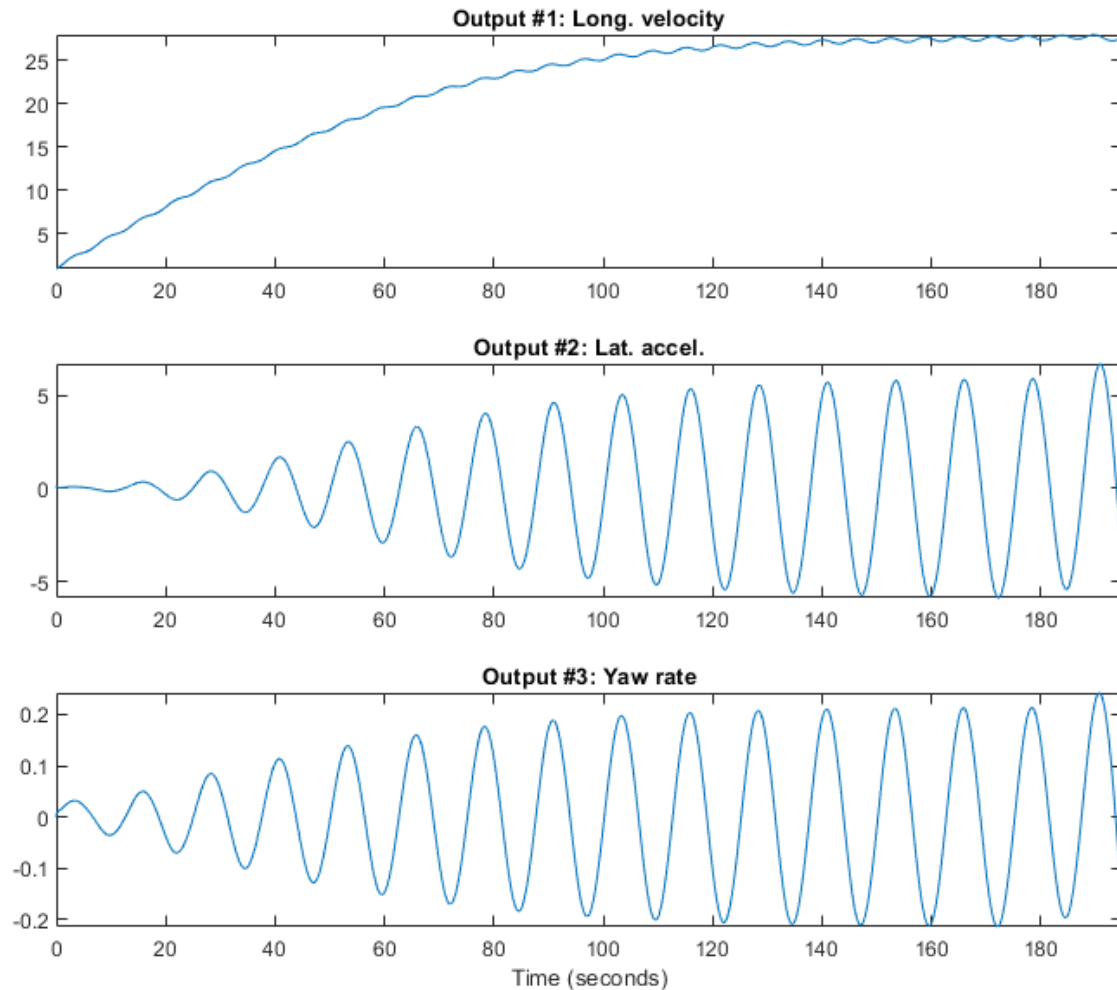

Figure 3: Outputs from a vehicle system with high tire stiffness.

The next step is to investigate the performance of the initial model and for this we perform a simulation. Notice that the initial state has been fixed to a non-zero value as the first state (the longitudinal vehicle velocity) is used as denominator in the model structure. A comparison between the true and the simulated outputs (with the initial model) is shown in a plot window.

```
clf
compare(z1, nlgr1, [], compareOptions('InitialCondition', 'model'));
```
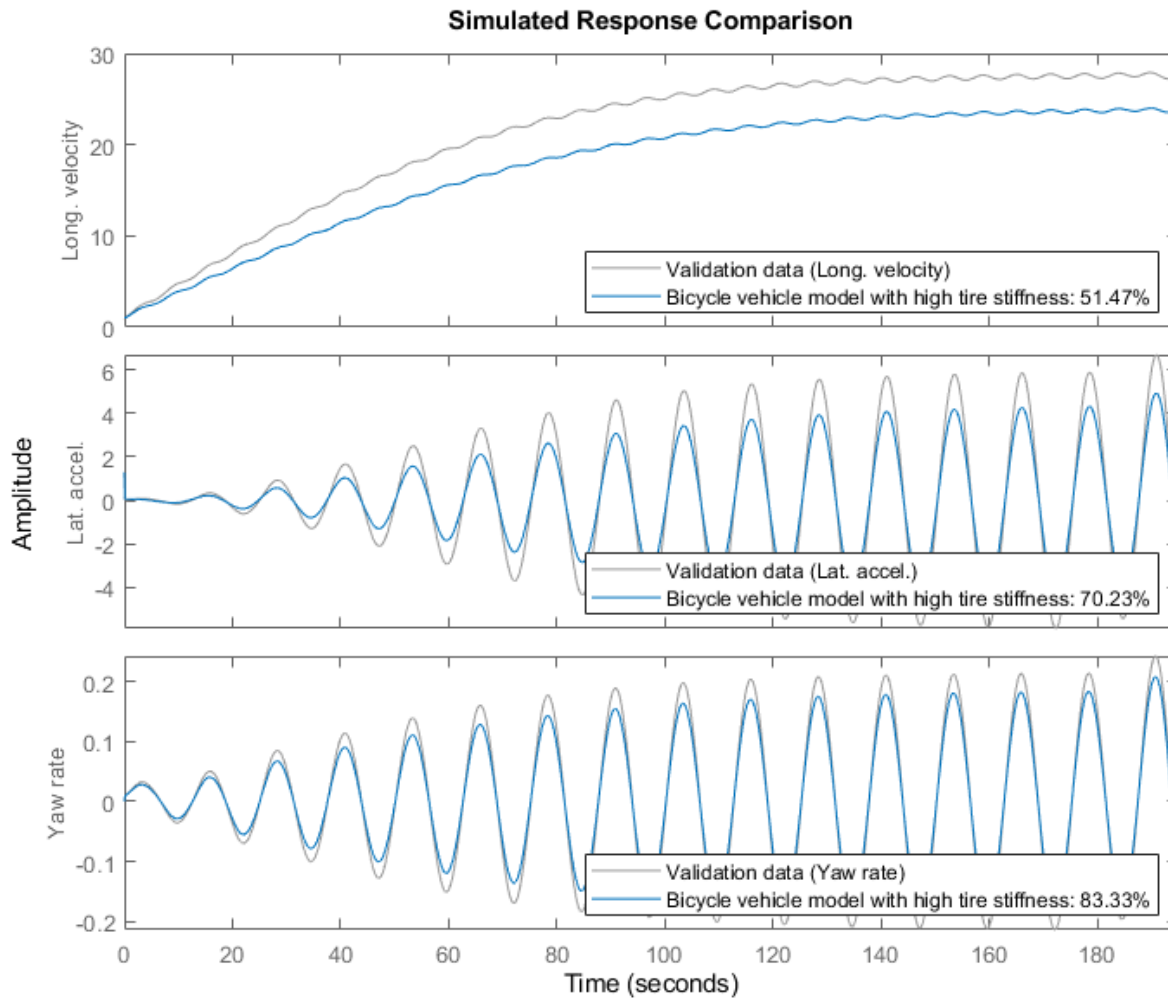
Figure 4: Comparison between true outputs and the simulated outputs of the initial vehicle model with high tire stiffness.

In order to improve the model fit, the two tire stiffness parameters Cx and Cy are next estimated, and a new simulation with the estimated model is carried out.

```
nlgr1 = nlgreyest(z1, nlgr1);
```

A comparison between the true and the simulated outputs (with the estimated model) is shown in a plot window.

```
compare(z1, nlgr1, [], compareOptions('InitialCondition', 'model'));
```
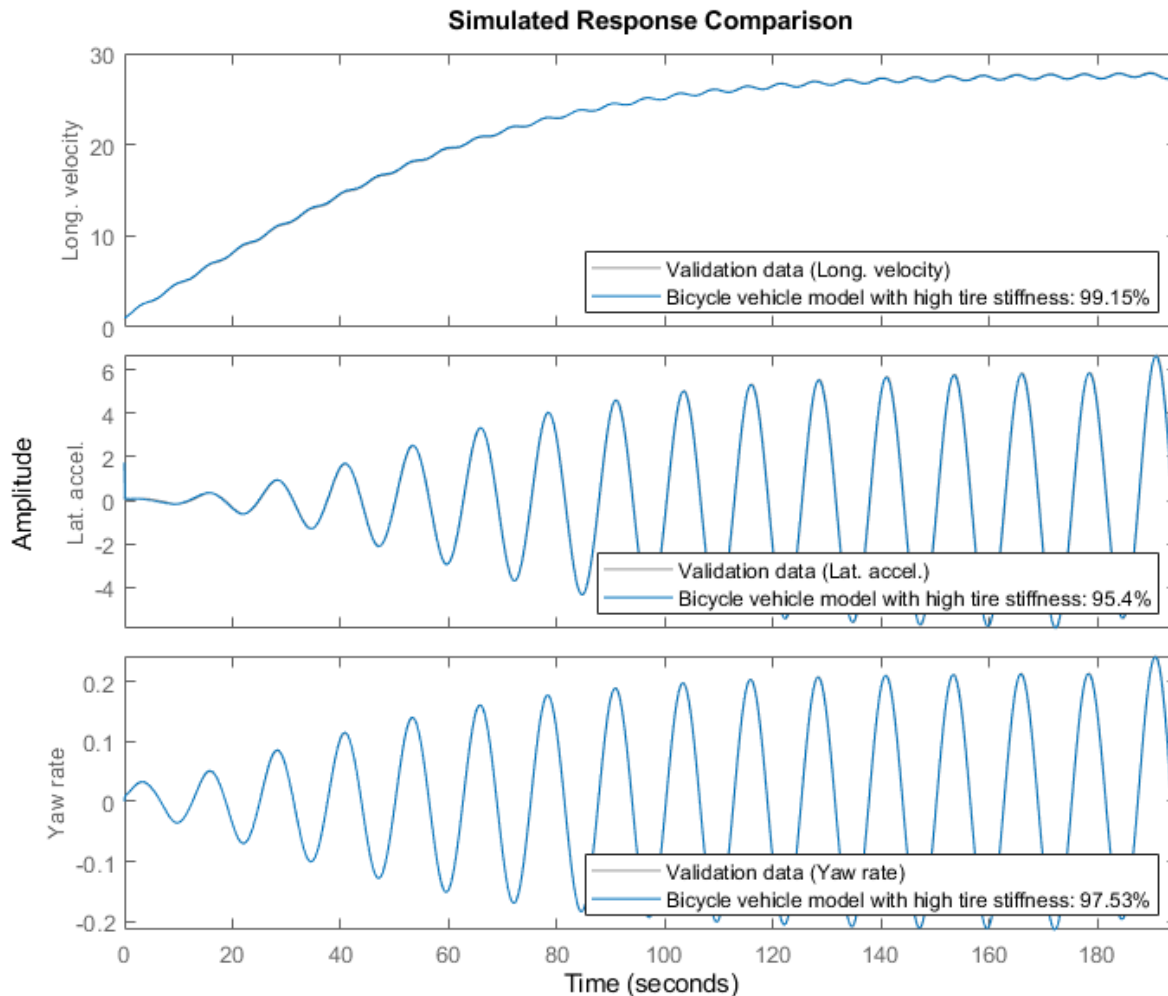
Figure 5: Comparison between true outputs and the simulated outputs of the estimated vehicle model with high tire stiffness.

The simulation performance of the estimated model is quite good. The estimated stiffness parameters are also close to the ones used in Simulink® to generate the true output data:

```
disp('                         True      Estimated');
fprintf('Longitudinal stiffness: %6.0f    %6.0f\n', 2e5, nlgr1.Parameters(4).Value);
fprintf('Lateral stiffness     : %6.0f    %6.0f\n', 5e4, nlgr1.Parameters(5).Value);
```

```
                        True      Estimated
Longitudinal stiffness: 200000     198517
Lateral stiffness     :  50000      53752
```

B. System Identification Using Simulated Low Tire Stiffness Data

In the second experiment we repeat the modeling from the first experiment, but now with simulated low tire stiffness data.

```
nlgr2 = nlgr;
nlgr2.Name = 'Bicycle vehicle model with low tire stiffness';
z2 = iddata(y2, u2, 0.1, 'Name', 'Simulated low tire stiffness vehicle data');
z2.InputName = nlgr2.InputName;
z2.InputUnit = nlgr2.InputUnit;
z2.OutputName = nlgr2.OutputName;
z2.OutputUnit = nlgr2.OutputUnit;
z2.Tstart = 0;
z2.TimeUnit =  's';
```

The inputs and outputs are shown in two plot figures.

```
clf
for i = 1:z2.Nu
    subplot(z2.Nu, 1, i);
    plot(z2.SamplingInstants, z2.InputData(:,i));
    title(['Input #' num2str(i) ': ' z2.InputName{i}]);
    xlabel('');
    axis tight;
end
xlabel([z2.Domain ' (' z2.TimeUnit ')']);
```
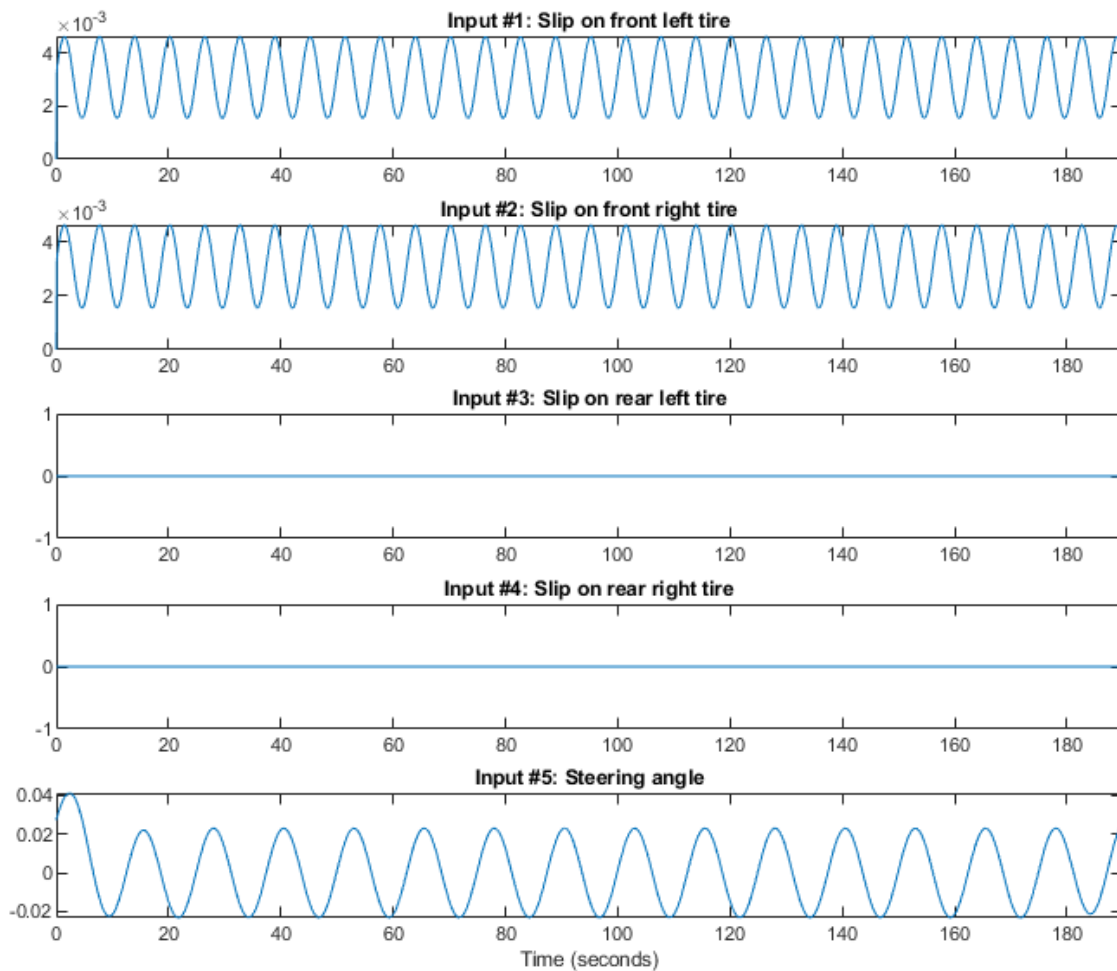
Figure 6: Inputs to a vehicle system with low tire stiffness.

```
clf
for i = 1:z2.Ny
   subplot(z2.Ny, 1, i);
   plot(z2.SamplingInstants, z2.OutputData(:,i));
   title(['Output #' num2str(i) ': ' z2.OutputName{i}]);
   xlabel('');
   axis tight;
end
xlabel([z2.Domain ' (' z2.TimeUnit ')']);
```
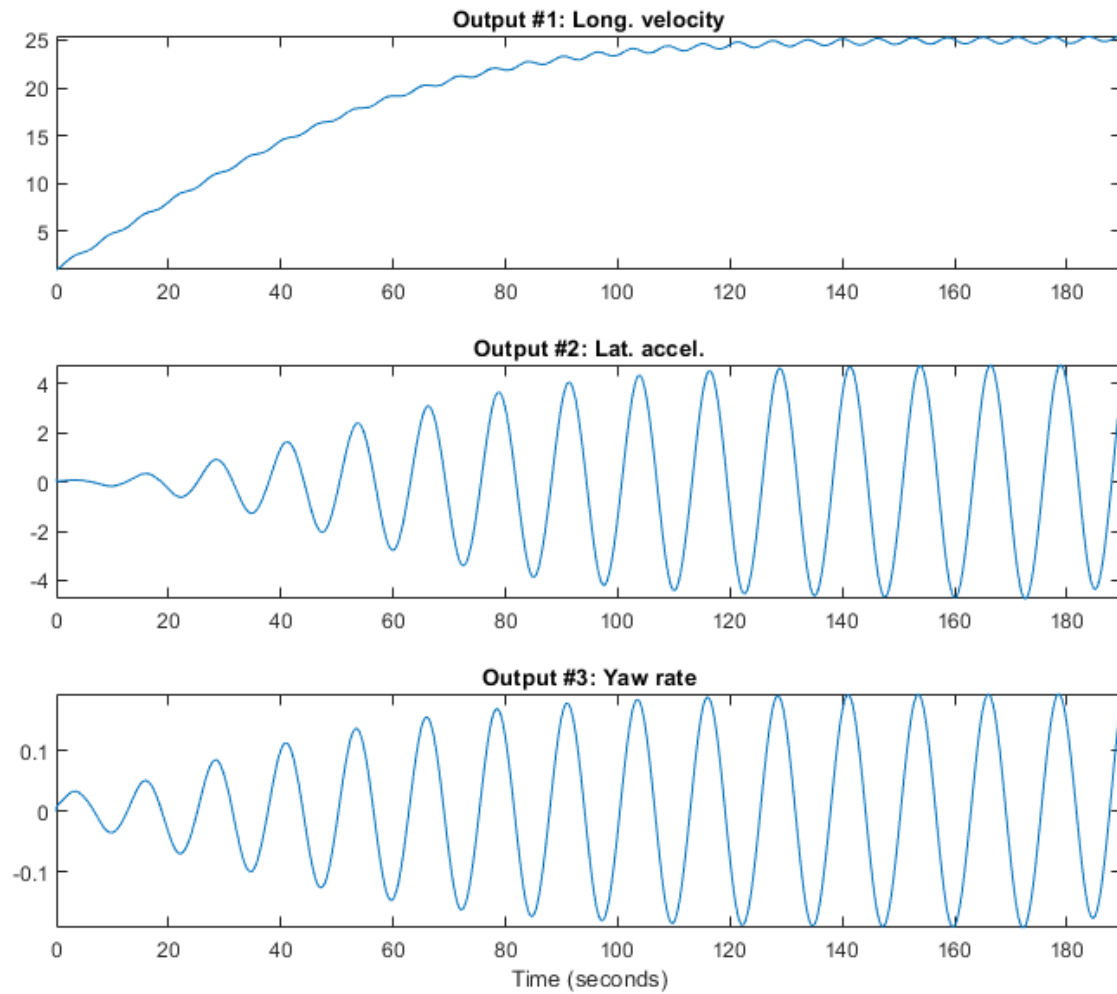
Figure 7: Outputs from a vehicle system with low tire stiffness.

Next we investigate the performance of the initial model (which has the same parameters as the initial high tire stiffness model). A comparison between the true and the simulated outputs (with the initial model) is shown in a plot window.

```
clf
compare(z2, nlgr2, [], compareOptions('InitialCondition', 'model'));
```
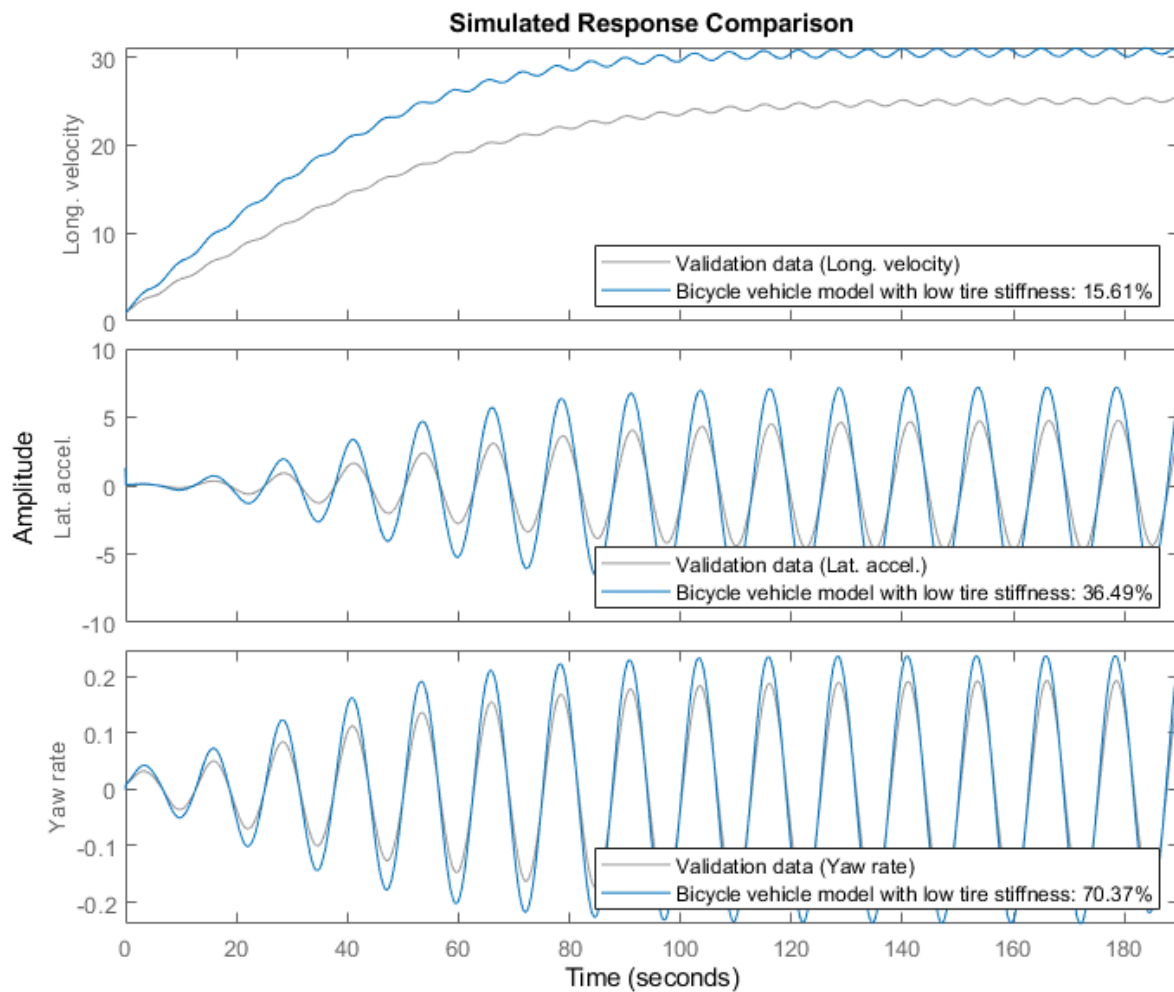
Figure 8: Comparison between true outputs and the simulated outputs of the initial vehicle model with low tire stiffness.

The two stiffness parameters are next estimated.

```
nlgr2 = nlgreyest(z2, nlgr2);
```

A comparison between the true and the simulated outputs (with the estimated model) is shown in a plot window.

```
compare(z2, nlgr2, [], compareOptions('InitialCondition', 'model'));
```
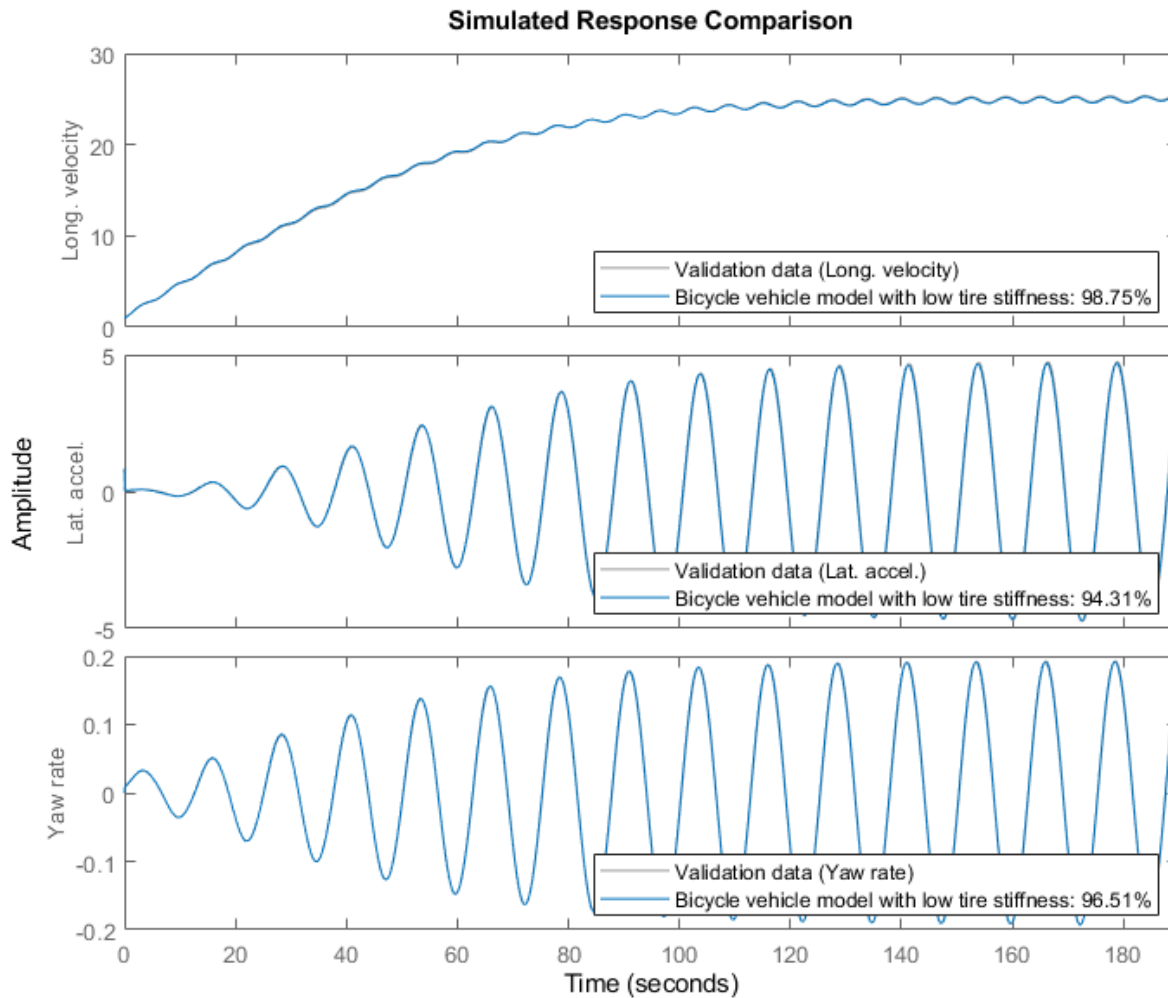
Figure 9: Comparison between true outputs and the simulated outputs of the estimated vehicle model with low tire stiffness.

The simulation performance of the estimated model is again really good. Even with the same parameter starting point as was used in the high tire stiffness case, the estimated stiffness parameters are also here close to the ones used in Simulink to generate the true output data:

```
disp('                          True      Estimated');
fprintf('Longitudinal stiffness: %6.0f    %6.0f\n', 1e5, nlgr2.Parameters(4).Value);
fprintf('Lateral stiffness     : %6.0f    %6.0f\n', 2.5e4, nlgr2.Parameters(5).Value);
```

```
                          True      Estimated
Longitudinal stiffness: 100000      99573
Lateral stiffness     :  25000      26117
```

C. System Identification Using Measured Volvo V70 Data

In the final experiment we consider data collected in a Volvo V70. As above, we make a copy of the generic vehicle model object nlgr and create a new IDDATA object containing the measured data.

Here we have also increased the air resistance coefficient from 0.50 to 0.70 to better reflect the Volvo V70 situation.

```
nlgr3 = nlgr;
nlgr3.Name = 'Volvo V70 vehicle model';
nlgr3.Parameters(6).Value = 0.70;    % Use another initial CA for the Volvo data.
z3 = iddata(y3, u3, 0.1, 'Name', 'Volvo V70 data');
z3.InputName = nlgr3.InputName;
z3.InputUnit = nlgr3.InputUnit;
z3.OutputName = nlgr3.OutputName;
z3.OutputUnit = nlgr3.OutputUnit;
z3.Tstart = 0;
z3.TimeUnit = 's';
```

The inputs and outputs are shown in two plot figures. As can be seen, the measured data is rather noisy.

```
clf
for i = 1:z3.Nu
   subplot(z3.Nu, 1, i);
   plot(z3.SamplingInstants, z3.InputData(:,i));
   title(['Input #' num2str(i) ': ' z3.InputName{i}]);
   xlabel('');
   axis tight;
end
xlabel([z3.Domain ' (' z3.TimeUnit ')']);
```
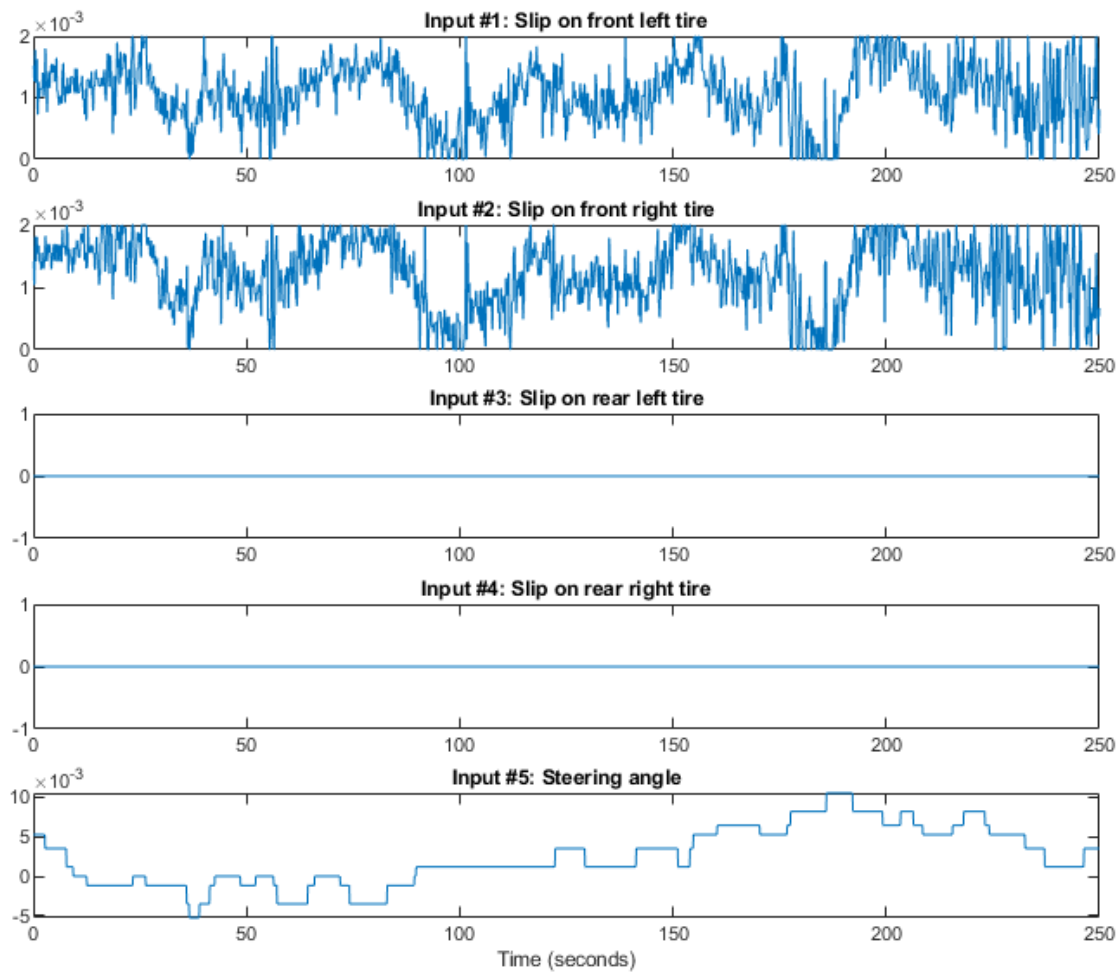
Figure 10: Measured inputs from a Volvo V70 vehicle.

```
clf
for i = 1:z3.Ny
    subplot(z3.Ny, 1, i);
    plot(z3.SamplingInstants, z3.OutputData(:,i));
    title(['Output #' num2str(i) ': ' z3.OutputName{i}]);
    xlabel('');
    axis tight;
end
xlabel([z3.Domain ' (' z3.TimeUnit ')']);
```
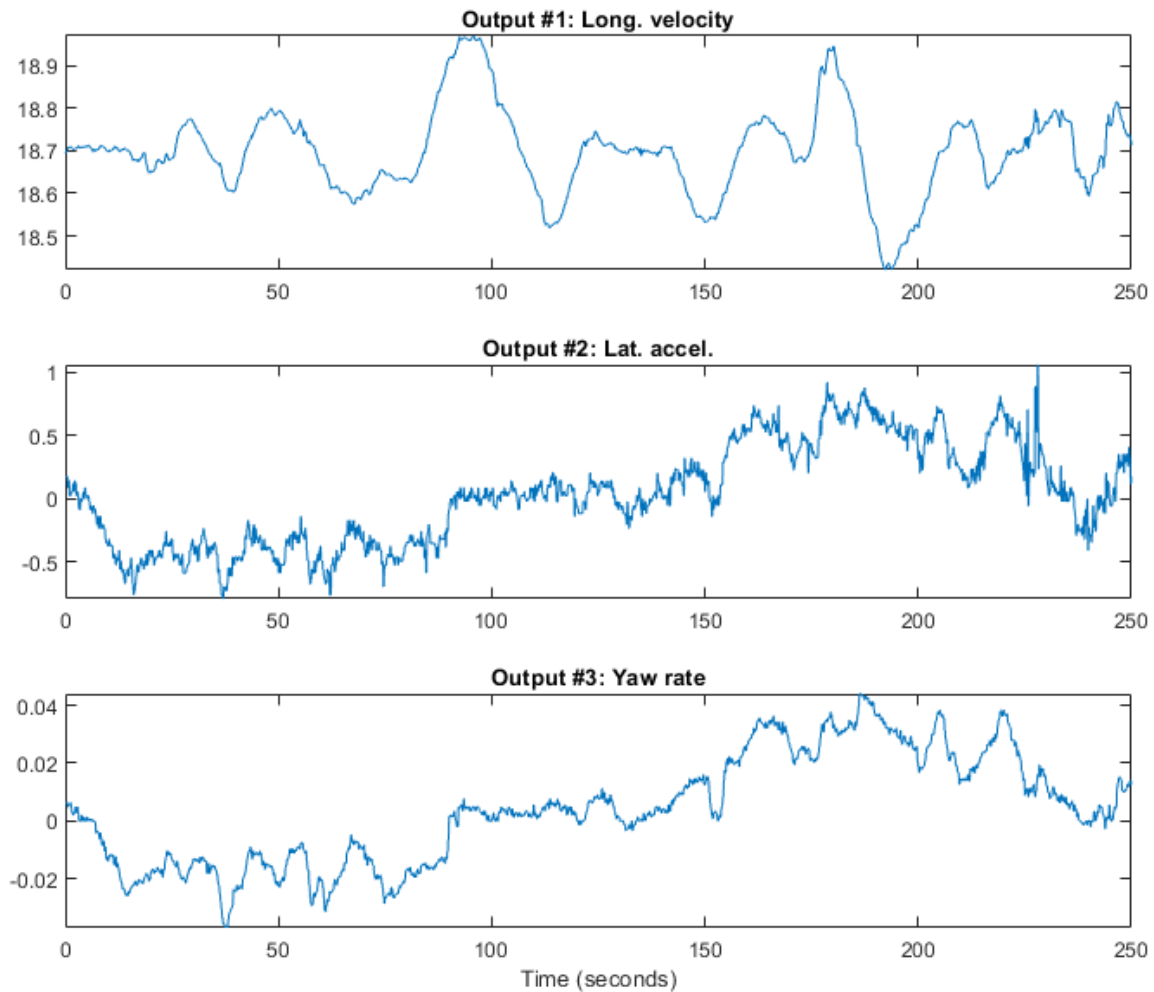
Figure 11: Measured outputs from a Volvo V70 vehicle.

Next we investigate the performance of the initial model with the initial states being estimated. A comparison between the true and the simulated outputs (with the initial model) is shown in a plot window.

```
nlgr3 = setinit(nlgr3, 'Value', {18.7; 0; 0});    % Initial value of initial states.
clf
compare(z3, nlgr3);
```
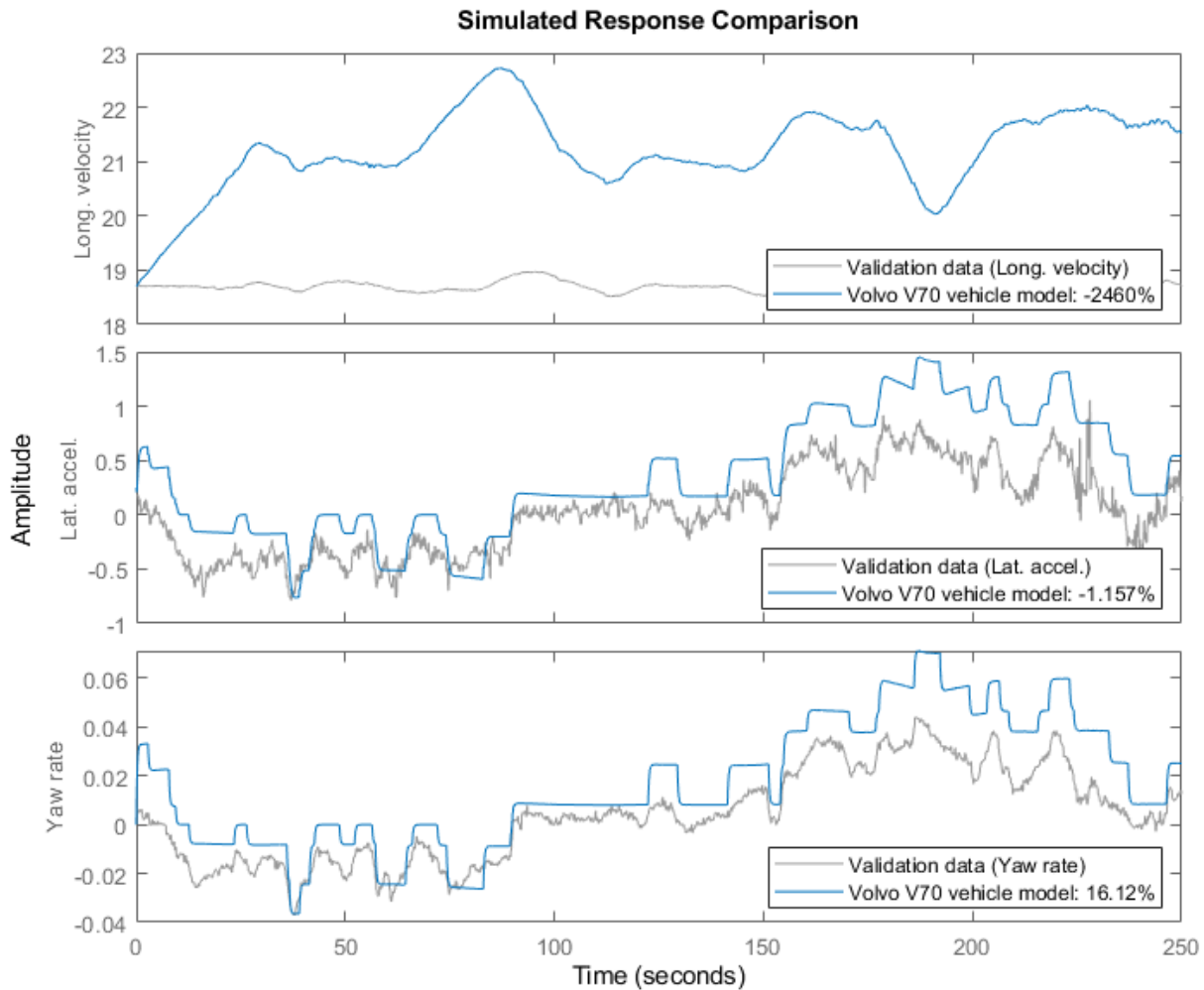
Figure 12: Comparison between measured outputs and the simulated outputs of the initial Volvo V70 vehicle model.

The tire stiffness parameters Cx and Cy are next estimated, in this case using the Levenberg-Marquardt search method, whereupon a new simulation with the estimated model is performed. In addition, we here estimate the initial value of the longitudinal velocity, whereas the initial values of the lateral velocity and the yaw rate are kept fixed.

```
nlgr3 = setinit(nlgr3, 'Fixed', {false; true; true});
nlgr3 = nlgreyest(z3, nlgr3, nlgreyestOptions('SearchMethod', 'lm'));
```

A comparison between the true and the simulated outputs (with the estimated model) is shown in a plot window.

```
compare(z3, nlgr3);
```
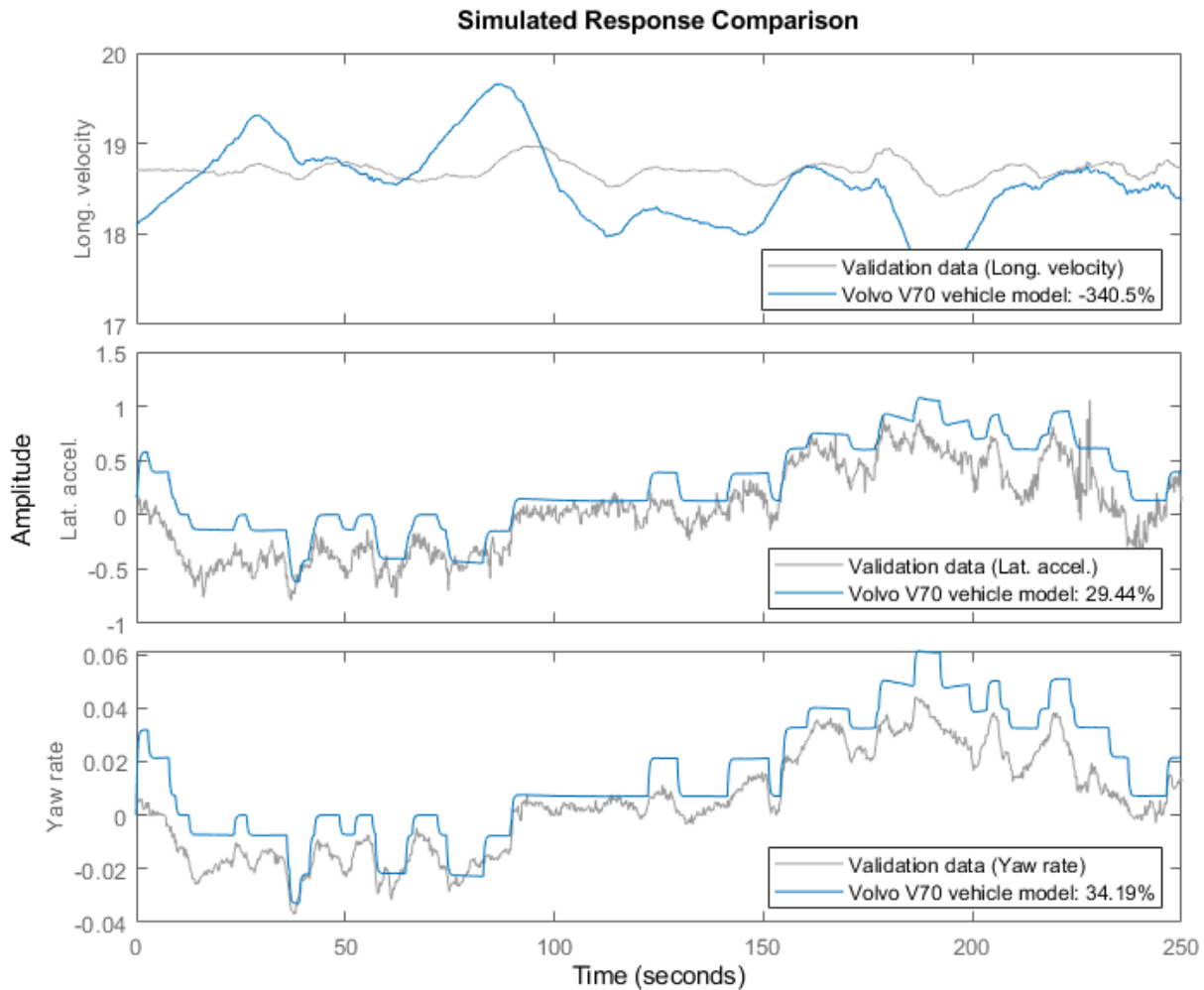
Figure 13: Comparison between measured outputs and the simulated outputs of the first estimated Volvo V70 vehicle model.

The estimated stiffness parameters of the final Volvo V70 model are reasonable, yet it is here unknown what their real values are.

```
disp('                    Estimated');
fprintf('Longitudinal stiffness: %6.0f\n', nlgr3.Parameters(4).Value);
fprintf('Lateral stiffness    : %6.0f\n', nlgr3.Parameters(5).Value);
```

```
                    Estimated
Longitudinal stiffness: 108873
Lateral stiffness    :  29964
```

Further information about the estimated Volvo V70 vehicle model is obtained through PRESENT. It is here interesting to note that the uncertainty related to the estimated lateral tire stiffness is quite high (and significantly higher than for the longitudinal tire stiffness). This uncertainty originates partly from that the lateral acceleration is varied so little during the test drive.

```
present(nlgr3);


nlgr3 =

Continuous-time nonlinear grey-box model defined by 'vehicle_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 5 input(s), 3 state(s), 3 output(s), and 2 free parameter(s) (out of 6).

 Inputs:
    u(1)  Slip on front left tire(t) [ratio]
    u(2)  Slip on front right tire(t) [ratio]
    u(3)  Slip on rear left tire(t) [ratio]
    u(4)  Slip on rear right tire(t) [ratio]
    u(5)  Steering angle(t) [rad]
 States:                                       Initial value
    x(1)  Longitudinal vehicle velocity(t) [m/s]  xinit@exp1   17.6049   (estimated) in ]0, Inf]
    x(2)  Lateral vehicle velocity(t) [m/s]        xinit@exp1        0   (fixed) in [-Inf, Inf]
    x(3)  Yaw rate(t) [rad/s]                      xinit@exp1        0   (fixed) in [-Inf, Inf]
 Outputs:
    y(1)  Long. velocity(t) [m/s]
    y(2)  Lat. accel.(t) [m/s^2]
    y(3)  Yaw rate(t) [rad/s]
 Parameters:                                   ValueStandard Deviation
    p1   Vehicle mass [kg]                           1700          0   (fixed) in ]0, Inf]
    p2   Distance from front axle to COG [m]           1.5          0   (fixed) in ]0, Inf]
    p3   Distance from rear axle to COG [m]            1.5          0   (fixed) in ]0, Inf]
    p4   Longitudinal tire stiffness [N]            108873    26.8501   (estimated) in ]0, Inf]
    p5   Lateral tire stiffness [N/rad]            29963.5    217.877   (estimated) in ]0, Inf]
    p6   Air resistance coefficient [1/m]              0.7          0   (fixed) in ]0, Inf]

Name: Volvo V70 vehicle model

Status:
Termination condition: Maximum number of iterations reached..
Number of iterations: 20, Number of function evaluations: 41

Estimated using Solver: ode45; Search: lm on time domain data "Volvo V70 data".
Fit to estimation data: [-374.2;29.74;34.46]%
FPE: 2.362e-07, MSE: 0.3106
More information in model's "Report" property.

<a href="matlab:if exist('nlgr3','var'), if isa(nlgr3,'idnlgrey'), disp(' '); disp(get(nlgr3)), e
```

Concluding Remarks

Estimating the tire stiffness parameters is in practice a rather intricate problem. First, the approximations introduced in the model structure above are valid for a rather narrow operation region, and data during high accelerations, braking, etc., cannot be used. The stiffness also varies with the environmental condition, e.g., the surrounding temperature, the temperature in the tires and the road surface conditions, which are not accounted for in the used model structure. Secondly, the estimation of the stiffness parameters relies heavily on the driving style. When mostly going straight

ahead as in the third identification experiment, it becomes hard to estimate the stiffness parameters (especially the lateral one), or put another way, the parameter uncertainties become rather high.

Modeling an Aerodynamic Body

This example shows the grey-box modeling of a large and complex nonlinear system. The purpose is to show the ability to use the IDNLGREY model to estimate a large number of parameters (16) in a system having many inputs (10) and outputs (5). The system is an aerodynamic body. We create a model that predicts the acceleration and velocity of the body using the measurements of its velocities (translational and angular) and various angles related to its control surfaces.

Input-Output Data

We load the measured velocities, angles and dynamic pressure from a data file named aerodata.mat:

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'aerodata'));
```

This file contains a uniformly sampled data set with 501 samples in the variables u and y. The sample time is 0.02 seconds. This data set was generated from another more elaborate model of the aerodynamic body.

Next, we create an IDDATA object to represent and store the data:

```
z = iddata(y, u, 0.02, 'Name', 'Data');
z.InputName = {'Aileron angle' 'Elevator angle'        ...
               'Rudder angle' 'Dynamic pressure'       ...
               'Velocity'                              ...
               'Measured angular velocity around x-axis' ...
               'Measured angular velocity around y-axis' ...
               'Measured angular velocity around z-axis' ...
               'Measured angle of attack'               ...
               'Measured angle of sideslip'};
z.InputUnit = {'rad' 'rad' 'rad' 'kg/(m*s^2)' 'm/s'     ...
   'rad/s' 'rad/s' 'rad/s' 'rad' 'rad'};
z.OutputName = {'V(x)'         ... % Angular velocity around x-axis
                'V(y)'         ... % Angular velocity around y-axis
                'V(z)'         ... % Angular velocity around z-axis
                'Accel.(y)'    ... % Acceleration in y-direction
                'Accel.(z)'    ... % Acceleration in z-direction
                };
z.OutputUnit = {'rad/s' 'rad/s' 'rad/s' 'm/s^2' 'm/s^2'};
z.Tstart =  0;
z.TimeUnit = 's';
```

View the input data:

```
figure('Name', [z.Name ': input data'],...
   'DefaultAxesTitleFontSizeMultiplier',1,...
   'DefaultAxesTitleFontWeight','normal',...
   'Position',[50, 50, 850, 620]);
for i = 1:z.Nu
   subplot(z.Nu/2, 2, i);
   plot(z.SamplingInstants, z.InputData(:,i));
   title(['Input #' num2str(i) ': ' z.InputName{i}],'FontWeight','normal');
   xlabel('');
   axis tight;
   if (i > z.Nu-2)
       xlabel([z.Domain ' (' z.TimeUnit ')']);
```

```
    end
end
```

Figure 1: Input signals.

View the output data:

```
figure('Name', [z.Name ': output data']);
h_gcf = gcf;
Pos = h_gcf.Position;
h_gcf.Position = [Pos(1), Pos(2)-Pos(4)/2, Pos(3), Pos(4)*1.5];
for i = 1:z.Ny
   subplot(z.Ny, 1, i);
   plot(z.SamplingInstants, z.OutputData(:,i));
   title(['Output #' num2str(i) ': ' z.OutputName{i}]);
   xlabel('');
   axis tight;
end
xlabel([z.Domain ' (' z.TimeUnit ')']);
```

Figure 2: Output signals.

At a first glance, it might look strange to have measured variants of some of the outputs in the input vector. However, the model used for generating the data contains several integrators, which often result in an unstable simulation behavior. To avoid this, the measurements of some of the output signals are fed back via a nonlinear observer. These are the input 6 to 8 in the dataset z. Thus, this is a system operating in closed loop and the goal of the modeling exercise is to predict "future" values of those outputs using measurements of current and past behavior.

Modeling the System

To model the dynamics of interest, we use an IDNLGREY model object to represent a state space structure of the system. A reasonable structure is obtained by using Newton's basic force and momentum laws (balance equations). To completely describe the model structure, basic aerodynamic relations (constitutive relations) are also used.

The C MEX-file, aero_c.c, describes the system using the state and output equations and initial conditions, as described next. Omitting the derivation details for the equations of motion, we only display the final state and output equations, and observe that the structure is quite complex as well as nonlinear.

```
/* State equations. */

void compute_dx(double *dx, double *x, double *u, double **p)

{

/* Retrieve model parameters. */

double *F, *M, *C, *d, *A, *I, *m, *K;

F = p[0]; /* Aerodynamic force coefficient. */

M = p[1]; /* Aerodynamic momentum coefficient. */

C = p[2]; /* Aerodynamic compensation factor. */
```

```
d = p[3]; /* Body diameter. */

A = p[4]; /* Body reference area. */

I = p[5]; /* Moment of inertia, x-y-z. */

m = p[6]; /* Mass. */

K = p[7]; /* Feedback gain. */

/* x[0]: Angular velocity around x-axis. */

/* x[1]: Angular velocity around y-axis. */

/* x[2]: Angular velocity around z-axis. */

/* x[3]: Angle of attack. */

/* x[4]: Angle of sideslip. */

dx[0] = 1/I[0]*(d[0]*A[0]*(M[0]*x[4]+0.5*M[1]*d[0]*x[0]/u[4]+M[2]*u[0])*u[3]-
(I[2]-I[1])*x[1]*x[2])+K[0]*(u[5]-x[0]);

dx[1] = 1/I[1]*(d[0]*A[0]*(M[3]*x[3]+0.5*M[4]*d[0]*x[1]/u[4]+M[5]*u[1])*u[3]-
(I[0]-I[2])*x[0]*x[2])+K[0]*(u[6]-x[1]);

dx[2] = 1/I[2]*(d[0]*A[0]*(M[6]*x[4]+M[7]*x[3]*x[4]+0.5*M[8]*d[0]*x[2]/
u[4]+M[9]*u[0]+M[10]*u[2])*u[3]-(I[1]-I[0])*x[0]*x[1])+K[0]*(u[7]-x[2]);

dx[3] = (-A[0]*u[3]*(F[2]*x[3]+F[3]*u[1]))/(m[0]*u[4])-
x[0]*x[4]+x[1]+K[0]*(u[8]/u[4]-x[3])+C[0]*pow(x[4],2);

dx[4] = (-A[0]*u[3]*(F[0]*x[4]+F[1]*u[2]))/(m[0]*u[4])-
x[2]+x[0]*x[3]+K[0]*(u[9]/u[4]-x[4]);

}

/* Output equations. */

void compute_y(double *y, double *x, double *u, double **p)

{

/* Retrieve model parameters. */

double *F, *A, *m;

F = p[0]; /* Aerodynamic force coefficient. */

A = p[4]; /* Body reference area. */

m = p[6]; /* Mass. */

/* y[0]: Angular velocity around x-axis. */

/* y[1]: Angular velocity around y-axis. */
```

```
/* y[2]: Angular velocity around z-axis. */

/* y[3]: Acceleration in y-direction. */

/* y[4]: Acceleration in z-direction. */

y[0] = x[0];

y[1] = x[1];

y[2] = x[2];

y[3] = -A[0]*u[3]*(F[0]*x[4]+F[1]*u[2])/m[0];

y[4] = -A[0]*u[3]*(F[2]*x[3]+F[3]*u[1])/m[0];

}
```

We must also provide initial values of the 23 parameters. We specify some of the parameters (aerodynamic force coefficients, aerodynamic momentum coefficients, and moment of inertia factors) as vectors in 8 different parameter objects. The initial parameter values are obtained partly by physical reasoning and partly by quantitative guessing. The last 4 parameters (A, I, m and K) are more or less constants, so by fixing these parameters we get a model structure with 16 free parameters, distributed among parameter objects F, M, C and d.

```
Parameters = struct('Name', ...
   {'Aerodynamic force coefficient'      ... % F, 4-by-1 vector.
    'Aerodynamic momentum coefficient'   ... % M, 11-by-1 vector.
    'Aerodynamic compensation factor'    ... % C, scalar.
    'Body diameter'                      ...  % d, scalar.
    'Body reference area'                ...  % A, scalar.
    'Moment of inertia, x-y-z'           ... % I, 3-by-1 vector.
    'Mass'                     ...           % m, scalar.
    'Feedback gain'},                    ... % K, scalar.
    'Unit', ...
   {'1/(rad*m^2), 1/(rad*m^2), 1/(rad*m^2), 1/(rad*m^2)' ...
   ['1/rad, 1/(s*rad), 1/rad, 1/rad, '    ...
    '1/(s*rad), 1/rad, 1/rad, 1/rad^2, ' ...
    '1/(s*rad), 1/rad, 1/rad']           ...
    '1/(s*rad)' 'm' 'm^2'                 ...
    'kg*m^2, kg*m^2,kg*m^2' 'kg' '-'},    ...
    'Value', ...
   {[20.0; -6.0; 35.0; 13.0]             ...
    [-1.0; 15; 3.0; -16.0; -1800; -50; 23.0; -200; -2000; -17.0; -50.0] ...
     -5.0, 0.17, 0.0227                  ...
    [0.5; 110; 110] 107 6},              ...
    'Minimum',...
   {-Inf(4, 1) -Inf(11, 1) -Inf -Inf -Inf -Inf(3, 1) -Inf -Inf}, ... % Ignore constraints.
    'Maximum', ...
   {Inf(4, 1) Inf(11, 1) Inf Inf Inf Inf(3, 1) Inf Inf}, ... % Ignore constraints.
    'Fixed', ...
   {false(4, 1) false(11, 1) false true true true(3, 1) true true});
```

We also define the 5 states of the model structure in the same manner:

```
InitialStates = struct('Name', {'Angular velocity around x-axis'       ...
                                'Angular velocity around y-axis'       ...
```

```
                                'Angular velocity around z-axis'      ...
                                'Angle of attack' 'Angle of sideslip'}, ...
                  'Unit',    {'rad/s' 'rad/s' 'rad/s' 'rad' 'rad'},   ...
                  'Value',   {0 0 0 0 0},                             ...
                  'Minimum', {-Inf -Inf -Inf -Inf -Inf},... % Ignore constraints.
                  'Maximum', {Inf Inf Inf Inf Inf},... % Ignore constraints.
                  'Fixed',   {true true true true true});
```

The model file along with order, parameter and initial states data are now used to create an IDNLGREY object describing the system:

```
FileName    = 'aero_c';              % File describing the model structure.
Order       = [5 10 5];              % Model orders [ny nu nx].
Ts          = 0;                     % Time-continuous system.
nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts, ...
                'Name', 'Model', 'TimeUnit', 's');
```

Next, we use data from the IDDATA object to specify the input and output signals of the system:

```
nlgr.InputName = z.InputName;
nlgr.InputUnit = z.InputUnit;
nlgr.OutputName = z.OutputName;
nlgr.OutputUnit = z.OutputUnit;
```

Thus, we have an IDNLGREY object with 10 input signals, 5 states, and 5 output signals. As mentioned previously, the model also contains 23 parameters, 7 of which are fixed and 16 of which are free:

```
nlgr
```

Performance of the Initial Model

Before estimating the 16 free parameters, we simulate the system using the initial parameter vector. Simulation provides useful information about the quality of the initial model:

```
clf
compare(z, nlgr); % simulate the model and compare the response to measured values
```

Figure 3: Comparison between measured outputs and the simulated outputs of the initial model.

From the plot, we see that the measured and simulated signals match each other closely except for the time span 4 to 6 seconds. This fact is clearly revealed in a plot of the prediction errors:

```
figure;
h_gcf = gcf;
Pos = h_gcf.Position;
h_gcf.Position = [Pos(1), Pos(2)-Pos(4)/2, Pos(3), Pos(4)*1.5];
pe(z, nlgr);
```

Figure 4: Prediction errors of the initial model.

Parameter Estimation

The initial model as defined above is a plausible starting point for parameter estimation. Next, we compute the prediction error estimate of the 16 free parameters. This computation will take some time.

```
duration = clock;
nlgr = nlgreyest(z, nlgr, nlgreyestOptions('Display', 'on'));
duration = etime(clock, duration);
```

Performance of the Estimated Aerodynamic Body Model

On the computer used, estimation of the parameters took the following amount of time to complete:

```
disp(['Estimation time   : ' num2str(duration, 4) ' seconds']);
disp(['Time per iteration: ' num2str(duration/nlgr.Report.Termination.Iterations, 4) ' seconds.']
```

To evaluate the quality of the estimated model and to illustrate the improvement compared to the initial model, we simulate the estimated model and compare the measured and simulated outputs:

```
clf
compare(z, nlgr);
```

Figure 5: Comparison between measured outputs and the simulated outputs of the estimated model.

The figure clearly indicates the improvement compared to the simulation result obtained with the initial model. The system dynamics in the time span 4 to 6 seconds is now captured with much higher accuracy than before. This is best displayed by looking at the prediction errors:

```
figure;
h_gcf = gcf;
Pos = h_gcf.Position;
h_gcf.Position = [Pos(1), Pos(2)-Pos(4)/2, Pos(3), Pos(4)*1.5];
pe(z, nlgr);
```

Figure 6: Prediction errors of the estimated model.

Let us conclude the case study by displaying the model and the estimated uncertainty:

```
present(nlgr);
```

Concluding Remarks

The estimated model is a good starting point for investigating the fundamental performance of different control strategies. High-fidelity models that preferably have a physical interpretation are, for example, vital components of so-called "model predictive control systems".

Additional Information

For more information on identifying dynamic systems with System Identification Toolbox™, see the System Identification Toolbox product page.

Modeling an Industrial Robot Arm

This example shows grey-box modeling of the dynamics of an industrial robot arm. The robot arm is described by a nonlinear three-mass flexible model according to Figure 1. This model is idealized in the sense that the movements are assumed to be around an axis not affected by gravity. For simplicity, the modeling is also performed with gear ratio r = 1 and the true physical parameters are afterwards obtained through a straightforward scaling with the true gear ratio. The modeling and identification experiments detailed below is based on the work published in

> E. Wernholt and S. Gunnarsson. Nonlinear Identification of a
> Physically Parameterized Robot Model. In preprints of the 14th IFAC
> Symposium on System Identification, pages 143-148, Newcastle,
> Australia, March 2006.
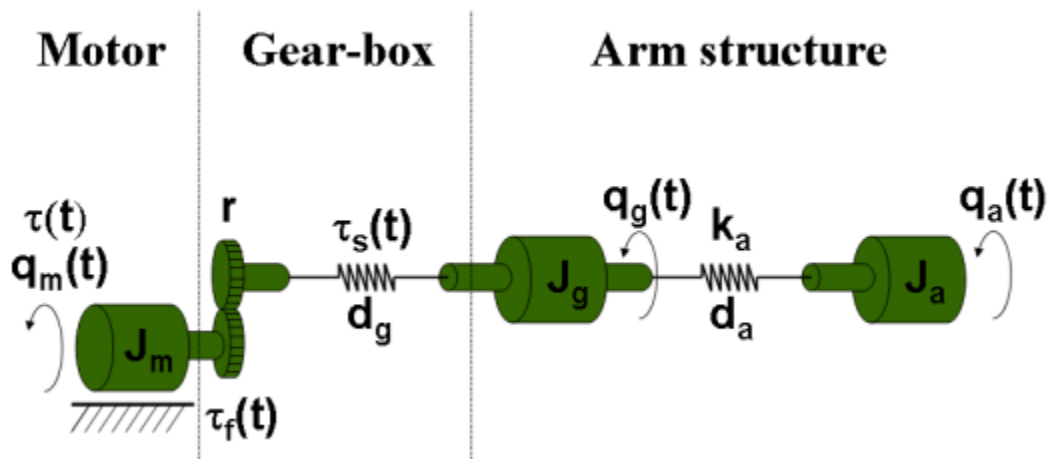
Figure 1: Schematic diagram of an industrial robot arm.

Modeling the Robot Arm

The input to the robot is the applied torque u(t)=tau(t) generated by the electrical motor, and the resulting angular velocity of the motor y(t) = d/dt q_m(t) is the measured output. The angular positions of the masses after the gear-box and at the end of the arm structure, q_g(t) and q_a(t), are non-measurable. Flexibilities within the gear-box is modeled by a nonlinear spring, described by the spring torque tau_s(t), which is located between the motor and the second mass, while the "linear" spring between the last two masses models flexibilities in the arm structure. The friction of the system acts mainly on the first mass and is here modeled by a nonlinear friction torque tau_f(t).

Introducing the states:

```
        ( x1(t) )   ( q_m(t) - q_g(t) )
        ( x2(t) )   ( q_g(t) - q_a(t) )
x(t) =  ( x3(t) ) = (   d/dt q_m(t)   )
        ( x4(t) )   (   d/dt q_g(t)   )
        ( x5(t) )   (   d/dt q_a(t)   )
```

and applying torque balances for the three masses result in the following nonlinear state-space model structure:

```
d/dt x1(t) = x3(t) - x4(t)
d/dt x2(t) = x4(t) - x5(t)
d/dt x3(t) = 1/J_m*(-tau_s(t) - d_g*(x3(t)-x4(t)) - tau_f(t) + u(t))
d/dt x4(t) = 1/J_g*(tau_s(t) + d_g*(x3(t)-x4(t)) - k_a*x2(t) - d_a*(x4(t)-x5(t)))
d/dt x5(t) = 1/J_a*(k_a*x2(t) + d_a*(x4(t)-x5(t)))

      y(t) = x3(t)
```

where J_m, J_g, and J_a are the moments of inertia of the motor, the gear-box and the arm structure, respectively, d_g and d_a are damping parameters, and k_a is the stiffness of the arm structure.

The gear-box friction torque, tau_f(t), is modeled to include many of the friction phenomena encountered in practice, among other things so-called Coulomb friction and the Stribeck effect:

```
tau_f(t) = Fv*x3(t) + (Fc+Fcs*sech(alpha*x3(t)))*tanh(beta*x3(t))
```

where Fv and Fc are the viscous and the Coulomb friction coefficients, Fcs and alpha are coefficients for reflecting the Stribeck effect, and beta a parameter used to obtain a smooth transition from negative to positive velocities of x3(t). (A similar approach, but based on a slightly different model structure, for describing the static relationship between the velocity and the friction torque/force is further discussed in the tutorial named idnlgreydemo5: "Static Modeling of Friction".)

The torque of the spring, tau_s(t), is assumed to be described by a cubic polynomial without a square term in x1(t):

```
tau_s(t) = k_g1*x1(t) + k_g3*x1(t)^3
```

where k_g1 and k_g3 are two stiffness parameters of the gear-box spring.

In other types of identification experiments discussed in the paper by Wernholt and Gunnarsson, it is possible to identify the overall moment of inertia J = J_m+J_g+J_a. With this we can introduce the unknown scaling factors a_m and a_g, and perform the following reparameterizations:

```
J_m = J*a_m
J_g = J*a_g
J_a = J*(1-a_m-a_g)
```

where (if J is known) only a_m and a_g need to be estimated.

All in all, this gives the following state space structure, involving 13 different parameters: Fv, Fc, Fcs, alpha, beta, J, a_m, a_g, k_g1, k_g3, d_g, k_a and d_a. (By definition we have also used the fact that sech(x) = 1/cosh(x).)

```
  tau_f(t) = Fv*x3(t) + (Fc+Fcs/cosh(alpha*x3(t)))*tanh(beta*x3(t))
  tau_s(t) = k_g1*x1(t) + k_g3*x1(t)^3

d/dt x1(t) = x3(t) - x4(t)
d/dt x2(t) = x4(t) - x5(t)
d/dt x3(t) = 1/(J*a_m)*(-tau_s(t) - d_g*(x3(t)-x4(t)) - tau_f(t) + u(t))
d/dt x4(t) = 1/(J*a_g)*(tau_s(t) + d_g*(x3(t)-x4(t)) - k_a*x2(t) - d_a*(x4(t)-x5(t)))
d/dt x5(t) = 1/(J(1-a_m-a_g))*(k_a*x2(t) + d_a*(x4(t)-x5(t)))

      y(t) = x3(t)
```

IDNLGREY Robot Arm Model Object

The above model structure is entered into a C MEX-file named robotarm_c.c, with state and output update functions as follows (the whole file can be viewed by the command "type robotarm_c.c"). In

the state update function, notice that we have here used two intermediate double variables, on one hand to enhance the readability of the equations and on the other hand to improve the execution speed (taus appears twice in the equations, but is only computed once).

```
/* State equations. */
void compute_dx(double *dx, double *x, double *u, double **p)
{
    /* Declaration of model parameters and intermediate variables. */
    double *Fv, *Fc, *Fcs, *alpha, *beta, *J, *am, *ag, *kg1, *kg3, *dg, *ka, *da;
    double tauf, taus;   /* Intermediate variables. */

    /* Retrieve model parameters. */
    Fv    = p[0];    /* Viscous friction coefficient.           */
    Fc    = p[1];    /* Coulomb friction coefficient.           */
    Fcs   = p[2];    /* Striebeck friction coefficient.         */
    alpha = p[3];    /* Striebeck smoothness coefficient.       */
    beta  = p[4];    /* Friction smoothness coefficient.        */
    J     = p[5];    /* Total moment of inertia.                */
    am    = p[6];    /* Motor moment of inertia scale factor.   */
    ag    = p[7];    /* Gear-box moment of inertia scale factor. */
    kg1   = p[8];    /* Gear-box stiffness parameter 1.         */
    kg3   = p[9];    /* Gear-box stiffness parameter 3.         */
    dg    = p[10];   /* Gear-box damping parameter.             */
    ka    = p[11];   /* Arm structure stiffness parameter.      */
    da    = p[12];   /* Arm structure damping parameter.        */

    /* Determine intermediate variables. */
    /* tauf: Gear friction torque. (sech(x) = 1/cosh(x)! */
    /* taus: Spring torque. */
    tauf = Fv[0]*x[2]+(Fc[0]+Fcs[0]/(cosh(alpha[0]*x[2])))*tanh(beta[0]*x[2]);
    taus = kg1[0]*x[0]+kg3[0]*pow(x[0],3);

    /* x[0]: Rotational velocity difference between the motor and the gear-box. */
    /* x[1]: Rotational velocity difference between the gear-box and the arm. */
    /* x[2]: Rotational velocity of the motor. */
    /* x[3]: Rotational velocity after the gear-box. */
    /* x[4]: Rotational velocity of the robot arm. */
    dx[0] = x[2]-x[3];
    dx[1] = x[3]-x[4];
    dx[2] = 1/(J[0]*am[0])*(-taus-dg[0]*(x[2]-x[3])-tauf+u[0]);
    dx[3] = 1/(J[0]*ag[0])*(taus+dg[0]*(x[2]-x[3])-ka[0]*x[1]-da[0]*(x[3]-x[4]));
    dx[4] = 1/(J[0]*(1.0-am[0]-ag[0]))*(ka[0]*x[1]+da[0]*(x[3]-x[4]));
}

/* Output equation. */
void compute_y(double y[], double x[])
{
    /* y[0]: Rotational velocity of the motor. */
    y[0] = x[2];
}
```

The next step is to create an IDNLGREY object reflecting the modeling situation. It should here be noted that finding proper initial parameter values for the robot arm requires some additional effort. In the paper by Wernholt and Gunnarsson, this was carried out in two preceding steps, where other model structures and identification techniques were employed. The initial parameter values used below are the results of those identification experiments.

```
FileName       = 'robotarm_c';              % File describing the model structure.
Order          = [1 1 5];                   % Model orders [ny nu nx].
```

```
Parameters     = [ 0.00986346744839  0.74302635727901 ...
                   3.98628540790595  3.24015074090438 ...
                   0.79943497008153  0.03291699877416 ...
                   0.17910964111956  0.61206166914114 ...
                  20.59269827430799  0.00000000000000 ...
                   0.06241814047290 20.23072060978318 ...
                   0.00987527995798]';        % Initial parameter vector.
InitialStates = zeros(5, 1);                  % Initial states.
Ts            = 0;                            % Time-continuous system.
nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts, ...
                'Name', 'Robot arm',                          ...
                'InputName', 'Applied motor torque',          ...
                'InputUnit', 'Nm',                            ...
                'OutputName', 'Angular velocity of motor',     ...
                'OutputUnit', 'rad/s',                        ...
                'TimeUnit', 's');
```

The names and the units of the states are provided for better bookkeeping:

```
nlgr = setinit(nlgr, 'Name', {'Angular position difference between the motor and the gear-box' .
                     'Angular position difference between the gear-box and the arm'    ...
                     'Angular velocity of motor'                                       ...
                     'Angular velocity of gear-box'                                    ...
                     'Angular velocity of robot arm'}');
nlgr = setinit(nlgr, 'Unit', {'rad' 'rad' 'rad/s' 'rad/s' 'rad/s'});
```

The parameter names are also specified in detail. Furthermore, the modeling was done is such a way that all parameters ought to be positive, i.e., the minimum of each parameter should be set to 0 (and hence constrained estimation will later on be performed). As in the paper by Wernholt and Gunnarsson, we also consider the first 6 parameters, i.e., Fv, Fc, Fcs, alpha, beta, and J, to be so good that they do not need to be estimated.

```
nlgr = setpar(nlgr, 'Name', {'Fv   : Viscous friction coefficient'          ... % 1.
                     'Fc   : Coulomb friction coefficient'          ... % 2.
                     'Fcs  : Striebeck friction coefficient'        ... % 3.
                     'alpha: Striebeck smoothness coefficient'      ... % 4.
                     'beta : Friction smoothness coefficient'       ... % 5.
                     'J    : Total moment of inertia'               ... % 6.
                     'a_m  : Motor moment of inertia scale factor'  ... % 7.
                     'a_g  : Gear-box moment of inertia scale factor' ... % 8.
                     'k_g1 : Gear-box stiffness parameter 1'         ... % 9.
                     'k_g3 : Gear-box stiffness parameter 3'         ... % 10.
                     'd_g  : Gear-box damping parameter'             ... % 11.
                     'k_a  : Arm structure stiffness parameter'      ... % 12.
                     'd_a  : Arm structure damping parameter'        ... % 13.
                    });
nlgr = setpar(nlgr, 'Minimum', num2cell(zeros(size(nlgr, 'np'), 1)));   % All parameters >= 0!
for parno = 1:6   % Fix the first six parameters.
    nlgr.Parameters(parno).Fixed = true;
end
```

The modeling steps carried out so far have left us with an initial robot arm model with properties as follows:

```
present(nlgr);
```

```
nlgr =
```

```
Continuous-time nonlinear grey-box model defined by 'robotarm_c' (MEX-file):

   dx/dt = F(t, u(t), x(t), p1, ..., p13)
    y(t) = H(t, u(t), x(t), p1, ..., p13) + e(t)

 with 1 input(s), 5 state(s), 1 output(s), and 7 free parameter(s) (out of 13).

 Inputs:
    u(1)   Applied motor torque(t) [Nm]
 States:                                                                 Initial value
    x(1)   Angular position difference between the motor and the gear-box(t) [rad]    xinit@exp1
    x(2)   Angular position difference between the gear-box and the arm(t) [rad]      xinit@exp1
    x(3)   Angular velocity of motor(t) [rad/s]                                       xinit@exp1
    x(4)   Angular velocity of gear-box(t) [rad/s]                                    xinit@exp1
    x(5)   Angular velocity of robot arm(t) [rad/s]                                   xinit@exp1
 Outputs:
    y(1)   Angular velocity of motor(t) [rad/s]
 Parameters:                                              Value
    p1    Fv   : Viscous friction coefficient             0.00986347    (fixed) in [0, Inf]
    p2    Fc   : Coulomb friction coefficient             0.743026      (fixed) in [0, Inf]
    p3    Fcs  : Striebeck friction coefficient           3.98629       (fixed) in [0, Inf]
    p4    alpha: Striebeck smoothness coefficient         3.24015       (fixed) in [0, Inf]
    p5    beta : Friction smoothness coefficient          0.799435      (fixed) in [0, Inf]
    p6    J    : Total moment of inertia                  0.032917      (fixed) in [0, Inf]
    p7    a_m  : Motor moment of inertia scale factor     0.17911       (estimated) in [0, ]
    p8    a_g  : Gear-box moment of inertia scale factor  0.612062      (estimated) in [0, ]
    p9    k_g1 : Gear-box stiffness parameter 1           20.5927       (estimated) in [0, ]
    p10   k_g3 : Gear-box stiffness parameter 3           0             (estimated) in [0, ]
    p11   d_g  : Gear-box damping parameter               0.0624181     (estimated) in [0, ]
    p12   k_a  : Arm structure stiffness parameter        20.2307       (estimated) in [0, ]
    p13   d_a  : Arm structure damping parameter          0.00987528    (estimated) in [0, ]

Name: Robot arm

Status:
Created by direct construction or transformation. Not estimated.
More information in model's "Report" property.

<a href="matlab:if exist('nlgr','var'), if isa(nlgr,'idnlgrey'), disp(' '); disp(get(nlgr)), else
```

Input-Output Data

A large number of real-world data sets were collected from the experimental robot. In order to keep the robot around its operating point, but also for safety reasons, the data was collected using an experimental feedback control arrangement, which subsequently allows off-line computations of the reference signals for the joint controllers.

In this case study we will limit the onward discussion to four different data sets, one for estimation and the remaining ones for validation purposes. In each case, a periodic excitation signal with approximately 10 seconds duration was employed to generate a reference speed for the controller. The chosen sampling frequency was 2 kHz (sample time, Ts, = 0.0005 seconds). For the data sets, three different types of input signals were used: (ue: input signal of the estimation data set; uv1, uv2, uv3: input signals of the three validation data sets)

```
  ue, uv1: Multisine signals with a flat amplitude spectrum in the
           frequency interval 1-40 Hz with a peak value of 16 rad/s. The
```

> multisine signal is superimposed on a filtered square wave
> with amplitude 20 rad/s and cut-off frequency 1 Hz.

> uv2: Similar to ue and uv1, but without the square wave.

> uv3: Multisine signal (sum of sinusoids) with frequencies 0.1,
> 0.3, and 0.5 Hz, with peak value 40 rad/s.

Let us load the available data and put all four data sets into one single IDDATA object z:

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'robotarmdata'));
z = iddata({ye yv1 yv2 yv3}, {ue uv1 uv2 uv3}, 0.5e-3, 'Name', 'Robot arm');
z.InputName = 'Applied motor torque';
z.InputUnit = 'Nm';
z.OutputName = 'Angular velocity of motor';
z.OutputUnit = 'rad/s';
z.ExperimentName = {'Estimation' 'Validation 1'  'Validation 2' 'Validation 3'};
z.Tstart = 0;
z.TimeUnit = 's';
present(z);
```

```
th =
Time domain data set containing 4 experiments.

Experiment                        Samples        Sample Time
   Estimation                      19838             0.0005
   Validation 1                    19838             0.0005
   Validation 2                    19838             0.0005
   Validation 3                    19838             0.0005

Name: Robot arm

Outputs                           Unit (if specified)
   Angular velocity of motor         rad/s

Inputs                            Unit (if specified)
   Applied motor torque              Nm
```

The following figure shows the input-output data used in the four experiments.

```
figure('Name', [z.Name ': input-output data'],...
   'DefaultAxesTitleFontSizeMultiplier',1,...
   'DefaultAxesTitleFontWeight','normal',...
   'Position',[100 100 900 600]);
for i = 1:z.Ne
   zi = getexp(z, i);
   subplot(z.Ne, 2, 2*i-1);   % Input.
   plot(zi.u);
   title([z.ExperimentName{i} ': ' zi.InputName{1}],'FontWeight','normal');
   if (i < z.Ne)
       xlabel('');
   else
       xlabel([z.Domain ' (' zi.TimeUnit ')']);
   end
   subplot(z.Ne, 2, 2*i);     % Output.
   plot(zi.y);
   title([z.ExperimentName{i} ': ' zi.OutputName{1}],'FontWeight','normal');
```

```
    if (i < z.Ne)
        xlabel('');
    else
        xlabel([z.Domain ' (' zi.TimeUnit ')']);
    end
end
```
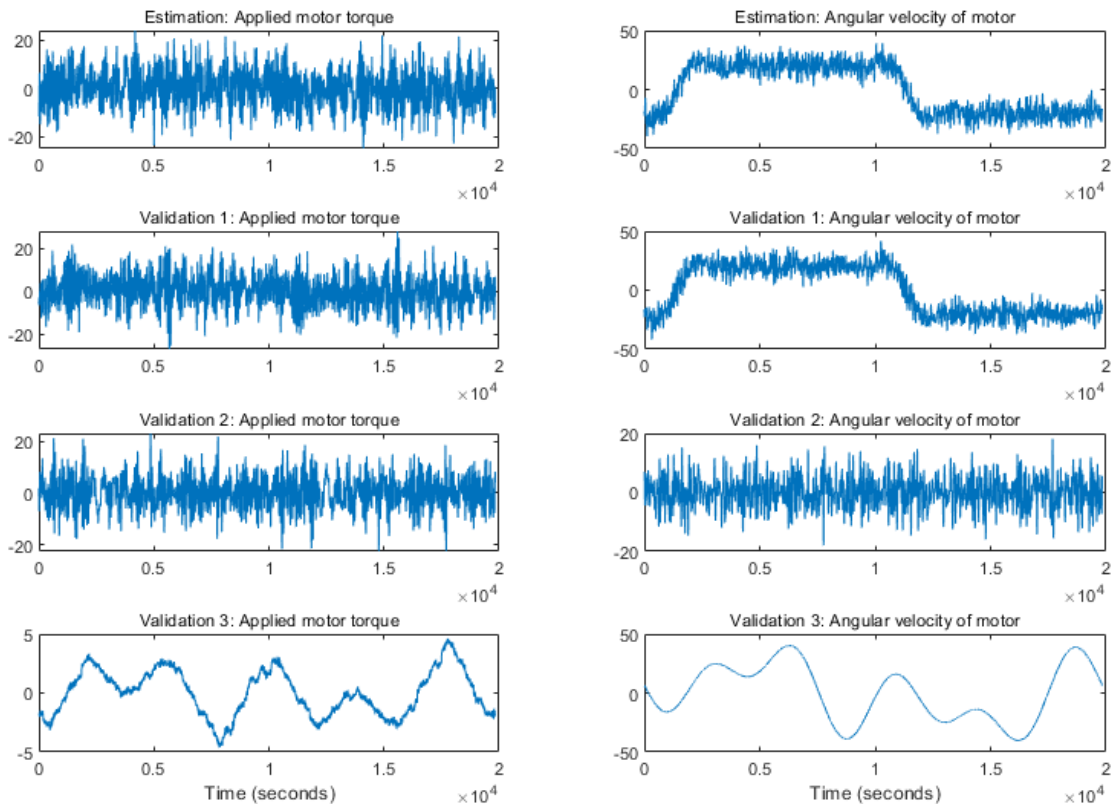

Figure 2: Measured input-output data of an experimental robot arm.

Performance of the Initial Robot Arm Model

How good is the initial robot arm model? Let us use COMPARE to simulate the model outputs (for all four experiments) and compare the result with the corresponding measured outputs. For all four experiments, we know that the values of the first two states are 0 (fixed), while the values of the remaining three states are initially set to the measured output at the starting time (non-fixed). However, by default COMPARE estimates all initial states, and with z holding four different experiments this would mean 4*5 = 20 initial states to estimate. Even after fixing the first two states, 4*3 = 12 initial states would remain to estimate (in case the internal model initial state strategy is followed). Because the data set is rather large, this would result in lengthy computations, and to avoid this we estimate the 4*3 free components of the initial states using PREDICT (possible if the initial state is passed as an initial state structure), but restrict the estimation to the first 10:th of the available data. We then instruct COMPARE to use the resulting 5-by-4 initial state matrix X0init without performing any initial state estimation.

```
zred = z(1:round(zi.N/10));
nlgr = setinit(nlgr, 'Fixed', {true true false false false});
X0 = nlgr.InitialStates;
[X0.Value] = deal(zeros(1, 4), zeros(1, 4), [ye(1) yv1(1) yv2(1) yv3(1)], ...
    [ye(1) yv1(1) yv2(1) yv3(1)], [ye(1) yv1(1) yv2(1) yv3(1)]);
[~, X0init] = predict(zred, nlgr, [], X0);
nlgr = setinit(nlgr, 'Value', num2cell(X0init(:, 1)));
clf
compare(z, nlgr, [], compareOptions('InitialCondition', X0init));
```
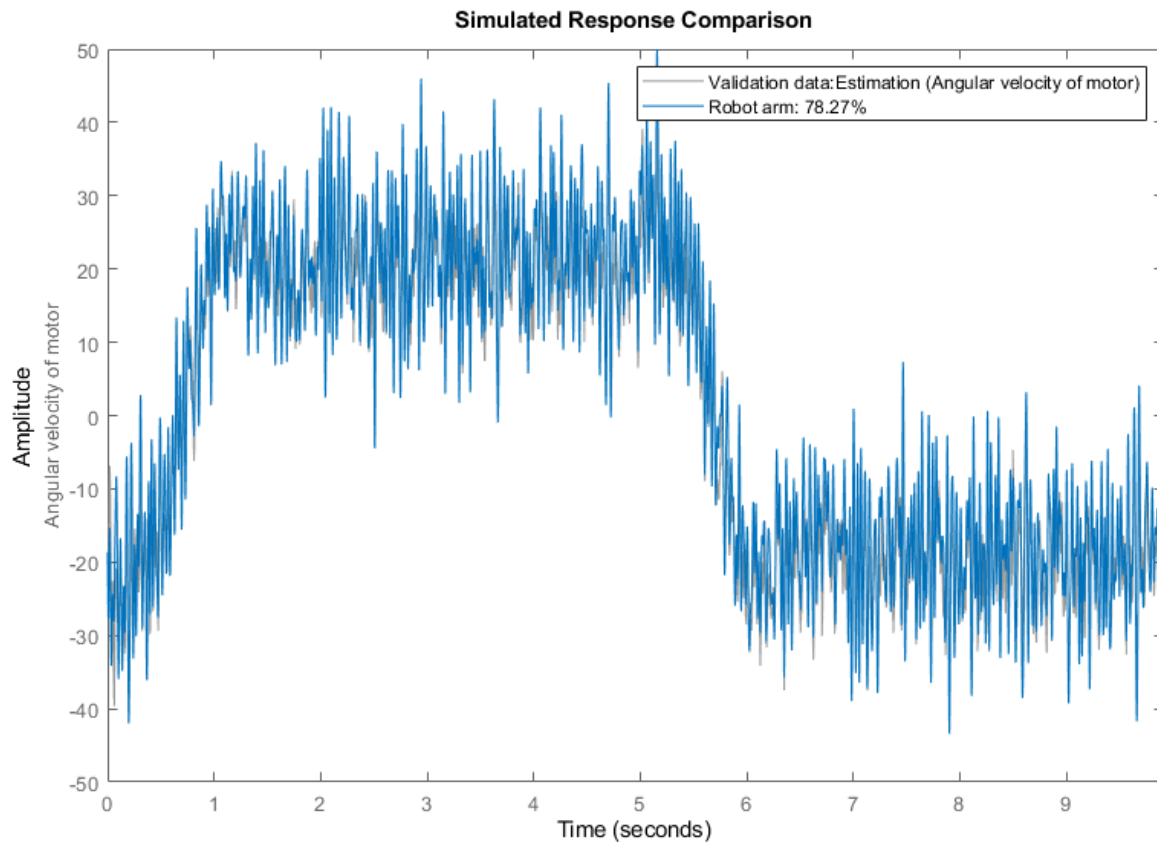

Figure 3: Comparison between measured outputs and the simulated outputs of the initial robot arm model.

As can be seen, the performance of the initial robot arm model is decent or quite good. The fit for the three types of data sets are around 79% for ye and yv1, 37% for yv2, and 95% for yv3. Notice that the higher fit for ye/yv1 as compared to yv2 is in large due to the initial model's ability to capture the square wave, while the multisine part is not captured equally well. We can also look at the prediction errors for the four experiments:

```
pe(z, nlgr, peOptions('InitialCondition',X0init));
```
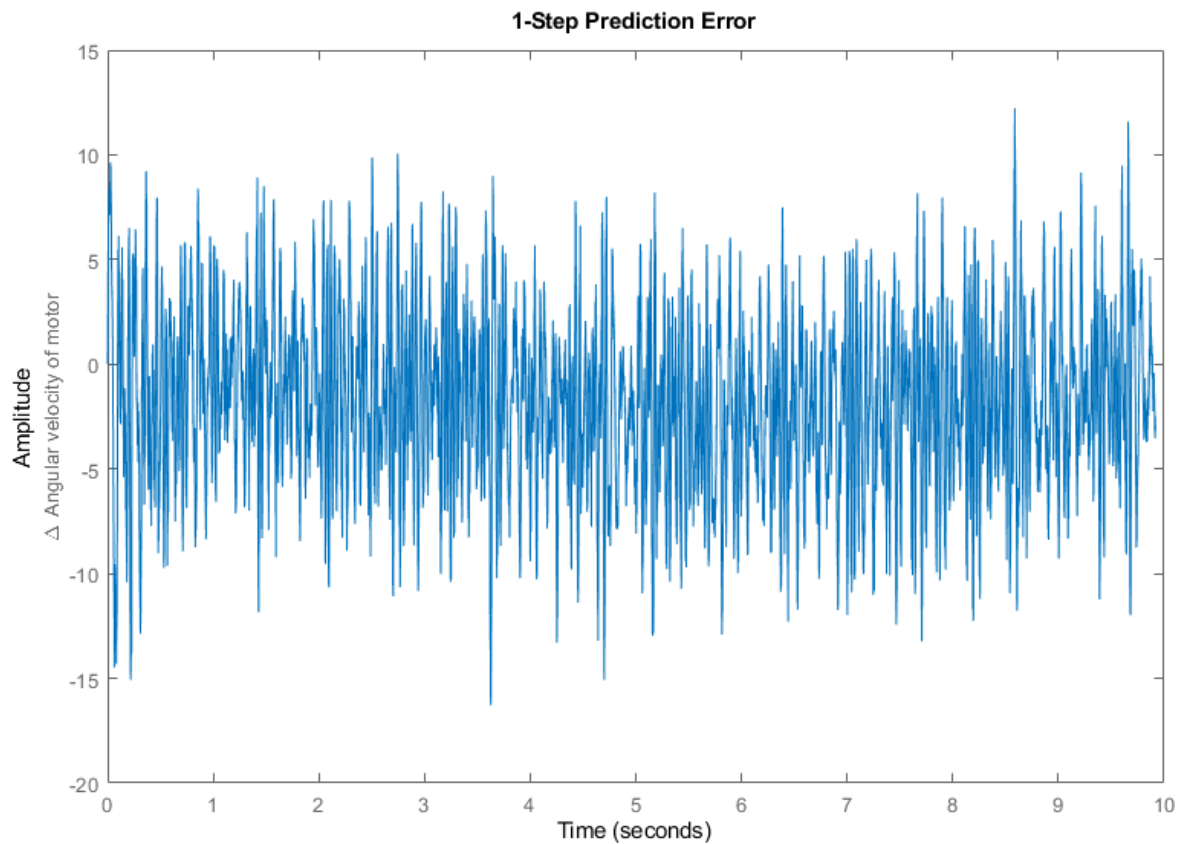
Figure 4: Prediction errors of the initial robot arm model.

Parameter Estimation

Let us now try to improve the performance of the initial robot arm model by estimating the 7 free model parameters and the 3 free initial states of the first experiment of z (the estimation data set). This estimation will take some time (typically a couple of minutes).

```
nlgr = nlgreyest(nlgr, getexp(z, 1), nlgreyestOptions('Display', 'on'));
```

```
Nonlinear Grey Box Model Estimation
Data has 1 outputs, 1 inputs and 19838 samples.
ODE Function: robotarm_c
Number of parameters: 13
```

ESTIMATION PROGRESS

```
Algorithm: Trust-Region Reflective Newton
-----------------------------------------------------------------
                              Norm of      First-order
Iteration      Cost            step         optimality
-----------------------------------------------------------------
     0        20.5003            -               -
     1        10.4968          7.57          6.48e+05
     2         9.91002        11.8           2.65e+05
     3         9.70427         2.4           1.38e+05
     4         9.59532         1.85          2.84e+04
     5         9.52905         5.35          5.72e+04
     6         9.50683         1.02          6.13e+04
     7         9.49771         0.857         4.11e+04
     8         9.49446         0.326         2.08e+04
     9         9.49385         0.315         6.15e+03
```

RESULT

```
Termination condition: Change in cost was less than the specified tolerance..
Number of iterations: 10, Number of function evaluations: 11

Status: Estimated using NLGREYEST
Fit to estimation data: 85.21%, FPE: 9.50341
```

■ Stop   Close

Performance of the Estimated Robot Arm Model

COMPARE is again used to assess the performance of the estimated robot arm model. We also here instruct COMPARE to not perform any initial state estimation. For the first experiment we replace the guessed initial state with the one estimated by NLGREYEST and for the remaining three experiments we employ PREDICT to estimate the initial state based on the reduced IDDATA object zred.

```
X0init(:, 1) = cell2mat(getinit(nlgr, 'Value'));
X0 = nlgr.InitialStates;
[X0.Value] = deal(zeros(1, 3), zeros(1, 3), [yv1(1) yv2(1) yv3(1)], ...
    [yv1(1) yv2(1) yv3(1)], [yv1(1) yv2(1) yv3(1)]);
[yp, X0init(:, 2:4)] = predict(getexp(zred, 2:4), nlgr, [], X0);
```

```
clf
compare(z, nlgr, [], compareOptions('InitialCondition', X0init));
```
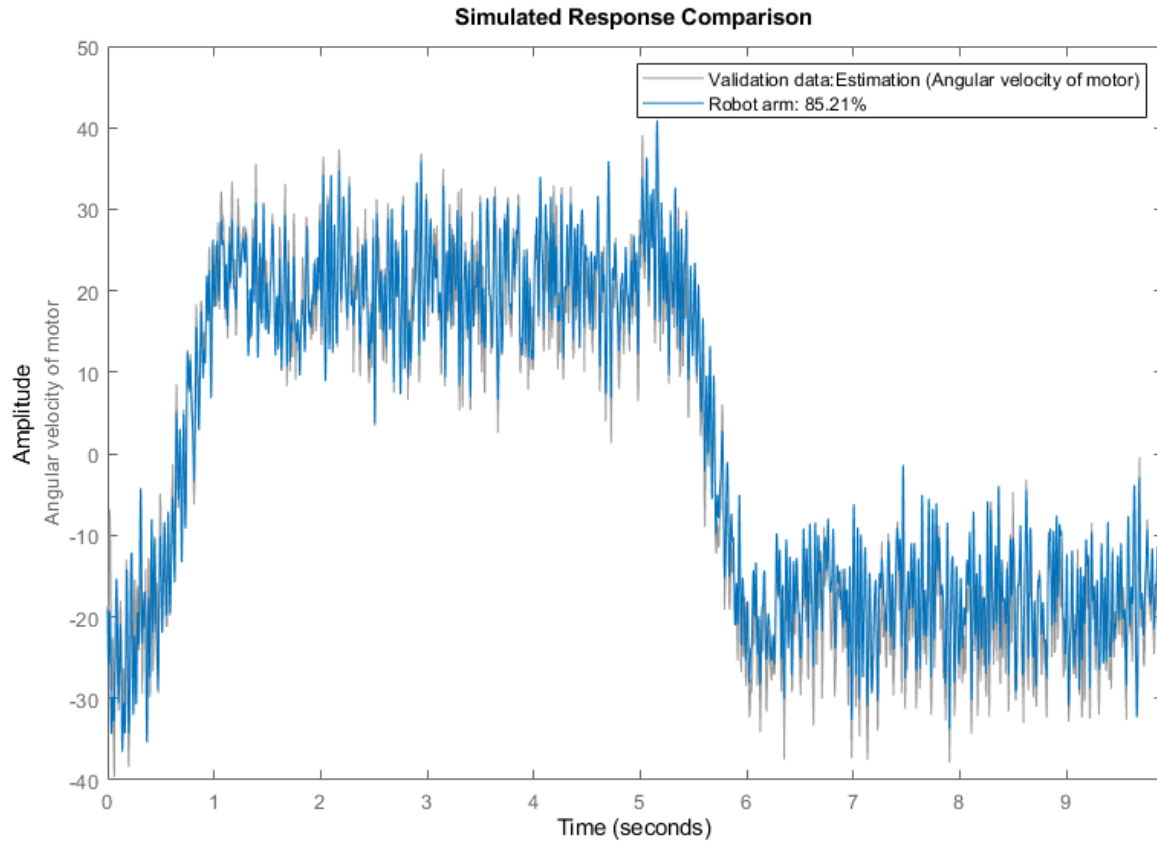

Figure 5: Comparison between measured outputs and the simulated outputs of the estimated robot arm model.

The comparison plot shows an improvement in terms of better fits. For ye and yv1 the fit is now around 85% (before: 79%), for yv2 around 63% (before: 37%), and for yv3 somewhat less than 95.5% (before: also little less than 95.5%). The improvement is most pronounced for the second validation data set, where a multisine signal without any square wave was applied as the input. However, the estimated model's ability to follow the multisine part of ye and yv1 has also been improved considerably (yet this is not reflected by the fit figures, as these are more influenced by the fit to the square wave). A plot of the prediction errors also reveals that the residuals are now in general smaller than with the initial robot arm model:

```
figure;
pe(z, nlgr, peOptions('InitialCondition',X0init));
```
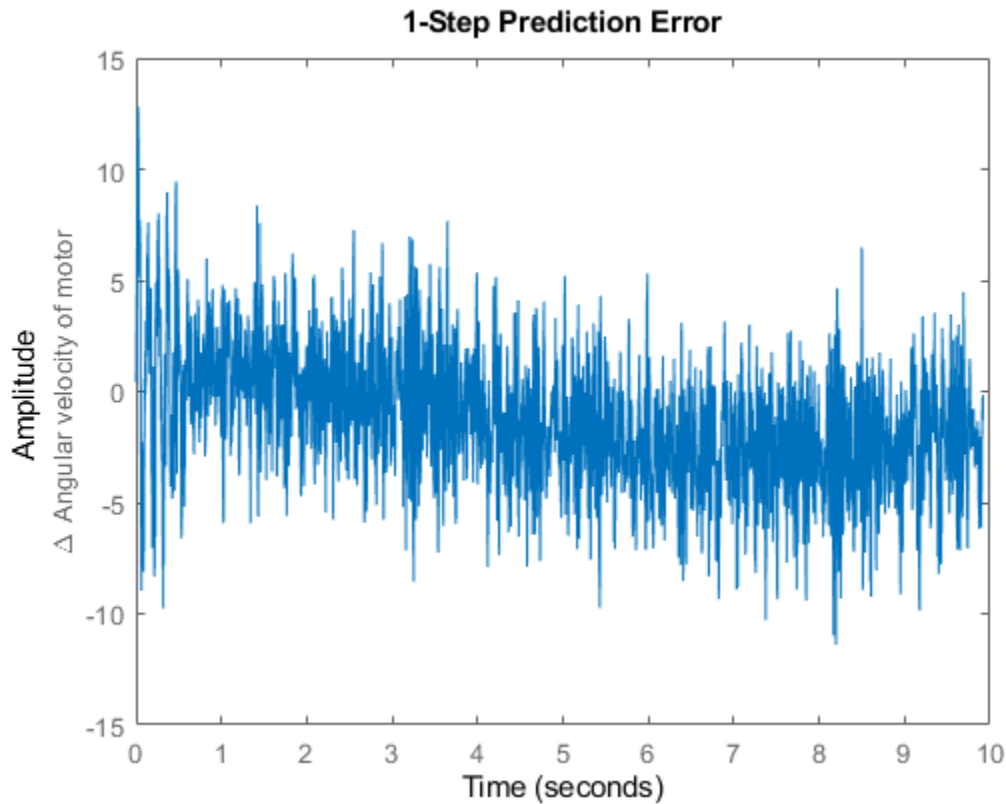
Figure 6: Prediction errors of the estimated robot arm model.

We conclude the case study by textually summarizing various properties of the estimated robot arm model.

```
present(nlgr);
```

```
nlgr =

Continuous-time nonlinear grey-box model defined by 'robotarm_c' (MEX-file):

   dx/dt = F(t, u(t), x(t), p1, ..., p13)
    y(t) = H(t, u(t), x(t), p1, ..., p13) + e(t)

 with 1 input(s), 5 state(s), 1 output(s), and 7 free parameter(s) (out of 13).

 Inputs:
    u(1)  Applied motor torque(t) [Nm]
 States:                                                                 Initial value
    x(1)   Angular position difference between the motor and the gear-box(t) [rad]   xinit@exp1
    x(2)   Angular position difference between the gear-box and the arm(t) [rad]     xinit@exp1
    x(3)   Angular velocity of motor(t) [rad/s]                                      xinit@exp1
    x(4)   Angular velocity of gear-box(t) [rad/s]                                   xinit@exp1
    x(5)   Angular velocity of robot arm(t) [rad/s]                                  xinit@exp1
 Outputs:
    y(1)   Angular velocity of motor(t) [rad/s]
 Parameters:                                                Value    Standard Deviation
```

```
    p1    Fv   : Viscous friction coefficient                    0.00986347             0    (fixe
    p2    Fc   : Coulomb friction coefficient                      0.743026             0    (fixe
    p3    Fcs  : Striebeck friction coefficient                     3.98629             0    (fixe
    p4    alpha: Striebeck smoothness coefficient                   3.24015             0    (fixe
    p5    beta : Friction smoothness coefficient                   0.799435             0    (fixe
    p6    J    : Total moment of inertia                           0.032917             0    (fixe
    p7    a_m  : Motor moment of inertia scale factor              0.266481   0.000286927    (estim
    p8    a_g  : Gear-box moment of inertia scale factor           0.647529   0.000190952    (estim
    p9    k_g1 : Gear-box stiffness parameter 1                     20.0772     0.0263493    (estim
    p10   k_g3 : Gear-box stiffness parameter 3                     24.1797      0.332314    (estim
    p11   d_g  : Gear-box damping parameter                       0.0305121   0.000282685    (estim
    p12   k_a  : Arm structure stiffness parameter                 11.7554     0.0311118    (estim
    p13   d_a  : Arm structure damping parameter                 0.00283442   8.05123e-05    (estim

Name: Robot arm

Status:
Termination condition: Change in cost was less than the specified tolerance..
Number of iterations: 10, Number of function evaluations: 11

Estimated using Solver: ode45; Search: lsqnonlin on time domain data "Robot arm".
Fit to estimation data: 85.21%
FPE: 9.503, MSE: 9.494
More information in model's "Report" property.

<a href="matlab:if exist('nlgr','var'), if isa(nlgr,'idnlgrey'), disp(' '); disp(get(nlgr)), else
```

Concluding Remarks

System identification techniques are widely used in robotics. "Good" robot models are vital for modern robot control concepts, and are often considered as a necessity for meeting the continuously increasing demand in speed and precision. The models are also crucial components in various robot diagnosis applications, where the models are used for predicting problems related to wear and for detecting the actual cause of a robot malfunction.

Two Tank System: C MEX-File Modeling of Time-Continuous SISO System

This example shows how to perform IDNLGREY modeling based on C MEX model files. It uses a simple system where nonlinear state space modeling really pays off.

A Two Tank System

The objective is to model the liquid level of the lower tank of a laboratory scale two tank system, schematically shown in Figure 1.
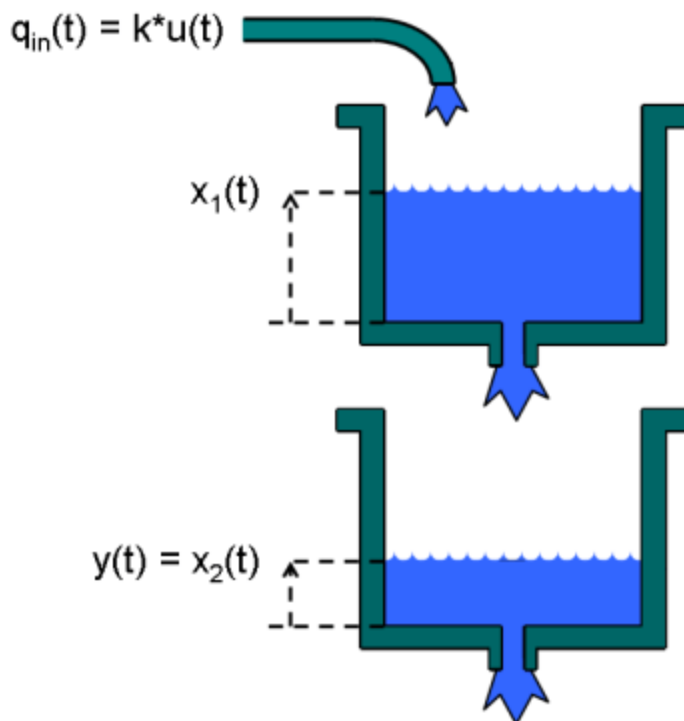
Figure 1: Schematic view of a two tank system.

Input-Output Data

We start the modeling job by loading the available input-output data, which was simulated using the below IDNLGREY model structure, with noise added to the output. The twotankdata.mat file contains one data set with 3000 input-output samples, generated using a sampling rate (Ts) of 0.2 seconds. The input u(t) is the voltage [V] applied to a pump, which generates an inflow to the upper tank. A rather small hole at the bottom of this upper tank yields an outflow that goes into the lower tank, and the output y(t) of the two tank system is then the liquid level [m] of the lower tank. We create an IDDATA object z to hold the tank data. For bookkeeping and documentation purposes we also specify channel names and units. This step is optional.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'twotankdata'));
z = iddata(y, u, 0.2, 'Name', 'Two tanks');
set(z, 'InputName', 'Pump voltage', 'InputUnit', 'V',              ...
```

```
        'OutputName', 'Lower tank water level', 'OutputUnit', 'm', ...
        'Tstart', 0, 'TimeUnit', 's');
```

The input-output data that will be used for estimation are shown in a plot window.

```
figure('Name', [z.Name ': input-output data']);
plot(z);
```
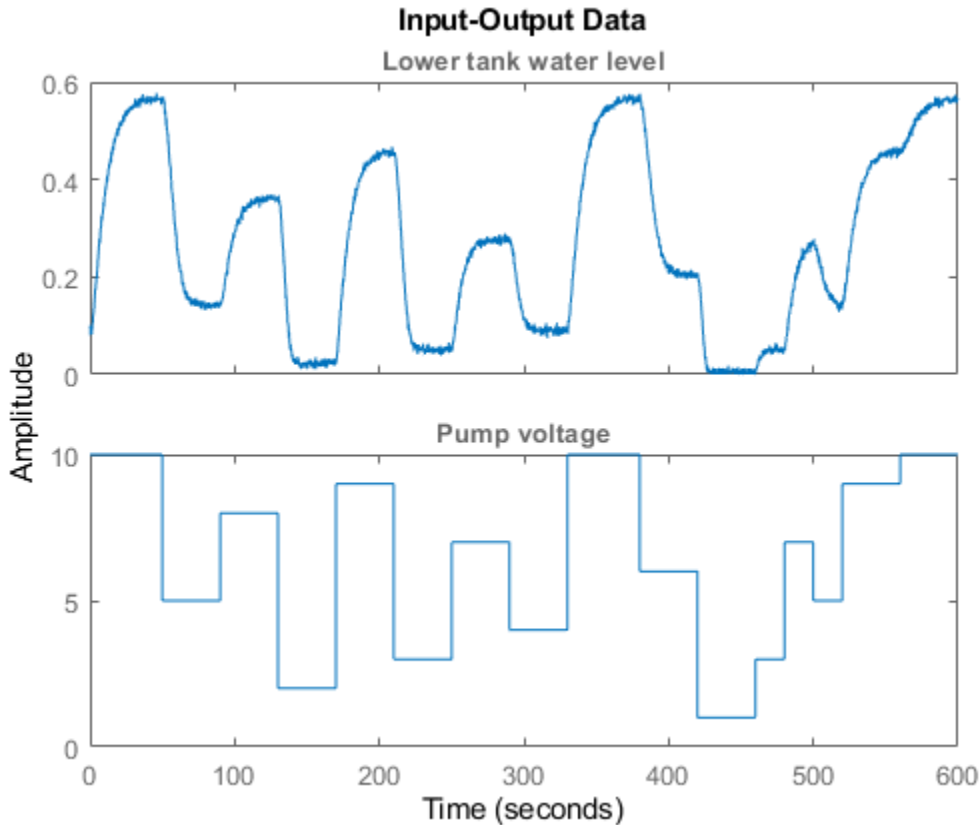

Figure 2: Input-output data from a two tank system.

Modeling the Two Tank System

The next step is to specify a model structure describing the two tank system. To do this, let x1(t) and x2(t) denote the water level in the upper and the lower tank, respectively. For each tank, fundamental physics (mass balance) states that the change of water volume depends on the difference between in- and outflow as (i = 1, 2):

```
  d/dt (Ai*xi(t)) = Qini(t) - Qouti(t)
```

where Ai [m^2] is the cross-sectional area of tank i and Qini(t) and Qouti(t) [m^3/s] are the inflow to and the outflow from tank i at time t.

For the upper tank, the inflow is assumed to be proportional to the voltage applied to the pump, i.e., Qin1(t) = k*u(t). Since the outlet hole of the upper tank is small, Bernoulli's law can be applied, stating that the outflow is proportional to the square root of the water level, or more precisely that:

```
  Qout1(t) = a1*sqrt(2*g*x1(t))
```

where a1 is the cross-sectional area of the outlet hole and g is the gravity constant. For the lower tank, the inflow equals the outflow from the upper tank, i.e., Qin2(t) = Qout1(t), and the outflow is given by Bernoulli's law:

```
Qout2(t) = a2*sqrt(2*g*x2(t))
```

where a2 is the cross-sectional area of the outlet hole.

Put altogether these facts lead to the following state-space structure:

```
d/dt x1(t) = 1/A1*(k*u(t) - a1*sqrt(2*g*x1(t)))
d/dt x2(t) = 1/A2*(a1*sqrt(2*g*x1(t)) - a2*sqrt(2*g*x2(t)))
      y(t) = x2(t)
```

Two Tank C MEX Model File

These equations are next put into a C MEX-file with 6 parameters (or constants), A1, k, a1, g, A2 and a2. The C MEX-file is normally a bit more involved than the corresponding file written using MATLAB language, but C MEX modeling generally gives a distinct advantage in terms of execution speed, especially for more complex models. A template C MEX-file is provided (see below) to help the user to structure the code. For most applications, it suffices to define the number of outputs and to enter the code lines that describe dx and y into this template. An IDNLGREY C MEX-file should always be structured to return two outputs:

```
dx: the right-hand side(s) of the state-space equation(s)
 y: the right-hand side(s) of the output equation(s)
```

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 our two tank system there are 6 scalar parameters and hence the number of input arguments to the C MEX modeling file should be 3+Npo = 3+6 = 9. The trailing 10:th argument can here be omitted as no optional FileArgument is employed in this application.

Writing a C MEX modeling file is normally done in four steps:

```
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. Writing 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.
```

```
Let us view the C MEX source file (except for some comments) for the two
tank system and based on this discuss these four items in some more
detail.
```

```
/* 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.
 *----------------------------------------------------------------*/
```
(1) (2) (3)

```
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 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);
}
```
(4)

Figure 3: C MEX source code for the two tank system.

1. Two C-libraries mex.h and math.h are normally included to provide access to a number of MEX-related as well as mathematical functions. 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
```

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.

The first step in these functions is to unpack the model parameters that will be used in the subsequent equations. Any valid variable name (except for those used in the input argument list) can be used to provide physically meaningful names of the individual parameters.

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.

The two tank model file involves square root computations. This is enabled through the inclusion of the mathematical C library math.h. The math library realizes the most common trigonometric functions (sin, cos, tan, asin, acos, atan, etc.), exponential (exp) and logarithms (log, log10), square root (sqrt) and power of functions (pow), and absolute value computations (fabs). The math.h library must be included whenever any math.h function is used; otherwise it can be omitted. See "Tutorials on Nonlinear Grey Box Model Identification: Creating IDNLGREY Model Files" for further details about the C math library.

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 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, like the number of states and the number of parameters.

Once the model source file has been completed it must be compiled, which can be done from the MATLAB command prompt using the mex command; see "help mex". (This step is omitted here.)

When developing model specific C MEX-files it is often useful to start the work by copying the IDNLGREY C MEX template file. 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 displayed by typing the following at the MATLAB command prompt.

```
fullfile(matlabroot, 'toolbox', 'ident', 'nlident', 'IDNLGREY_MODEL_TEMPLATE.c')
```

Also see "Creating IDNLGREY Model Files" example for more details on IDNLGREY C MEX model files.

Creating a Two Tank IDNLGREY Model Object

The next step is to create an IDNLGREY object describing the two tank system. For convenience we also set some bookkeeping information about the inputs and outputs (name and units).

```
FileName      = 'twotanks_c';           % File describing the model structure.
Order         = [1 1 2];                % Model orders [ny nu nx].
Parameters    = {0.5; 0.0035; 0.019; ...
                 9.81; 0.25; 0.016};    % Initial parameters.
InitialStates = [0; 0.1];               % Initial value of initial states.
Ts            = 0;                      % Time-continuous system.
nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts, ...
                'Name', 'Two tanks');
set(nlgr, 'InputName', 'Pump voltage', 'InputUnit', 'V',              ...
          'OutputName', 'Lower tank water level', 'OutputUnit', 'm', ...
          'TimeUnit', 's');
```

We continue to add information about the names and the units of the states and the model parameters via the commands SETINIT and SETPAR. Furthermore, both states x1(t) and x2(t) are tank levels that cannot be negative, and thus we also specify that x1(0) and x2(0) >= 0 via the 'Minimum' property. In fact, we also know that all model parameters ought to be strictly positive. We therefore set the 'Minimum' property of all parameters to some small positive value (eps(0)). These settings imply that constraint estimation will be carried out in the upcoming estimation step (i.e., the estimated model will be a model such that all entered constraints are honored).

```
nlgr = setinit(nlgr, 'Name', {'Upper tank water level' 'Lower tank water level'});
nlgr = setinit(nlgr, 'Unit', {'m' 'm'});
nlgr = setinit(nlgr, 'Minimum', {0 0});   % Positive levels!
nlgr = setpar(nlgr, 'Name', {'Upper tank area'        ...
                             'Pump constant'          ...
                             'Upper tank outlet area' ...
                             'Gravity constant'       ...
                             'Lower tank area'        ...
                             'Lower tank outlet area'});
nlgr = setpar(nlgr, 'Unit', {'m^2' 'm^3/(s*V)' 'm^2'  'm/(s^2)' 'm^2' 'm^2'});
nlgr = setpar(nlgr, 'Minimum', num2cell(eps(0)*ones(6,1)));   % All parameters > 0!
```

The cross-sectional areas (A1 and A2) of the two tanks can rather accurately be determined. We therefore treat these and g as constants and verify that the 'Fixed' field is properly set for all 6 parameters through the command GETPAR. All in all, this means that 3 of the model parameters will be estimated.

```
nlgr.Parameters(1).Fixed = true;
nlgr.Parameters(4).Fixed = true;
nlgr.Parameters(5).Fixed = true;
getpar(nlgr, 'Fixed')


ans =

  6x1 cell array

    {[1]}
    {[0]}
    {[0]}
    {[1]}
    {[1]}
    {[0]}
```

Performance of the Initial Two Tank Model

Before estimating the free parameters k, a1 and a2 we simulate the system using the initial parameter values. We use the default differential equation solver (a Runge-Kutta 45 solver with adaptive step length adjustment) and set the absolute and relative error tolerances to rather small values (1e-6 and 1e-5, respectively). Notice that the COMPARE command, when called with two input arguments, as default will estimate all initial state(s) regardless of whether any initial state has been defined to be 'Fixed'. In order to only estimate the free initial state(s), call COMPARE with a third and a fourth input argument as follows: compare(z, nlgr, 'init', 'm'); as both initial states of the tank model by default are 'Fixed', no initial state estimation will be performed by this command.

```
nlgr.SimulationOptions.AbsTol = 1e-6;
nlgr.SimulationOptions.RelTol = 1e-5;
```

```
compare(z, nlgr);
```
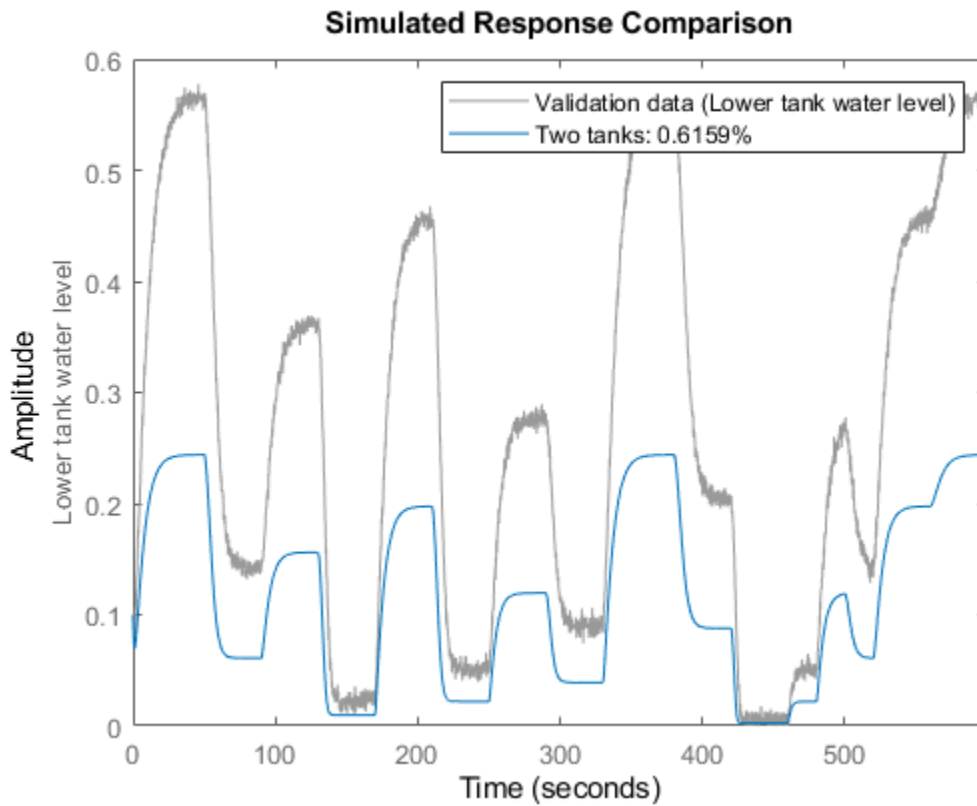

Figure 4: Comparison between true output and the simulated output of the initial two tank model.

The simulated and true outputs are shown in a plot window, and as can be seen the fit is not so impressive.

Parameter Estimation

In order to improve the fit, the 3 free parameters are next estimated using NLGREYEST. (Since, by default, the 'Fixed' fields of all initial states are false, no estimation of the initial states will be done in this call to the estimator.)

```
nlgr = nlgreyest(z, nlgr, nlgreyestOptions('Display', 'on'));
```

```
Nonlinear Grey Box Model Estimation
Data has 1 outputs, 1 inputs and 3000 samples.
ODE Function: twotanks_c
Number of parameters: 6
```

ESTIMATION PROGRESS

```
Algorithm: Trust-Region Reflective Newton
-----------------------------------------------------------------
                             Norm of       First-order
Iteration      Cost           step         optimality
-----------------------------------------------------------------
    0        0.0338423          -               -
    1        0.0032885        0.00426        6.32e+03
    2        0.000128102      0.00341           189
    3         2.5718e-05      0.00163           177
    4        2.46542e-05      0.00135          1.01
    5        2.43233e-05       0.0165          74.5
    6        2.41538e-05      0.000429        0.379
    7        2.41464e-05      0.00375          5.06
    8        2.41442e-05      0.00093         0.184
-----------------------------------------------------------------
```

RESULT

```
Termination condition: Change in cost was less than the specified tolerance..
Number of iterations: 8, Number of function evaluations: 9

Status: Estimated using NLGREYEST
Fit to estimation data: 97.35%, FPE: 2.41926e-05
```

■ Stop   Close

Performance of the Estimated Two Tank Model

To investigate the performance of the estimated model, a simulation of it is performed (the initial states are here reestimated).

```
compare(z, nlgr);
```
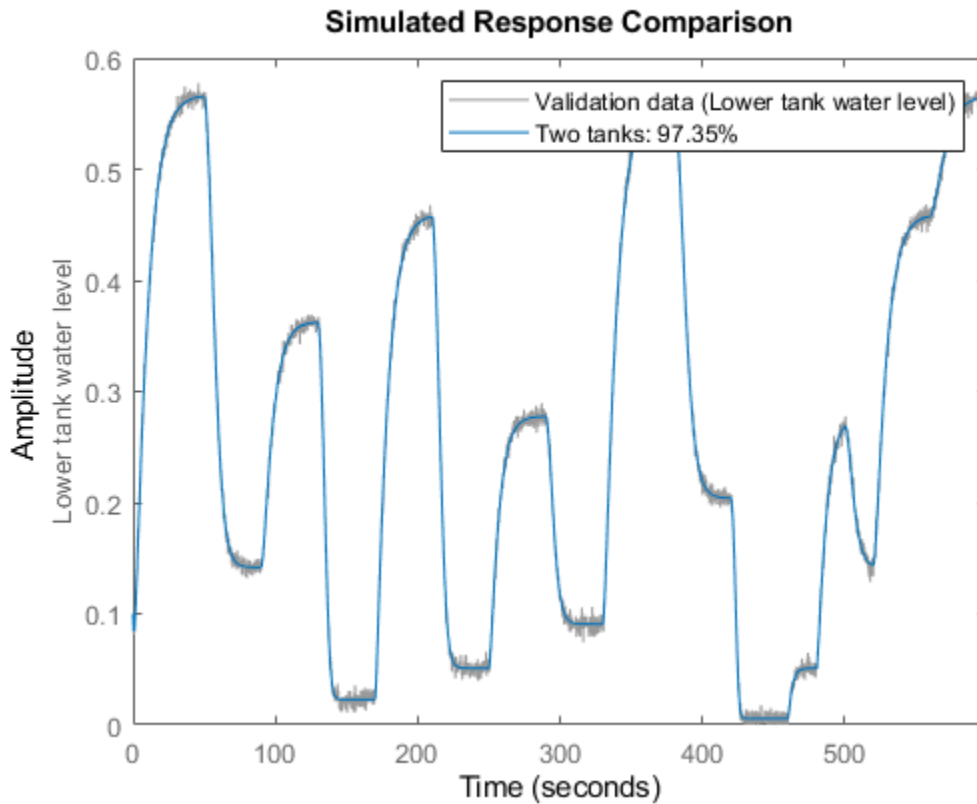
Figure 5: Comparison between true output and the simulated output of the estimated two tank model.

The agreement between the true and the simulated outputs is quite good. A remaining question is, however, if the two tank system can be accurately described using a simpler and linear model structure. To answer this, let us try to fit the data to some standard linear model structures, and then use COMPARE to see how well these models capture the dynamics of the tanks.

```
nk = delayest(z);
arx22 = arx(z, [2 2 nk]);    % Second order linear ARX model.
arx33 = arx(z, [3 3 nk]);    % Third order linear ARX model.
arx44 = arx(z, [4 4 nk]);    % Fourth order linear ARX model.
oe22  = oe(z,  [2 2 nk]);    % Second order linear OE model.
oe33  = oe(z,  [3 3 nk]);    % Third order linear OE model.
oe44  = oe(z,  [4 4 nk]);    % Fourth order linear OE model.
sslin = ssest(z);            % State-space model (order determined automatically)
compare(z, nlgr, 'b', arx22, 'm-', arx33, 'm:', arx44, 'm--', ...
        oe22, 'g-', oe33, 'g:', oe44, 'g--', sslin, 'r-');
```
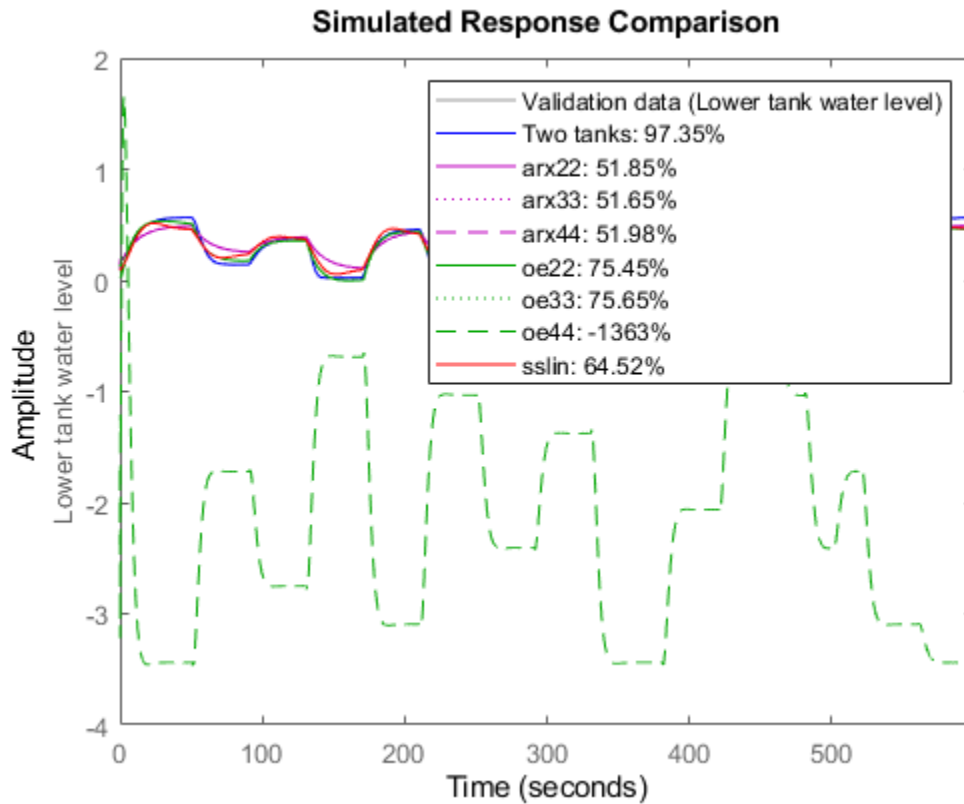
Figure 6: Comparison between true output and the simulated outputs of a number of estimated two tank models.

The comparison plot clearly reveals that the linear models cannot pick up all dynamics of the two tank system. The estimated nonlinear IDNLGREY model on the other hand shows an excellent fit to the true output. In addition, the IDNLGREY model parameters are also well in line with those used to generate the true output. In the following display computations, we are using the command GETPVEC, which returns a parameter vector created from the structure array holding the model parameters of an IDNLGREY object.

```
disp('   True      Estimated parameter vector');
ptrue = [0.5; 0.005; 0.02; 9.81; 0.25; 0.015];
fprintf('   %1.4f    %1.4f\n', [ptrue'; getpvec(nlgr)']);
```

```
   True      Estimated parameter vector
   0.5000    0.5000
   0.0050    0.0049
   0.0200    0.0200
   9.8100    9.8100
   0.2500    0.2500
   0.0150    0.0147
```

The prediction errors obtained using PE are small and look very much like random noise.

```
figure;
pe(z, nlgr);
```
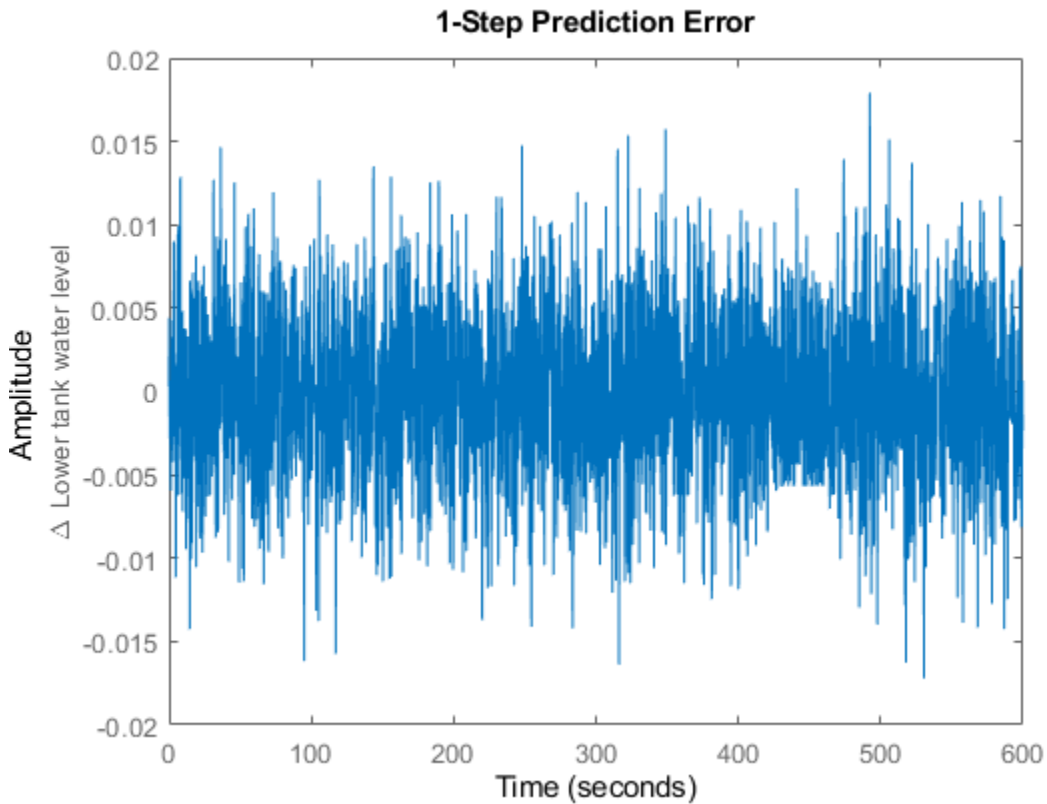
Figure 7: Prediction errors obtained for the estimated IDNLGREY two tank model.

Let us also investigate what happens if the input voltage is increased from 5 to 6, 7, 8, 9 and 10 V in a step-wise manner. We do this by calling STEP with different specified step amplitudes starting from a fixed offset of 5 Volts. The step response configuration is facilitated by a dedicated option-set created by `stepDataOptions`:

```
figure('Name', [nlgr.Name ': step responses']);
t = (-20:0.1:80)';
Opt = stepDataOptions('InputOffset',5,'StepAmplitude',6);
step(nlgr, t, 'b', Opt);
hold on

Opt.StepAmplitude = 7;  step(nlgr, t, 'g', Opt);
Opt.StepAmplitude = 8;  step(nlgr, t, 'r', Opt);
Opt.StepAmplitude = 9;  step(nlgr, t, 'm', Opt);
Opt.StepAmplitude = 10; step(nlgr, t, 'k', Opt);

grid on;
legend('5 -> 6 V', '5 -> 7 V', '5 -> 8 V', '5 -> 9 V', '5 -> 10 V', ...
       'Location', 'Best');
```
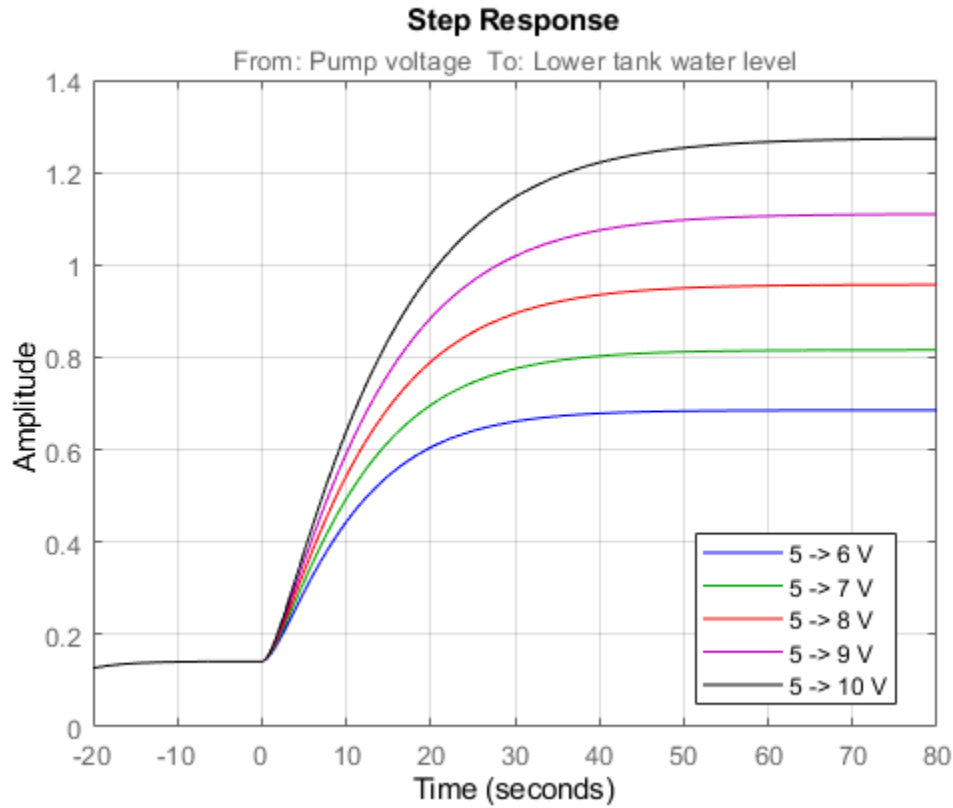
Figure 8: Step responses obtained for the estimated IDNLGREY two tank model.

By finally using the PRESENT command, we get summary information about the estimated model:

```
present(nlgr);
```

```
nlgr =

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(s), 2 state(s), 1 output(s), and 3 free parameter(s) (out of 6).

 Inputs:
    u(1)  Pump voltage(t) [V]
 States:                                  Initial value
    x(1)  Upper tank water level(t) [m]   xinit@exp1     0    (fixed) in [0, Inf]
    x(2)  Lower tank water level(t) [m]   xinit@exp1   0.1    (fixed) in [0, Inf]
 Outputs:
    y(1)  Lower tank water level(t) [m]
 Parameters:                              Value    Standard Deviation
    p1    Upper tank area [m^2]                0.5                   0    (fixed) in ]0, Inf]
    p2    Pump constant [m^3/(s*V)]     0.00488584           0.0259032    (estimated) in ]0, Inf]
    p3    Upper tank outlet area [m^2]   0.0199719           0.0064682    (estimated) in ]0, Inf]
    p4    Gravity constant [m/(s^2)]          9.81                   0    (fixed) in ]0, Inf]
```

```
      p5   Lower tank area [m^2]                    0.25              0   (fixed) in ]0, Inf]
      p6   Lower tank outlet area [m^2]        0.0146546      0.0776058   (estimated) in ]0, Inf]

Name: Two tanks

Status:
Termination condition: Change in cost was less than the specified tolerance..
Number of iterations: 8, Number of function evaluations: 9

Estimated using Solver: ode45; Search: lsqnonlin on time domain data "Two tanks".
Fit to estimation data: 97.35%
FPE: 2.419e-05, MSE: 2.414e-05
More information in model's "Report" property.

<a href="matlab:if exist('nlgr','var'), if isa(nlgr,'idnlgrey'), disp(' '); disp(get(nlgr)), else
```

Conclusions

In this example we have shown:

1. how to use C MEX-files for IDNLGREY modeling, and
2. provided a rather simple example where nonlinear state-space
   modeling shows good potential

Additional Information

For more information on identification of dynamic systems with System Identification Toolbox™ visit the System Identification Toolbox product information page.

Three Ecological Population Systems: MATLAB and C MEX-File Modeling of Time-Series

This example shows how to create nonlinear grey box time series models. Time series models are models not using any measured inputs. Three idealized ecological systems are studied where two species either:

- compete for the same food, or:
- are in a predator-prey situation

The example shows modeling based on both MATLAB and C MEX-files.

Ecological Population Systems

In all three population systems investigated we are interesting in how the population of two species vary with time. To model this, let x1(t) and x2(t) denote the number of individuals of the respective species at time t. Let l1 and l2 denote the birth-rate associated with x1(t) and x2(t), respectively, both assumed to be constants over time. The death-rates of the species depend both on the availability of food and, if predators are present, on the risk of being eaten. Quite often (and in general), the death-rate for species i (i = 1 or 2) can be written a ui(x1(t), x2(t)), where ui(.) is some appropriate function. In practice, this means that ui(x1(t), x2(t))*xi(t) animals of species i dies every time unit. The net effect of these statements can be summarized in a state space type of model structure: (a time-series):

d

-- x1(t) = l1*x1(t) - u1(x1(t), x2(t))*x1(t)

dt

d

-- x2(t) = l2*x2(t) - u2(x1(t), x2(t))*x2(t)

dt

It is here natural to choose the two states as outputs, i.e., we let y1(t) = x1(t) and y2(t) = x2(t).

A.1. Two Species that Compete for the Same Food

In case two species compete for the same food, then it is the overall population of the species that controls the availability of food, and in turn of their death-rates. A simple yet common approach is to assume that the death-rate can be written as:

ui(x1(t), x2(t)) = gi + di*(x1(t) + x2(t))

for both species (i = 1 or 2), where gi and di are unknown parameters. Altogether this gives the state space structure:

d

-- x1(t) = (l1-g1)*x1(t) - d1*(x1(t)+x2(t))*x1(t)

dt

d

-- x2(t) = (l2-g2)*x2(t) - d2*(x1(t)+x2(t))*x2(t)

dt

An immediate problem with this structure is that l1, g1, l2, and g2 cannot be identified separately. We can only hope to identify p1 = l1-g1 and p3 = l2-g2. By also letting p2 = d1 and p4 = d2, one gets the reparameterized model structure:

d

-- x1(t) = p1*x1(t) - p2*(x1(t)+x2(t))*x1(t)

dt

d

-- x2(t) = p3*x2(t) - p4*(x1(t)+x2(t))*x2(t)

dt

y1(t) = x1(t)

y2(t) = x2(t)

In this first population example we resort to MATLAB file modeling. The equations above are then entered into a MATLAB file, preys_m.m, with the following content.

function [dx, y] = preys_m(t, x, u, p1, p2, p3, p4, varargin)

%PREYS_M Two species that compete for the same food.

% Output equations.

y = [x(1); ... % Prey species 1.

x(2) ... % Prey species 2.

];

% State equations.

dx = [p1*x(1)-p2*(x(1)+x(2))*x(1); ... % Prey species 1.

p3*x(2)-p4*(x(1)+x(2))*x(2) ... % Prey species 2.

];

The MATLAB file, along with an initial parameter vector, an adequate initial state, and some administrative information are next fed as inputs to the IDNLGREY object constructor. Notice that the initial values of both the parameters and the initial states are specified as structure arrays with Npo (number of parameter objects = number of parameters if all parameters are scalars) and Nx (number of states) elements, respectively. Through these structure arrays it is possible to completely assign non-default property values (of 'Name', 'Unit', 'Value, 'Minimum', 'Maximum', and 'Fixed') to each parameter and initial state. Here we have assigned the 'Minimum' value of each initial state to zero (populations are positive!), and also specified that both initial states are to be estimated by default.

```
FileName     = 'preys_m';            % File describing the model structure.
Order        = [2 0 2];              % Model orders [ny nu nx].
Parameters   = struct('Name',    {'Survival factor, species 1' 'Death factor, species 1'   ...
    'Survival factor, species 2' 'Death factor, species 2'}, ...
    'Unit',    {'1/year' '1/year' '1/year' '1/year'},                    ...
    'Value',   {1.8 0.8 1.2 0.8},                                        ...
    'Minimum', {-Inf -Inf -Inf -Inf},                                    ...
    'Maximum', {Inf Inf Inf Inf},                                        ...
    'Fixed',   {false false false false});   % Estimate all 4 parameters.
InitialStates = struct('Name',    {'Population, species 1' 'Population, species 2'}, ...
    'Unit',    {'Size (in thousands)' 'Size (in thousands)'},    ...
    'Value',   {0.2 1.8},                                        ...
    'Minimum', {0 0},                                            ...
    'Maximum', {Inf Inf},                                        ...
    'Fixed',   {false false});   % Estimate both initial states.
Ts           = 0;                    % Time-continuous system.
nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts,          ...
    'Name', 'Two species competing for the same food',          ...
    'OutputName', {'Population, species 1' 'Population, species 2'}, ...
    'OutputUnit', {'Size (in thousands)' 'Size (in thousands)'},     ...
    'TimeUnit', 'year');
```

The PRESENT command can be used to view information about the initial model:

```
present(nlgr);
```

```
nlgr =

Continuous-time nonlinear grey-box model defined by 'preys_m' (MATLAB file):

   dx/dt = F(t, x(t), p1, ..., p4)
    y(t) = H(t, x(t), p1, ..., p4) + e(t)

 with 0 input(s), 2 state(s), 2 output(s), and 4 free parameter(s) (out of 4).

 States:                                      Initial value
    x(1)  Population, species 1(t) [Size (in t..]   xinit@exp1   0.2   (estimated) in [0, Inf]
    x(2)  Population, species 2(t) [Size (in t..]   xinit@exp1   1.8   (estimated) in [0, Inf]
 Outputs:
    y(1)  Population, species 1(t) [Size (in thousands)]
    y(2)  Population, species 2(t) [Size (in thousands)]
 Parameters:                                  Value
    p1   Survival factor, species 1 [1/year]      1.8      (estimated) in [-Inf, Inf]
    p2   Death factor, species 1 [1/year]         0.8      (estimated) in [-Inf, Inf]
    p3   Survival factor, species 2 [1/year]      1.2      (estimated) in [-Inf, Inf]
    p4   Death factor, species 2 [1/year]         0.8      (estimated) in [-Inf, Inf]

Name: Two species competing for the same food

Status:
Created by direct construction or transformation. Not estimated.
More information in model's "Report" property.

Model Properties
```

A.2. Input-Output Data

We next load (simulated, though noise corrupted) data and create an IDDATA object describing one particular situation where two species compete for the same food. This data set contains 201 data samples covering 20 years of evolution.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'preydata'));
z = iddata(y, [], 0.1, 'Name', 'Two species competing for the same food');
set(z, 'OutputName', {'Population, species 1', 'Population, species 2'}, ...
    'Tstart', 0, 'TimeUnit', 'Year');
```

A.3. Performance of the Initial Two Species Model

A simulation with the initial model clearly reveals that it cannot cope with the true population dynamics. See the plot figure. For a time-series type of IDNLGREY model notice that the model output is determined by the initial state.

```
compare(z, nlgr, 1);
```
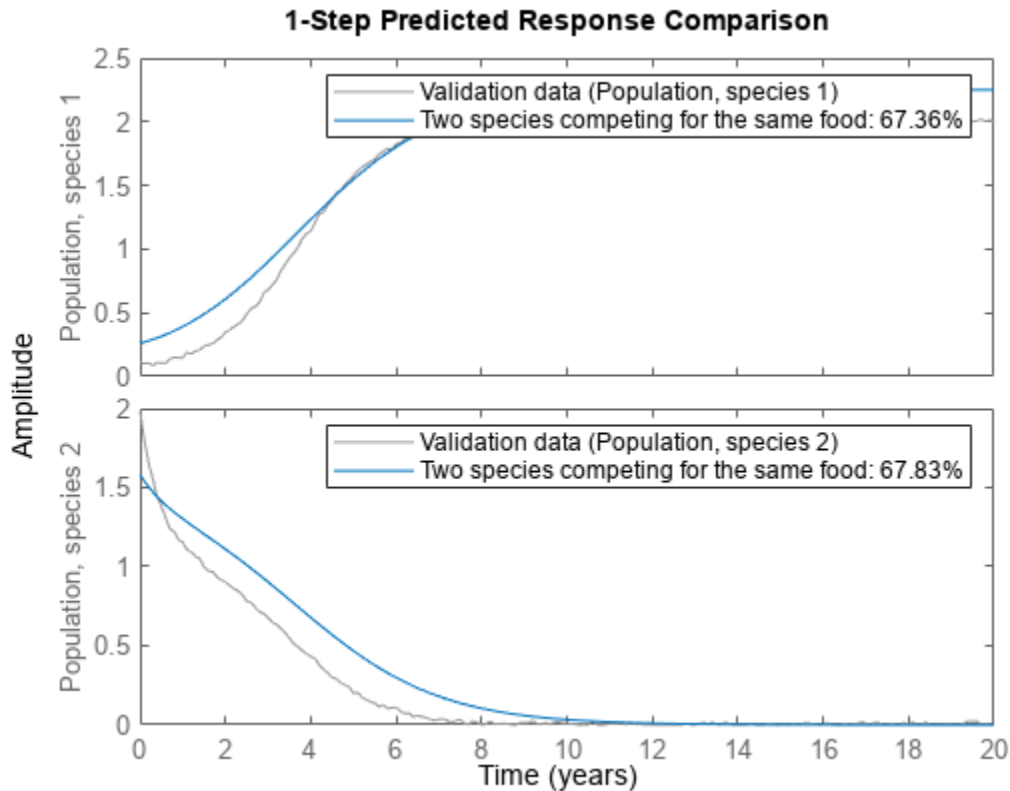

Figure 1: Comparison between true outputs and the simulated outputs of the initial two species model.

A.4. Parameter Estimation

In order to overcome the rather poor performance of the initial model we proceed to estimate the 4 unknown parameters and the 2 initial states using NLGREYEST. Specify estimation options using NLGREYESTOPTIONS; in this case 'Display' is set to 'on', which means that estimation progress

information is displayed in the progress window. You can use NLGREYESTOPTIONS to specify the basic algorithm properties such as 'GradientOptions', 'SearchMethod', 'MaxIterations', 'Tolerance', 'Display'.

```
opt = nlgreyestOptions;
opt.Display = 'on';
opt.SearchOptions.MaxIterations = 50;
nlgr = nlgreyest(z, nlgr, opt);
```

A.5. Performance of the Estimated Two Species Model

The estimated values of the parameters and the initial states are well in line with those used to generate the true output data:

```
disp('   True      Estimated parameter vector');
```

```
   True      Estimated parameter vector
```

```
ptrue = [2; 1; 1; 1];
fprintf('   %6.3f    %6.3f\n', [ptrue'; getpvec(nlgr)']);
```

```
    2.000     2.004
    1.000     1.002
    1.000     1.018
    1.000     1.010
```

```
disp(' ');
```

```
disp('   True      Estimated initial states');
```

```
   True      Estimated initial states
```

```
x0true = [0.1; 2];
fprintf('   %6.3f    %6.3f\n', [x0true'; cell2mat(getinit(nlgr, 'Value'))']);
```

```
    0.100     0.101
    2.000     1.989
```

To further evaluate the quality of the model (and to illustrate the improvement compared to the initial model) we also simulate the estimated model. The simulated outputs are compared to the true outputs in a plot window. As can be seen, the estimated model is quite good.
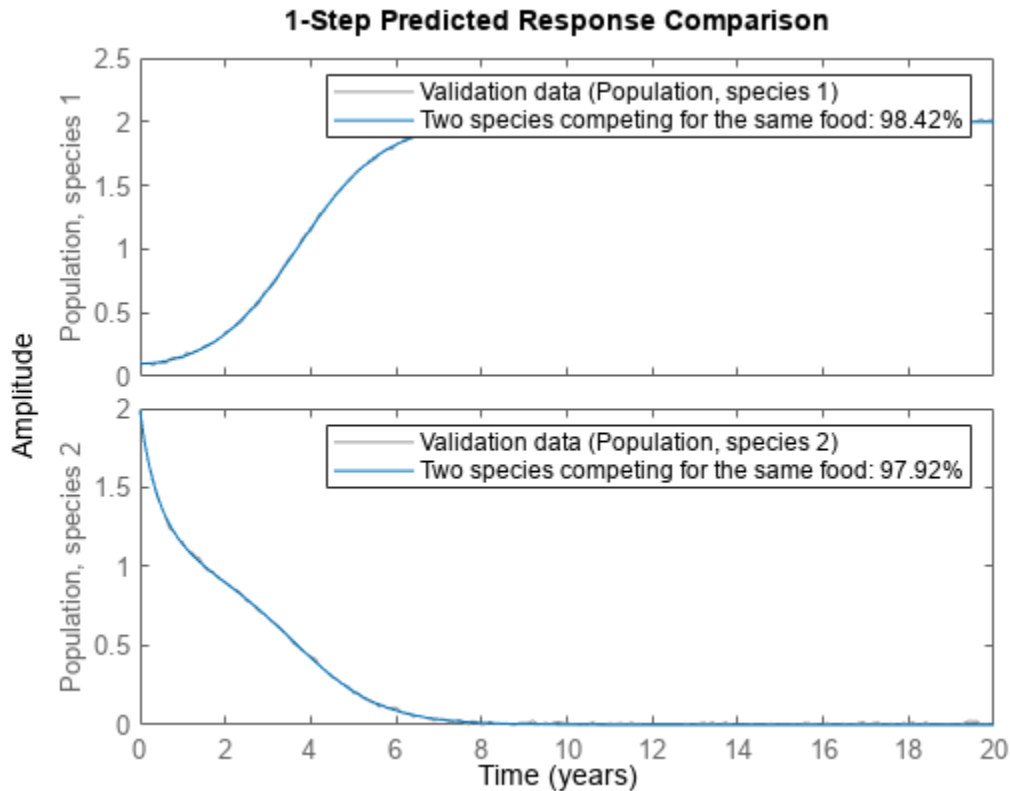
```
compare(z, nlgr, 1);
```

Figure 2: Comparison between true outputs and the simulated outputs of the estimated two species model.

PRESENT provides further information about the estimated model, e.g., about parameter uncertainties, and other estimation related quantities, like the loss function and Akaike's FPE (Final Prediction Error) measure.

```
present(nlgr);
```

```
nlgr =

Continuous-time nonlinear grey-box model defined by 'preys_m' (MATLAB file):

   dx/dt = F(t, x(t), p1, ..., p4)
    y(t) = H(t, x(t), p1, ..., p4) + e(t)

 with 0 input(s), 2 state(s), 2 output(s), and 4 free parameter(s) (out of 4).

 States:                                        Initial value
    x(1)  Population, species 1(t) [Size (in t..]    xinit@exp1   0.100729    (estimated) in [0, In
    x(2)  Population, species 2(t) [Size (in t..]    xinit@exp1   1.98855     (estimated) in [0, In
 Outputs:
    y(1)  Population, species 1(t) [Size (in thousands)]
    y(2)  Population, species 2(t) [Size (in thousands)]
 Parameters:                                    ValueStandard Deviation
    p1   Survival factor, species 1 [1/year]      2.00429      0.00971109    (estimated) in [-Inf,
```

```
    p2  Death factor, species 1 [1/year]        1.00235      0.00501783   (estimated) in [-Inf,
    p3  Survival factor, species 2 [1/year]     1.01779      0.0229598    (estimated) in [-Inf,
    p4  Death factor, species 2 [1/year]        1.0102       0.0163506    (estimated) in [-Inf,

Name: Two species competing for the same food

Status:
Termination condition: Near (local) minimum, (norm(g) < tol)..
Number of iterations: 5, Number of function evaluations: 6

Estimated using Solver: ode45; Search: lsqnonlin on time domain data "Two species competing for
Fit to estimation data: [98.42;97.92]%
FPE: 7.747e-09, MSE: 0.0001743
More information in model's "Report" property.

Model Properties
```

B.1. A Classical Predator-Prey System

Assume now that the first species lives on the second one. The availability of food for species 1 is then proportional to x2(t) (the number of individuals of species 2), which means that the death-rate of species 1 decreases when x2(t) increases. This fact is captured by the simple expression:

```
u1(x1(t), x2(t)) = g1 - a1*x2(t)
```

where g1 and a1 are unknown parameters. Similarly, the death-rate of species 2 will increase when the number of individuals of the first species increases, e.g., according to:

```
u2(x1(t), x2(t)) = g2 + a2*x1(t)
```

where g2 and a2 are two more unknown parameters. Using the linear birth-rate assumed above one gets the state space structure:

```
d

-- x1(t) = (l1-g1)*x1(t) + a1*x2(t)*x1(t)

dt

d

-- x2(t) = (l2-g2)*x2(t) - a2*x1(t)*x2(t)

dt
```

As in the previous population example, it is also here impossible to uniquely identify the six individual parameters. With the same kind of reparameterization as in the above case, i.e., p1 = l1-g1, p2 = a1, p3 = l2-g2, and p4 = a2, the following model structure is obtained:

```
d

-- x1(t) = p1*x1(t) + p2*x2(t)*x1(t)

dt

d
```

```
-- x2(t) = p3*x2(t) - p4*x1(t)*x2(t)

dt

y1(t) = x1(t)

y2(t) = x2(t)
```

which is better suited from an estimation point of view.

This time we enter this information into a C MEX-file named predprey1_c.c. The model file is structured as the standard IDNLGREY C MEX-file (see example titled "Creating IDNLGREY Model Files" or idnlgreydemo2.m), with the state and output update functions, compute_dx and compute_y, as follows.

```
void compute_dx(double *dx, double t, double *x, double **p,

const mxArray *auxvar)

{

/* Retrieve model parameters. */

double *p1, *p2, *p3, *p4;

p1 = p[0]; /* Survival factor, predators. */

p2 = p[1]; /* Death factor, predators. */

p3 = p[2]; /* Survival factor, preys. */

p4 = p[3]; /* Death factor, preys. */

/* x[0]: Predator species. */

/* x[1]: Prey species. */

dx[0] = p1[0]*x[0]+p2[0]*x[1]*x[0];

dx[1] = p3[0]*x[1]-p4[0]*x[0]*x[1];

}

/* Output equations. */

void compute_y(double *y, double t, double *x, double **p,

const mxArray *auxvar)

{

/* y[0]: Predator species. */

/* y[1]: Prey species. */

y[0] = x[0];
```

```
    y[1] = x[1];

    }
```

Since the model is of time-series type, neither compute_dx nor compute_y include u in the input argument list. In fact, the main interface function of predprey1_c.c, does not even declare u even though an empty u ([]) is always passed to predprey1_c by the IDNLGREY methods.

The compiled C MEX-file, along with an initial parameter vector, an adequate initial state, and some administrative information are next fed as input arguments to the IDNLGREY object constructor:

```
FileName     = 'predprey1_c';         % File describing the model structure.
Order        = [2 0 2];               % Model orders [ny nu nx].
Parameters   = struct('Name',    {'Survival factor, predators' 'Death factor, predators' ...
    'Survival factor, preys' 'Death factor, preys'},       ...
    'Unit',    {'1/year' '1/year' '1/year' '1/year'},                   ...
    'Value',   {-1.1 0.9 1.1 0.9},                                      ...
    'Minimum', {-Inf -Inf -Inf -Inf},                                  ...
    'Maximum', {Inf Inf Inf Inf},                                      ...
    'Fixed',   {false false false false});   % Estimate all 4 parameters.
InitialStates = struct('Name',     {'Population, predators' 'Population, preys'}, ...
    'Unit',    {'Size (in thousands)' 'Size (in thousands)'}, ...
    'Value',   {1.8 1.8},                                    ...
    'Minimum', {0 0},                                        ...
    'Maximum', {Inf Inf},                                    ...
    'Fixed',   {false false});   % Estimate both initial states.
Ts           = 0;                     % Time-continuous system.
nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts,          ...
    'Name', 'Classical 1 predator - 1 prey system',        ...
    'OutputName', {'Population, predators', 'Population, preys'}, ...
    'OutputUnit', {'Size (in thousands)' 'Size (in thousands)'},  ...
    'TimeUnit', 'year');
```

The predator prey model is next textually viewed via the PRESENT command.

```
present(nlgr);
```

```
nlgr =

Continuous-time nonlinear grey-box model defined by 'predprey1_c' (MEX-file):

   dx/dt = F(t, x(t), p1, ..., p4)
    y(t) = H(t, x(t), p1, ..., p4) + e(t)

 with 0 input(s), 2 state(s), 2 output(s), and 4 free parameter(s) (out of 4).

 States:                                    Initial value
    x(1)  Population, predators(t) [Size (in t..]    xinit@exp1   1.8   (estimated) in [0, Inf]
    x(2)  Population, preys(t) [Size (in t..]        xinit@exp1   1.8   (estimated) in [0, Inf]
 Outputs:
    y(1)  Population, predators(t) [Size (in thousands)]
    y(2)  Population, preys(t) [Size (in thousands)]
 Parameters:                                Value
    p1    Survival factor, predators [1/year]     -1.1      (estimated) in [-Inf, Inf]
    p2    Death factor, predators [1/year]         0.9      (estimated) in [-Inf, Inf]
    p3    Survival factor, preys [1/year]          1.1      (estimated) in [-Inf, Inf]
    p4    Death factor, preys [1/year]             0.9      (estimated) in [-Inf, Inf]
```

```
Name: Classical 1 predator - 1 prey system

Status:
Created by direct construction or transformation. Not estimated.
More information in model's "Report" property.

Model Properties
```

B.2. Input-Output Data

Our next step is to load (simulated, though noise corrupted) data and create an IDDATA object describing this particular predator-prey situation. This data set also contains 201 data samples covering 20 years of evolution.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'predprey1data'));
z = iddata(y, [], 0.1, 'Name', 'Classical 1 predator - 1 prey system');
set(z, 'OutputName', {'Population, predators', 'Population, preys'}, ...
    'Tstart', 0, 'TimeUnit', 'Year');
```

B.3. Performance of the Initial Classical Predator-Prey Model

A simulation with the initial model indicates that it cannot accurately cope with the true population dynamics. See the plot window.

```
compare(z, nlgr, 1);
```
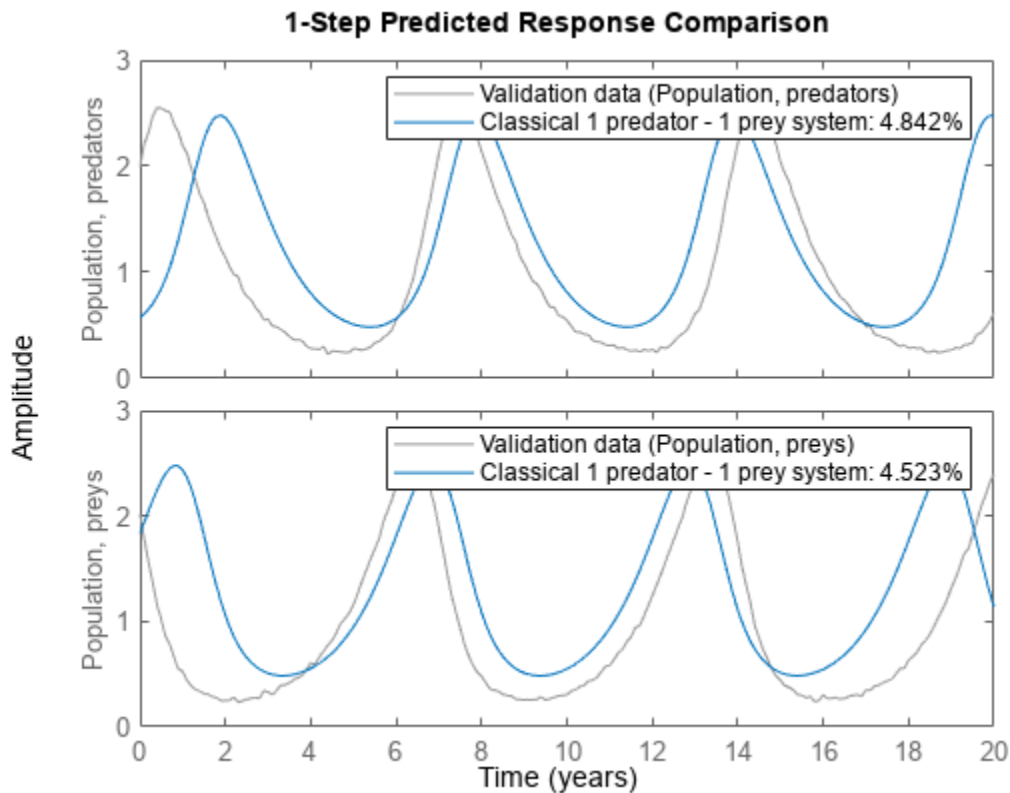
Figure 3: Comparison between true outputs and the simulated outputs of the initial classical predator-prey model.

B.4. Parameter Estimation

To improve the performance of the initial model we continue to estimate the 4 unknown parameters and the 2 initial states using NLGREYEST.

```
nlgr = nlgreyest(z, nlgr, nlgreyestOptions('Display', 'on'));
```

B.5. Performance of the Estimated Classical Predator-Prey Model

The estimated values of the parameters and initial states are very close to those that were used to generate the true output data:

```
disp('   True      Estimated parameter vector');
```
```
   True      Estimated parameter vector
```
```
ptrue = [-1; 1; 1; 1];
fprintf('   %6.3f    %6.3f\n', [ptrue'; getpvec(nlgr)']);
```
```
   -1.000    -1.000
    1.000     1.000
    1.000     1.000
    1.000     0.999
```
```
disp(' ');
```

```
disp('   True      Estimated initial states');
```
```
   True      Estimated initial states
```
```
x0true = [2; 2];
fprintf('   %6.3f    %6.3f\n', [x0true'; cell2mat(getinit(nlgr, 'Value'))']);
```
```
    2.000     2.002
    2.000     2.000
```

To further evaluate the model's quality (and to illustrate the improvement compared to the initial model) we also simulate the estimated model. The simulated outputs are compared to the true outputs in a plot window. As can be seen again, the estimated model is quite good.

```
compare(z, nlgr, 1);
```
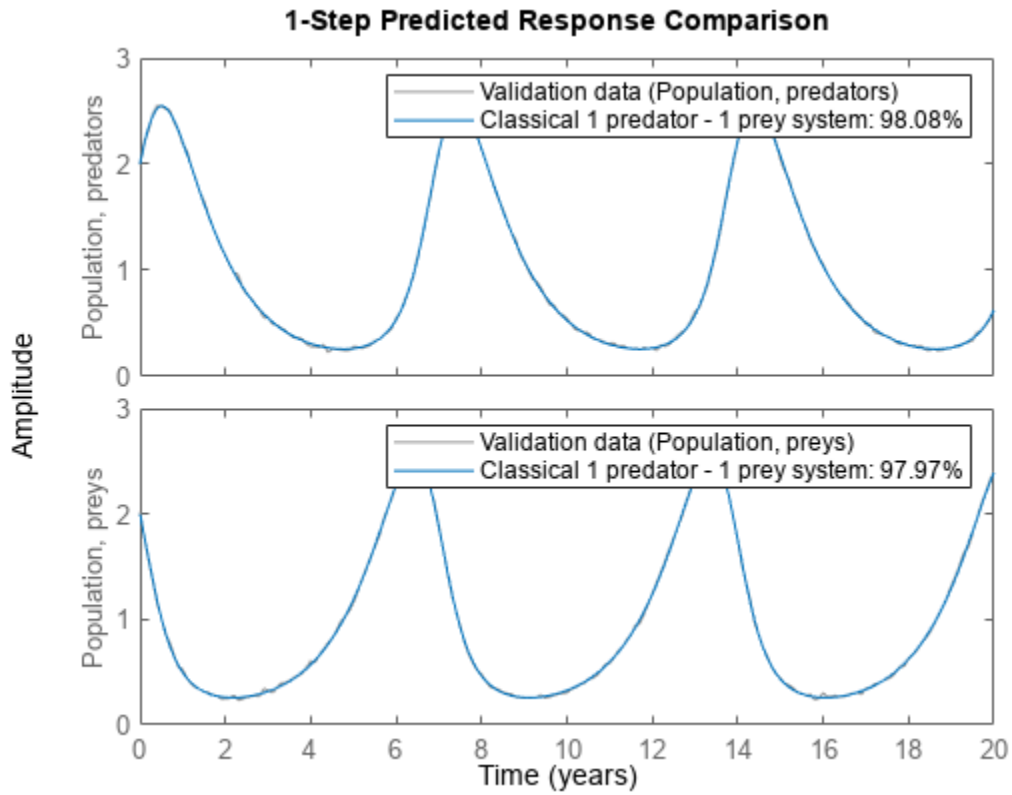
Figure 4: Comparison between true outputs and the simulated outputs of the estimated classical predator-prey model.

As expected, the prediction errors returned by PE are small and of a random nature.

```
pe(z, nlgr);
```
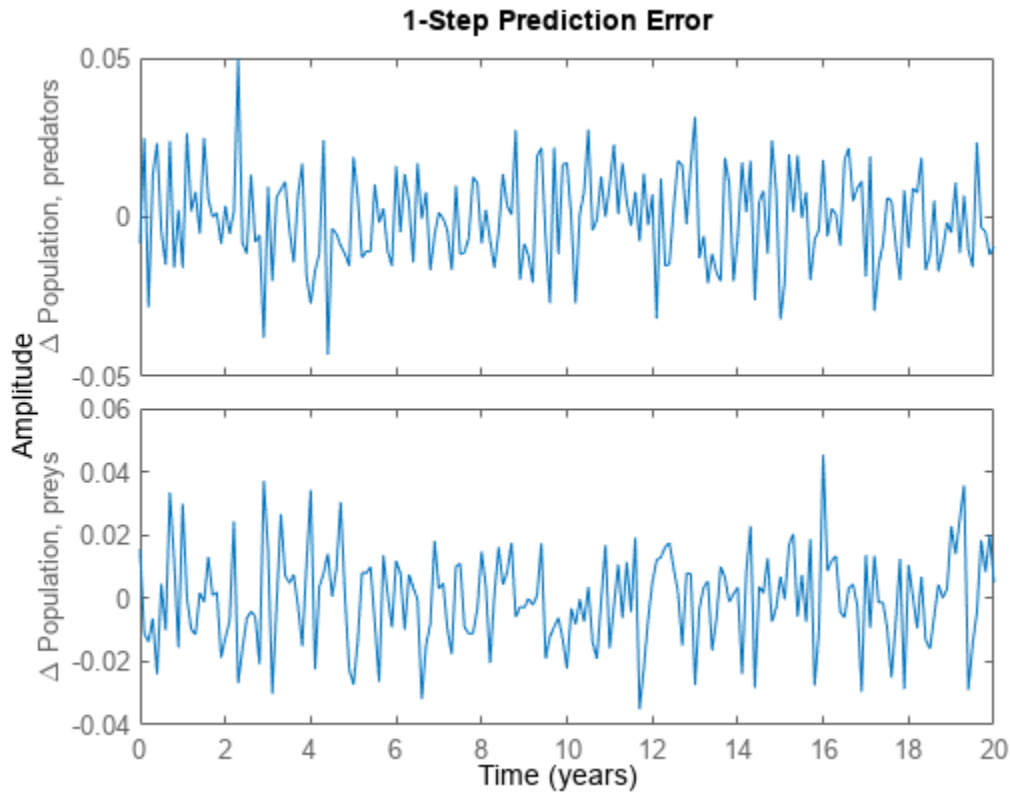
Figure 5: Prediction errors obtained with the estimated IDNLGREY classical predator-prey model.

C.1. A Predator-Prey System with Prey Crowding

The last population study is also devoted to a 1 predator and 1 prey system, with the difference being that we here introduce a term -p5*x2(t)^2 representing retardation of prey growth due to crowding. The reparameterized model structure from the previous example is then just complemented with this crowding term:

d

-- x1(t) = p1*x1(t) + p2*x2(t)*x1(t)

dt

d

-- x2(t) = p3*x2(t) - p4*x1(t)*x2(t) - p5*x2(t)^2

dt

y1(t) = x1(t)

y2(t) = x2(t)

The interpretation of these equations is essentially the same as above, except that in the absence of predators the growth of the prey population will be kept at bay.

The new modeling situation is reflected by a C MEX-file called predprey2_c.c, which is almost the same as predprey1_c.c. Aside from changes related to the number of model parameters (5 instead of 4), the main difference lies in the state update function compute_dx:

```c
/* State equations. */

void compute_dx(double *dx, double t, double *x, double **p,

const mxArray *auxvar)

{
/* Retrieve model parameters. */

double *p1, *p2, *p3, *p4, *p5;

p1 = p[0]; /* Survival factor, predators. */

p2 = p[1]; /* Death factor, predators. */

p3 = p[2]; /* Survival factor, preys. */

p4 = p[3]; /* Death factor, preys. */

p5 = p[4]; /* Crowding factor, preys. */

/* x[0]: Predator species. */

/* x[1]: Prey species. */

dx[0] = p1[0]*x[0]+p2[0]*x[1]*x[0];

dx[1] = p3[0]*x[1]-p4[0]*x[0]*x[1]-p5[0]*pow(x[1],2);

}
```

Notice that the added retardation term is computed as p5[0]*pow(x[1],2). The power of function pow(., .) is defined in the C library math.h, which must be included at the top of the model file through the statement #include "math.h" (this is not necessary in predprey1_c.c as it only holds standard C mathematics).

The compiled C MEX-file, along with an initial parameter vector, an adequate initial state, and some administrative information are next fed as input arguments to the IDNLGREY object constructor:

```matlab
FileName      = 'predprey2_c';           % File describing the model structure.
Order         = [2 0 2];                 % Model orders [ny nu nx].
Parameters    = struct('Name',    {'Survival factor, predators' 'Death factor, predators' ...
   'Survival factor, preys' 'Death factor, preys'         ...
   'Crowding factor, preys'},                             ...
   'Unit',    {'1/year' '1/year' '1/year' '1/year' '1/year'},        ...
   'Value',   {-1.1 0.9 1.1 0.9 0.2},                     ...
   'Minimum', {-Inf -Inf -Inf -Inf -Inf},                ...
   'Maximum', {Inf Inf Inf Inf Inf},                     ...
   'Fixed',   {false false false false false});   % Estimate all 5 parameters.
InitialStates = struct('Name',    {'Population, predators' 'Population, preys'}, ...
   'Unit',    {'Size (in thousands)' 'Size (in thousands)'}, ...
   'Value',   {1.8 1.8},                                  ...
```

```
        'Minimum', {0 0},                                         ...
        'Maximum', {Inf Inf},                                     ...
        'Fixed',   {false false});   % Estimate both initial states.
Ts            = 0;                        % Time-continuous system.
nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts,          ...
        'Name', '1 predator - 1 prey system exhibiting crowding',     ...
        'OutputName', {'Population, predators', 'Population, preys'}, ...
        'OutputUnit', {'Size (in thousands)' 'Size (in thousands)'},  ...
        'TimeUnit', 'year');
```

By typing the name of the model object (nlgr) basic information about the model is displayed in the command window. Note that, as before, present(nlgr) provides a more comprehensive summary of the model.

```
nlgr

nlgr =

Continuous-time nonlinear grey-box model defined by 'predprey2_c' (MEX-file):

    dx/dt = F(t, x(t), p1, ..., p5)
     y(t) = H(t, x(t), p1, ..., p5) + e(t)

 with 0 input(s), 2 state(s), 2 output(s), and 5 free parameter(s) (out of 5).

Name: 1 predator - 1 prey system exhibiting crowding

Status:
Created by direct construction or transformation. Not estimated.
```

C.2. Input-Output Data

Next we load (simulated, though noise corrupted) data and create an IDDATA object describing this crowding type of predator-prey situation. This data set contains 201 data samples covering 20 years of evolution.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'predprey2data'));
z = iddata(y, [], 0.1, 'Name', '1 predator - 1 prey system exhibiting crowding');
set(z, 'OutputName', {'Population, predators', 'Population, preys'}, ...
    'Tstart', 0, 'TimeUnit', 'Year');
```

C.3. Performance of the Initial Predator-Prey Model with Prey Crowding

A simulation with the initial model clearly shows that it cannot cope with the true population dynamics. See the figure.

```
clf
compare(z, nlgr, 1);
```
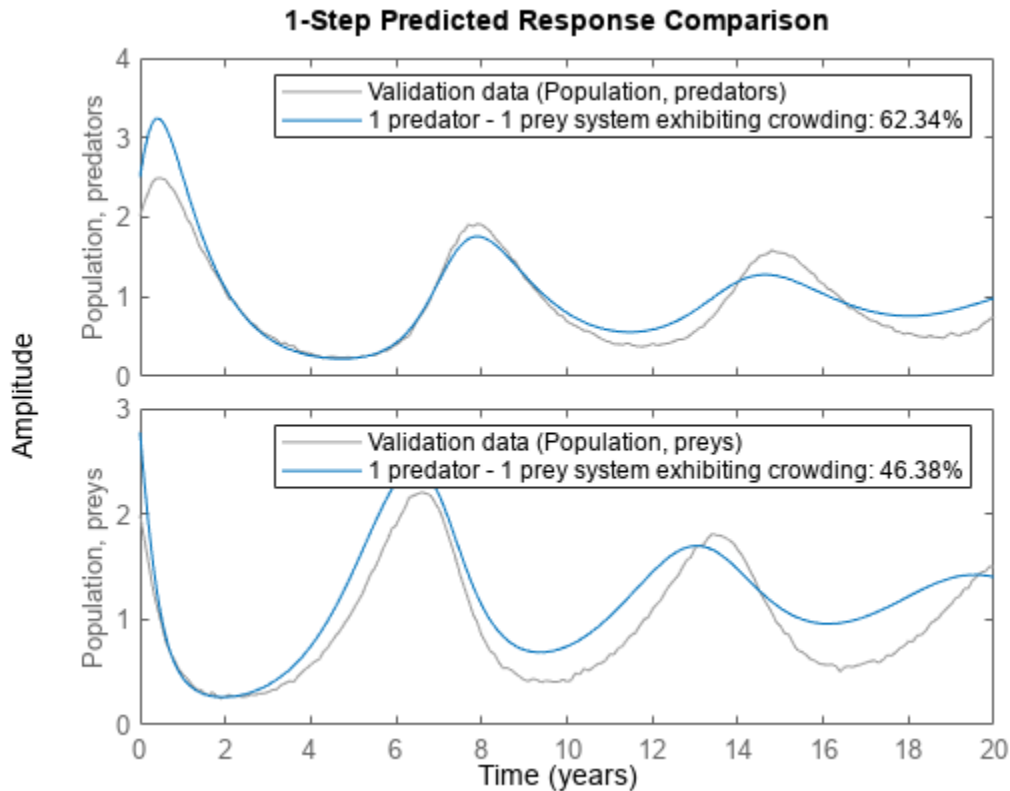
Figure 6: Comparison between true outputs and the simulated outputs of the initial predator-prey model with prey crowding.

C.4. Parameter Estimation

To improve the performance of the initial model we proceed to estimate the 5 unknown parameters and the 2 initial states.

```
nlgr = nlgreyest(z, nlgr, nlgreyestOptions('Display', 'on'));
```

C.5. Performance of the Estimated Predator-Prey Model with Prey Crowding

The estimated values of the parameters and the initial states are again quite close to those that were used to generate the true output data:

```
disp('   True      Estimated parameter vector');
```

```
   True      Estimated parameter vector
```

```
ptrue = [-1; 1; 1; 1; 0.1];
fprintf('   %6.3f    %6.3f\n', [ptrue'; getpvec(nlgr)']);
```

```
   -1.000    -1.000
    1.000     1.001
    1.000     1.002
    1.000     1.002
    0.100     0.101
```

```
disp(' ');
```

```
disp('   True      Estimated initial states');
```

```
   True     Estimated initial states
```

```
x0true = [2; 2];
fprintf('   %6.3f    %6.3f\n', [x0true'; cell2mat(getinit(nlgr, 'Value'))']);
```

```
   2.000    2.003
   2.000    2.002
```

To further evaluate the quality of the model (and to illustrate the improvement compared to the initial model) we also simulate the estimated model. The simulated outputs are compared to the true outputs in a plot window. As can be seen again, the estimated model is quite good.

```
compare(z, nlgr, 1);
```
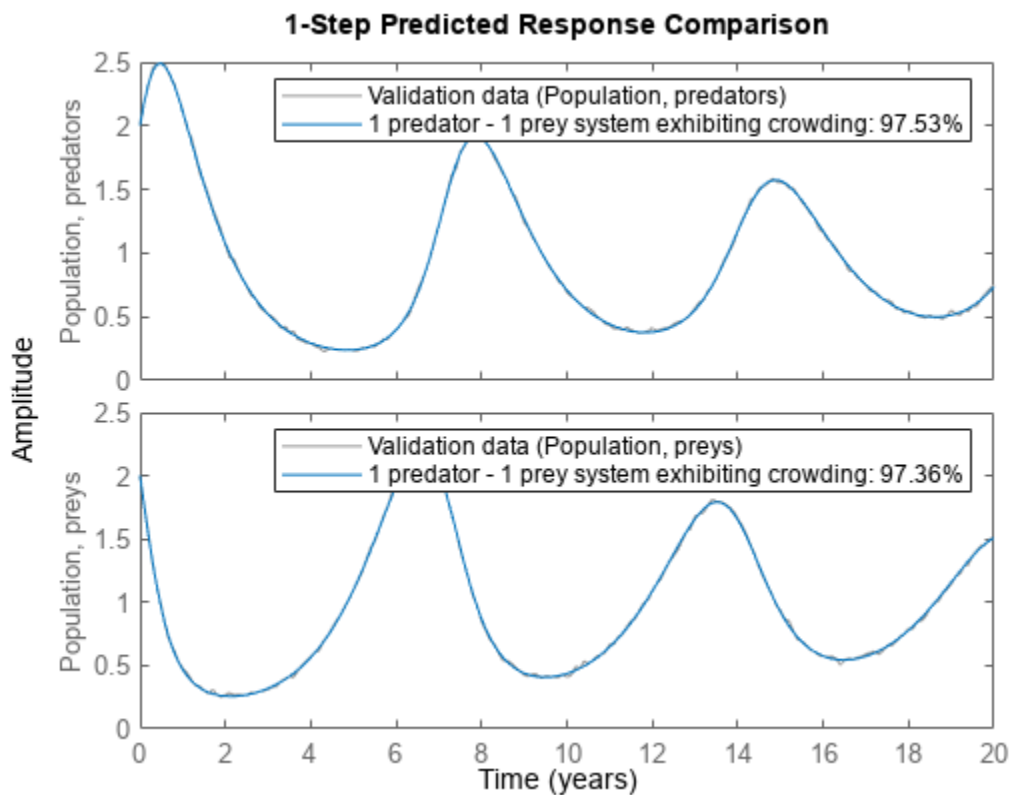

Figure 7: Comparison between true outputs and the simulated outputs of the estimated predator-prey model with prey crowding.

We conclude the third population example by presenting the model information returned by PRESENT.

```
present(nlgr);
```

```
nlgr =
```

```
Continuous-time nonlinear grey-box model defined by 'predprey2_c' (MEX-file):

   dx/dt = F(t, x(t), p1, ..., p5)
    y(t) = H(t, x(t), p1, ..., p5) + e(t)

 with 0 input(s), 2 state(s), 2 output(s), and 5 free parameter(s) (out of 5).

 States:                                        Initial value
    x(1)  Population, predators(t) [Size (in t..]   xinit@exp1   2.00281   (estimated) in [0, In
    x(2)  Population, preys(t) [Size (in t..]       xinit@exp1   2.00224   (estimated) in [0, In
 Outputs:
    y(1)  Population, predators(t) [Size (in thousands)]
    y(2)  Population, preys(t) [Size (in thousands)]
 Parameters:                                    Value   Standard Deviation
    p1   Survival factor, predators [1/year]    -0.999914     0.00280581   (estimated) in [-In
    p2   Death factor, predators [1/year]        1.00058      0.00276684   (estimated) in [-In
    p3   Survival factor, preys [1/year]         1.0019       0.00272154   (estimated) in [-In
    p4   Death factor, preys [1/year]            1.00224      0.00268423   (estimated) in [-In
    p5   Crowding factor, preys [1/year]         0.101331     0.0005023    (estimated) in [-In

Name: 1 predator - 1 prey system exhibiting crowding

Status:
Termination condition: Change in cost was less than the specified tolerance..
Number of iterations: 8, Number of function evaluations: 9

Estimated using Solver: ode45; Search: lsqnonlin on time domain data "1 predator - 1 prey system
Fit to estimation data: [97.53;97.36]%
FPE: 4.327e-08, MSE: 0.0004023
More information in model's "Report" property.

Model Properties
```

Conclusions

This example showed how to perform IDNLGREY time-series modeling based on MATLAB and MEX
model files.

Narendra-Li Benchmark System: Nonlinear Grey Box Modeling of a Discrete-Time System

This example shows how to identify the parameters of a complex yet artificial nonlinear discrete-time system with one input and one output. The system was originally proposed and discussed by Narendra and Li in the article

```
K. S. Narendra and S.-M. Li. "Neural networks in control systems". In
Mathematical Perspectives on Neural Networks (P. Smolensky,
M. C. Mozer, and D. E. Rumelhard, eds.), Chapter 11, pages 347-394,
Lawrence Erlbaum Associates, Hillsdale, NJ, USA, 1996.
```

and has been considered in numerous discrete-time identification examples.

Discrete-Time Narendra-Li Benchmark System

The discrete-time equations of the Narendra-Li system are:

```
x1(t+Ts) = (x1(t)/(1+x1(t)^2)+p(1))*sin(x2(t))
x2(t+Ts) = x2(t)*cos(x2(t)) + x1(t)*exp(-(x1(t)^2+x2(t)^2)/p(2))
           + u(t)^3/(1+u(t)^2+p(3)*cos(x1(t)+x2(t)))

   y(t) = x1(t)/(1+p(4)*sin(x2(t))+p(5)*sin(x1(t)))
```

where x1(t) and x2(t) are the states, u(t) the input signal, y(t) the output signal and p a parameter vector with 5 elements.

IDNLGREY Discrete-Time Narendra-Li Model

From an IDNLGREY model file point of view, there is no difference between a continuous- and a discrete-time system. The Narendra-Li model file must therefore return the state update vector dx and the output y, and should take t (time), x (state vector), u (input), p (parameter(s)) and varargin as input arguments:

```
function [dx, y] = narendrali_m(t, x, u, p, varargin)
%NARENDRALI_M  A discrete-time Narendra-Li benchmark system.

% Output equation.
y = x(1)/(1+p(4)*sin(x(2)))+x(2)/(1+p(5)*sin(x(1)));

% State equations.
dx = [(x(1)/(1+x(1)^2)+p(1))*sin(x(2));                ... % State 1.
      x(2)*cos(x(2))+x(1)*exp(-(x(1)^2+x(2)^2)/p(2)) ... % State 2.
         + u(1)^3/(1+u(1)^2+p(3)*cos(x(1)+x(2)))     ...
     ];
```

Notice that we have here chosen to condense all 5 parameters into one parameter vector, where the i:th element is referenced in the usual MATLAB way, i.e., through p(i).

With this model file as a basis, we next create an IDNLGREY object reflecting the modeling situation. Worth stressing here is that the discreteness of the model is specified through a positive value of Ts (= 1 second). Notice also that Parameters only holds one element, a 5-by-1 parameter vector, and that SETPAR are used to specify that all the components of this vector are strictly positive.

```
FileName      = 'narendrali_m';                       % File describing the model structure.
Order         = [1 1 2];                              % Model orders [ny nu nx].
```

```
Parameters    = {[1.05; 7.00; 0.52; 0.52; 0.48]};   % True initial parameters (a vector).
InitialStates = zeros(2, 1);                         % Initial value of initial states.
Ts            = 1;                                    % Time-discrete system with Ts = 1s.
nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts, 'Name', ...
                'Discrete-time Narendra-Li benchmark system',  ...
                'TimeUnit', 's');
nlgr = setpar(nlgr, 'Minimum', {eps(0)*ones(5, 1)});
```

A summary of the entered Narendra-Li model structure is obtained through the PRESENT command:

```
present(nlgr);
```

```
nlgr =

Discrete-time nonlinear grey-box model defined by 'narendrali_m' (MATLAB file):

   x(t+Ts) = F(t, u(t), x(t), p1)
      y(t) = H(t, u(t), x(t), p1) + e(t)

 with 1 input(s), 2 state(s), 1 output(s), and 5 free parameter(s) (out of 5).

 Inputs:
    u(1)  (t)
 States:         Initial value
    x(1)  x1(t)   xinit@exp1   0   (fixed) in [-Inf, Inf]
    x(2)  x2(t)   xinit@exp1   0   (fixed) in [-Inf, Inf]
 Outputs:
    y(1)  (t)
 Parameters:     Value
   p1(1)   p1    1.05      (estimated) in ]0, Inf]
   p1(2)          7        (estimated) in ]0, Inf]
   p1(3)         0.52      (estimated) in ]0, Inf]
   p1(4)         0.52      (estimated) in ]0, Inf]
   p1(5)         0.48      (estimated) in ]0, Inf]

Name: Discrete-time Narendra-Li benchmark system
Sample time: 1 seconds

Status:
Created by direct construction or transformation. Not estimated.
More information in model's "Report" property.

<a href="matlab:if exist('nlgr','var'), if isa(nlgr,'idnlgrey'), disp(' '); disp(get(nlgr)), else
```

Input-Output Data

Two input-output data records with 300 samples each, one for estimation and one for validation purposes, are available. The input vector used for both these records is the same, and was chosen as a sum of two sinusoids:

```
u(t) = sin(2*pi*t/10) + sin(2*pi*t/25)
```

for t = 0, 1, ..., 299 seconds. We create two different IDDATA objects to hold the two data records, ze for estimation and zv for validation purposes.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'narendralidata'));
ze = iddata(y1, u, Ts, 'Tstart', 0, 'Name', 'Narendra-Li estimation data', ...
```

```
                'ExperimentName', 'ze');
zv = iddata(y2, u, Ts, 'Tstart', 0, 'Name', 'Narendra-Li validation data', ...
                'ExperimentName', 'zv');
```

The input-output data that will be used for estimation are shown in a plot window.

```
figure('Name', ze.Name);
plot(ze);
```
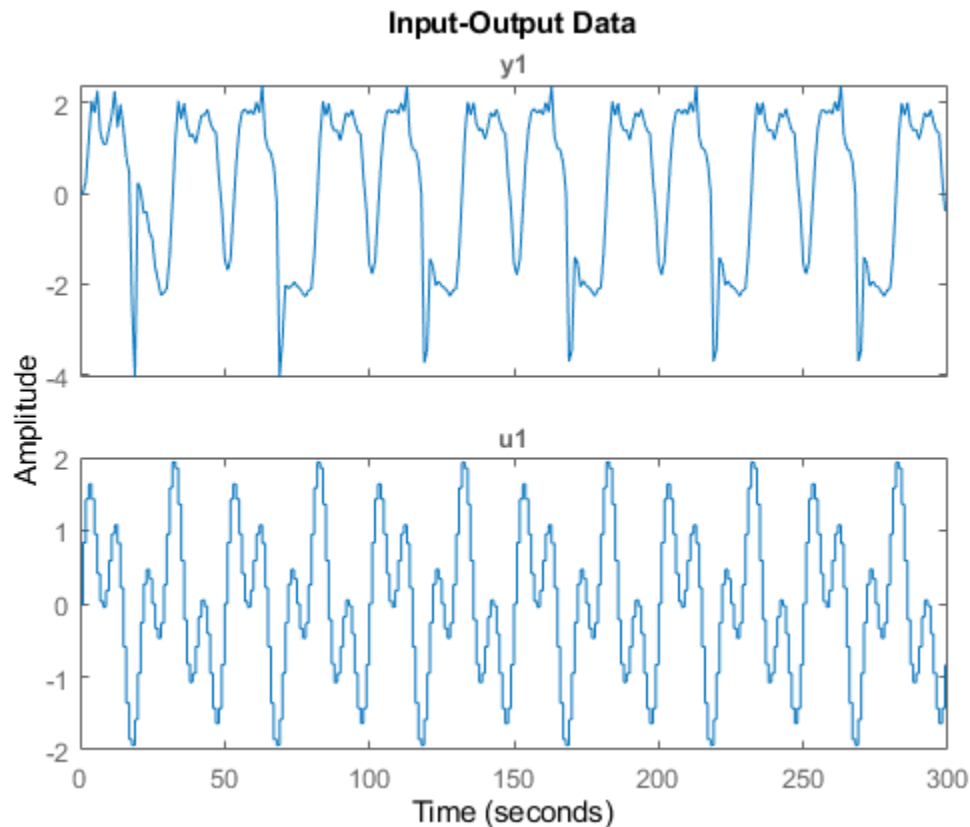

Figure 1: Input-output data from a Narendra-Li benchmark system.

Performance of the Initial Narendra-Li Model

Let us now use COMPARE to investigate the performance of the initial Narendra-Li model. The simulated and measured outputs for ze and zv are generated, and fit to ze is shown by default. Use the plot context menu to switch the data experiment to zv. The fit is not that bad for either dataset (around 74 %). Here we should point out that COMPARE by default estimates the initial state vector, in this case one initial state vector for ze and another one for zv.

```
compare(merge(ze, zv), nlgr);
```
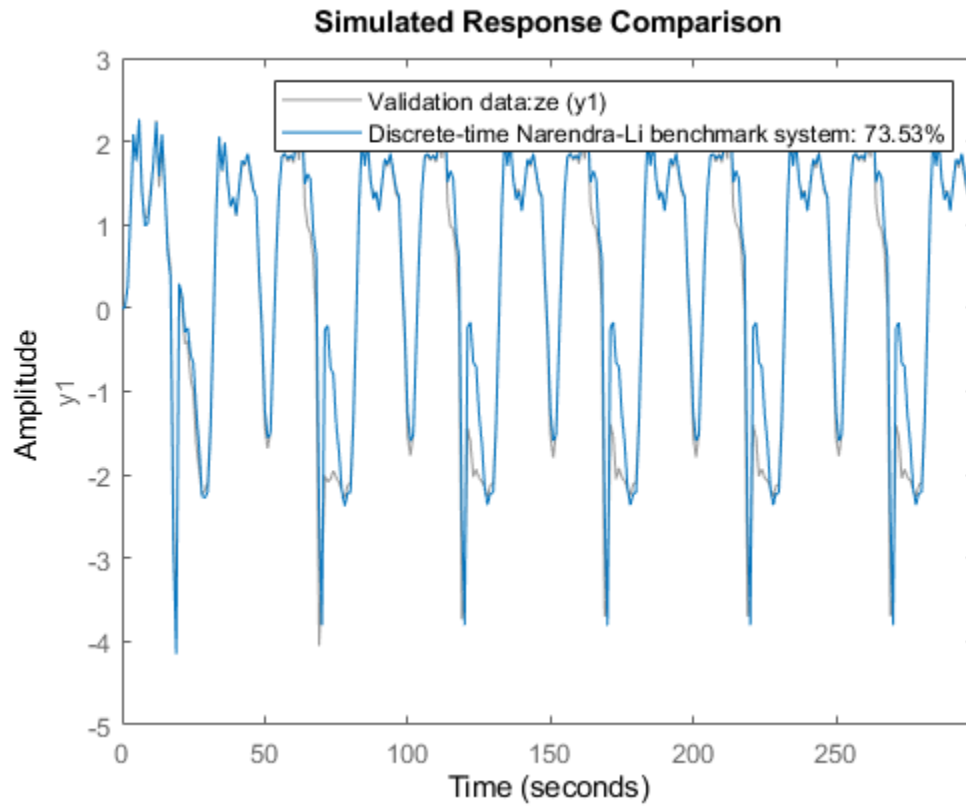
Figure 2: Comparison between true outputs and the simulated outputs of the initial Narendra-Li model.

```
%%4
% By looking at the prediction errors obtained via PE, we realize that the
% initial Narendra-Li model shows some systematic and periodic differences
% as compared to the true outputs.
pe(merge(ze, zv), nlgr);
```
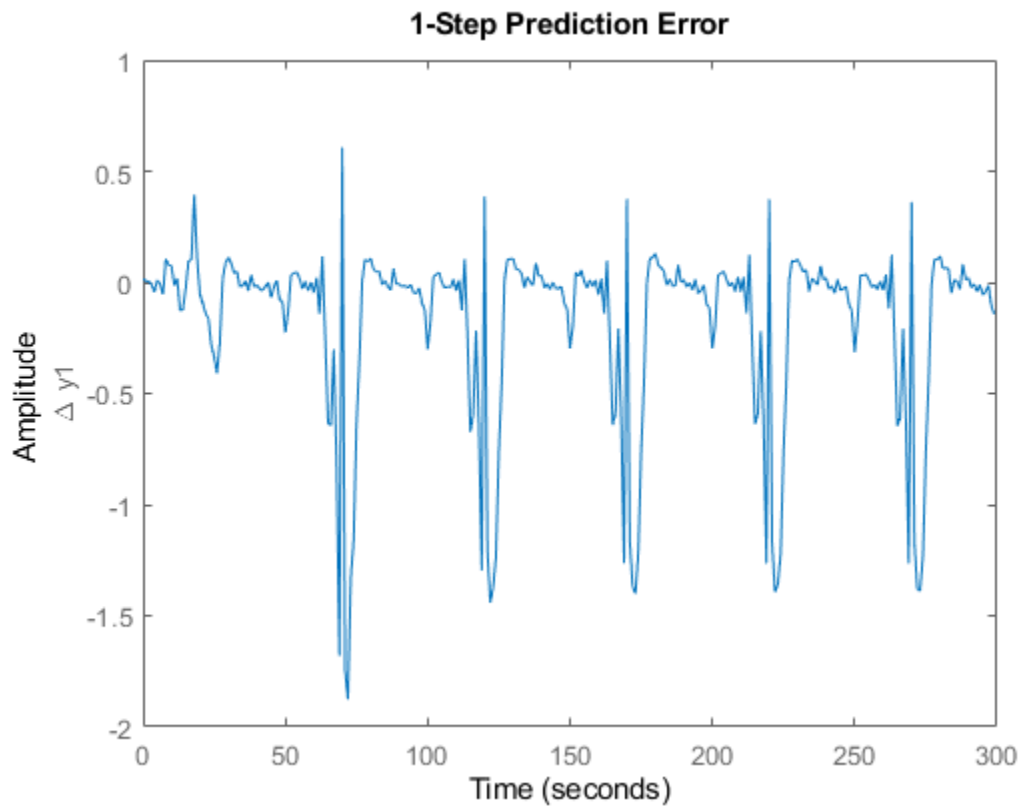
Figure 3: Prediction errors obtained with the initial Narendra-Li model. Use context menu to switch between experiments.

Parameter Estimation

In order to reduce the systematic discrepancies we estimate all 5 parameters of the Narendra-Li model structure using NLGREYEST. The estimation is carried out based on the estimation data set ze only.

```
opt = nlgreyestOptions('Display', 'on');
nlgr = nlgreyest(ze, nlgr, opt);
```

```
Nonlinear Grey Box Model Estimation
Data has 1 outputs, 1 inputs and 300 samples.
ODE Function: narendrali_m
Number of parameters: 5
```

ESTIMATION PROGRESS

```
Algorithm: Trust-Region Reflective Newton
-----------------------------------------------------------------
                              Norm of      First-order
Iteration      Cost            step         optimality
-----------------------------------------------------------------
     0        0.197618           -               -
     1        0.134505         0.332            168
     2        0.0830092        0.122            377
     3        0.0717264        0.85             821
     4        0.0492759        0.178          1.42e+03
     5        0.00218368       0.153            317
     6        0.000139495      0.0468           21.9
     7        9.42586e-05      0.0231           10.3
     8        8.99462e-05      0.000876          3.38
     9        8.93444e-05      0.00161             2
```

RESULT

```
Termination condition: Change in parameters was less than the specified tolerance..
Number of iterations: 16, Number of function evaluations: 17

Status: Estimated using NLGREYEST
Fit to estimation data: 99.44%, FPE: 9.22965e-05
```

■ Stop   Close

Performance of the Estimated Narendra-Li Model

COMPARE is once again employed to investigate the performance of the estimated model, but this time we use the model's internally stored initial state vector ([0; 0] for both experiments) without any initial state vector estimation.

```
clf
opt = compareOptions('InitialCondition','m');
compare(merge(ze, zv), nlgr, opt);
```
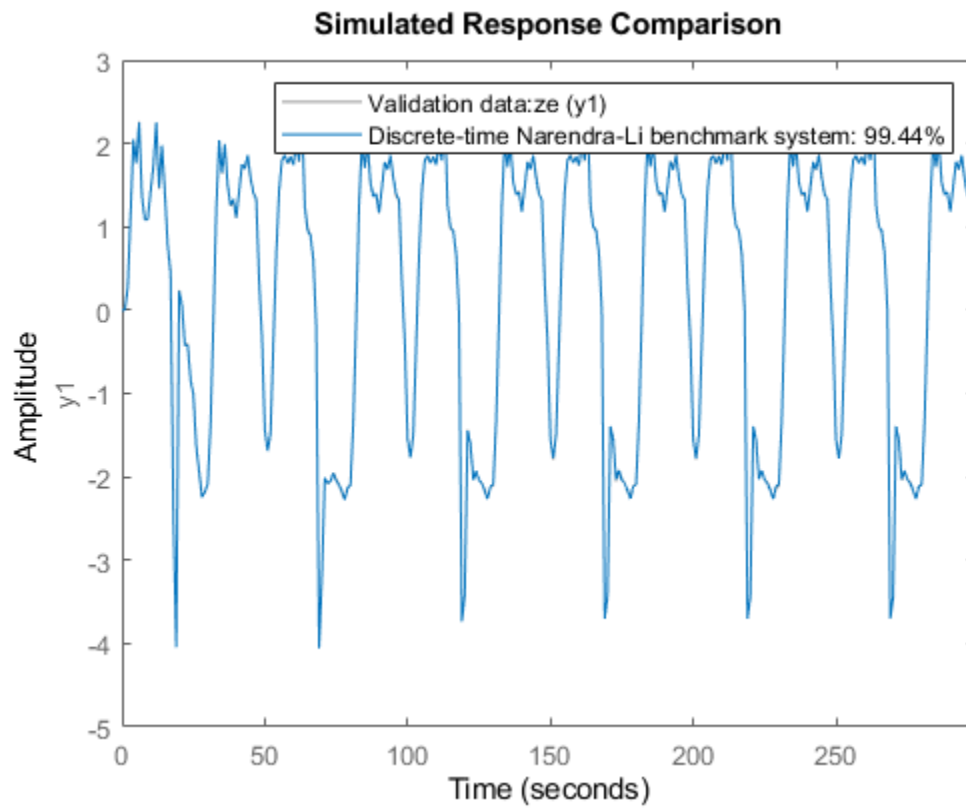
Figure 4: Comparison between true outputs and the simulated outputs of the estimated Narendra-Li model.

The improvement after estimation is significant, which perhaps is best illustrated through a plot of the prediction errors:

```
figure;
pe(merge(ze, zv), nlgr);
```
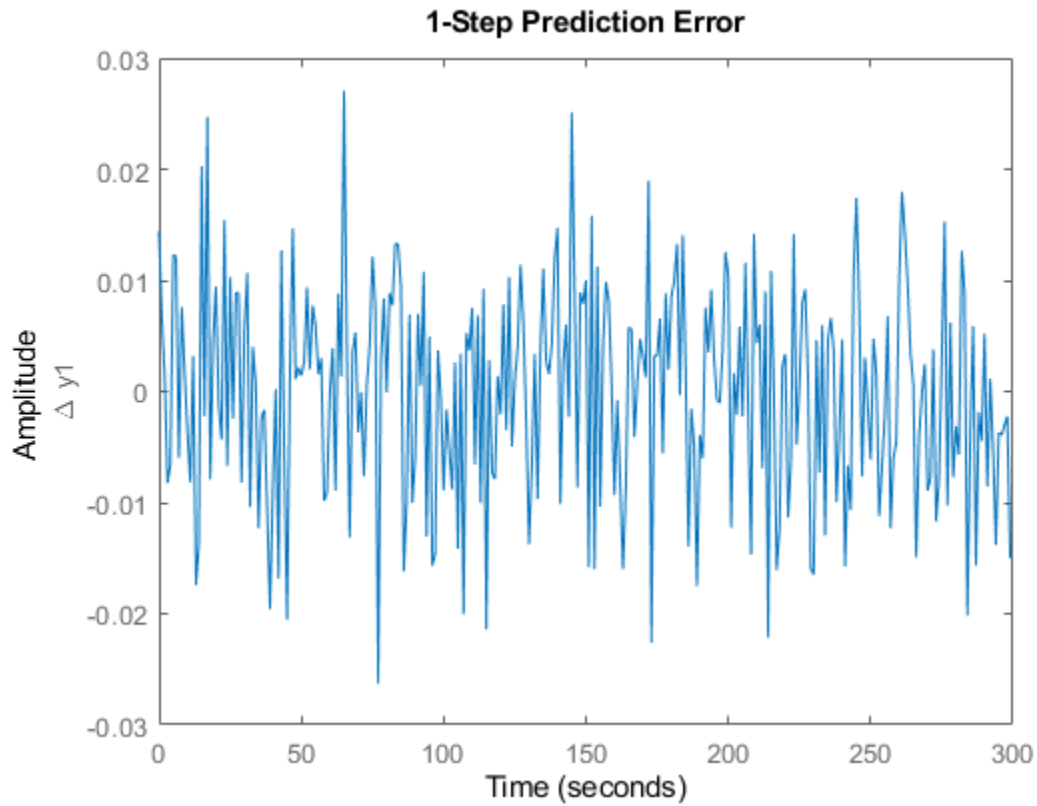
Figure 5: Prediction errors obtained with the estimated Narendra-Li model.

The modeling power of the estimated Narendra-Li model is also confirmed through the correlation analysis provided through RESID:

```
figure('Name', [nlgr.Name ': residuals of estimated model']);
resid(zv, nlgr);
```
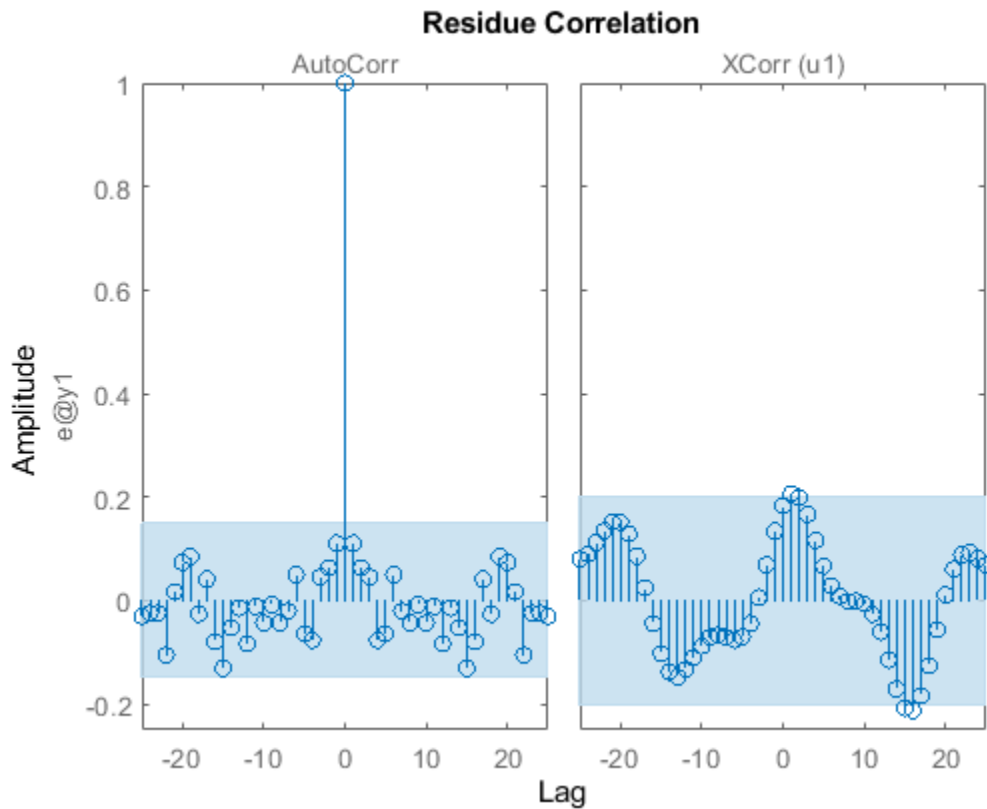
Figure 6: Residuals obtained with the estimated Narendra-Li model.

In fact, the obtained model parameters are also quite close to the ones that were used to generate the true model outputs:

```
disp('   True       Estimated parameter vector');
ptrue = [1; 8; 0.5; 0.5; 0.5];
fprintf('   %1.4f    %1.4f\n', [ptrue'; getpvec(nlgr)']);
```

```
   True       Estimated parameter vector
   1.0000    1.0000
   8.0000    8.0082
   0.5000    0.5003
   0.5000    0.4988
   0.5000    0.5018
```

Some additional model quality results (loss function, Akaike's FPE, and the estimated standard deviations of the model parameters) are next provided via the PRESENT command:

```
present(nlgr);
```

```
nlgr =

Discrete-time nonlinear grey-box model defined by 'narendrali_m' (MATLAB file):

    x(t+Ts) = F(t, u(t), x(t), p1)
       y(t) = H(t, u(t), x(t), p1) + e(t)
```

```
  with 1 input(s), 2 state(s), 1 output(s), and 5 free parameter(s) (out of 5).

  Inputs:
     u(1)  u1(t)
  States:         Initial value
     x(1)  x1(t)   xinit@exp1   0   (fixed) in [-Inf, Inf]
     x(2)  x2(t)   xinit@exp1   0   (fixed) in [-Inf, Inf]
  Outputs:
     y(1)  y1(t)
  Parameters:        Value Standard Deviation
    p1(1)   p1     0.999993        0.0117615   (estimated) in ]0, Inf]
    p1(2)          8.00821         0.885988    (estimated) in ]0, Inf]
    p1(3)          0.500319        0.0313092   (estimated) in ]0, Inf]
    p1(4)          0.498842        0.264806    (estimated) in ]0, Inf]
    p1(5)          0.501761        0.30998     (estimated) in ]0, Inf]

Name: Discrete-time Narendra-Li benchmark system
Sample time: 1 seconds

Status:
Termination condition: Change in parameters was less than the specified tolerance..
Number of iterations: 16, Number of function evaluations: 17

Estimated using Solver: FixedStepDiscrete; Search: lsqnonlin on time domain data "Narendra-Li est
Fit to estimation data: 99.44%
FPE: 9.23e-05, MSE: 8.927e-05
More information in model's "Report" property.

<a href="matlab:if exist('nlgr','var'), if isa(nlgr,'idnlgrey'), disp(' '); disp(get(nlgr)), else
```

As for continuous-time input-output systems, it is also possible to use STEP for discrete-time input-output systems. Let us do this for two different step levels, 1 and 2:

```
figure('Name', [nlgr.Name ': step responses of estimated model']);
t = (-5:50)';
step(nlgr, 'b', t, stepDataOptions('StepAmplitude',1));
line(t, step(nlgr, t, stepDataOptions('StepAmplitude',2)), 'color', 'g');
grid on;
legend('0 -> 1', '0 -> 2', 'Location', 'NorthWest');
```
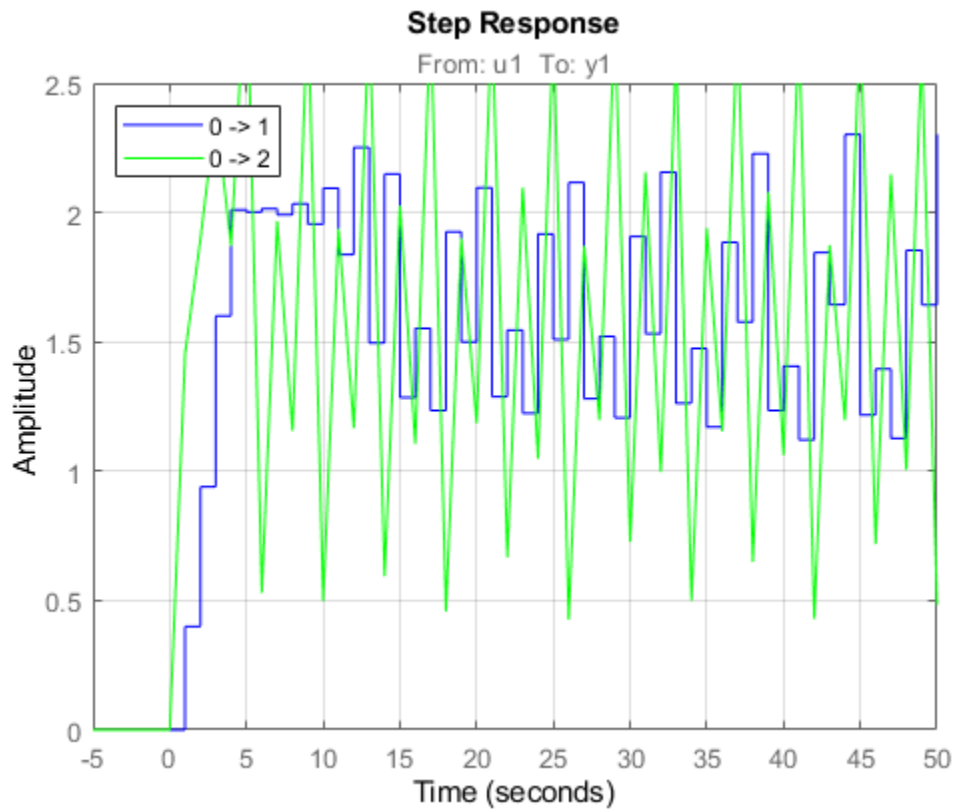
Figure 7: Step responses obtained with the estimated Narendra-Li model.

We conclude this example by comparing the performance of the estimated IDNLGREY Narendra-Li model with some basic linear models. The fit obtained for the latter models are considerably lower than what is returned for the estimated IDNLGREY model.

```
nk = delayest(ze);
arx22 = arx(ze, [2 2 nk]);    % Second order linear ARX model.
arx33 = arx(ze, [3 3 nk]);    % Third order linear ARX model.
arx44 = arx(ze, [4 4 nk]);    % Fourth order linear ARX model.
oe22  = oe(ze, [2 2 nk]);     % Second order linear OE model.
oe33  = oe(ze, [3 3 nk]);     % Third order linear OE model.
oe44  = oe(ze, [4 4 nk]);     % Fourth order linear OE model.
clf
fig = gcf;
Pos = fig.Position; fig.Position = [Pos(1:2)*0.7, Pos(3)*1.3, Pos(4)*1.3];
compare(zv, nlgr, 'b', arx22, 'm-', arx33, 'm:', arx44, 'm--', ...
        oe22, 'g-', oe33, 'g:', oe44, 'g--');
```
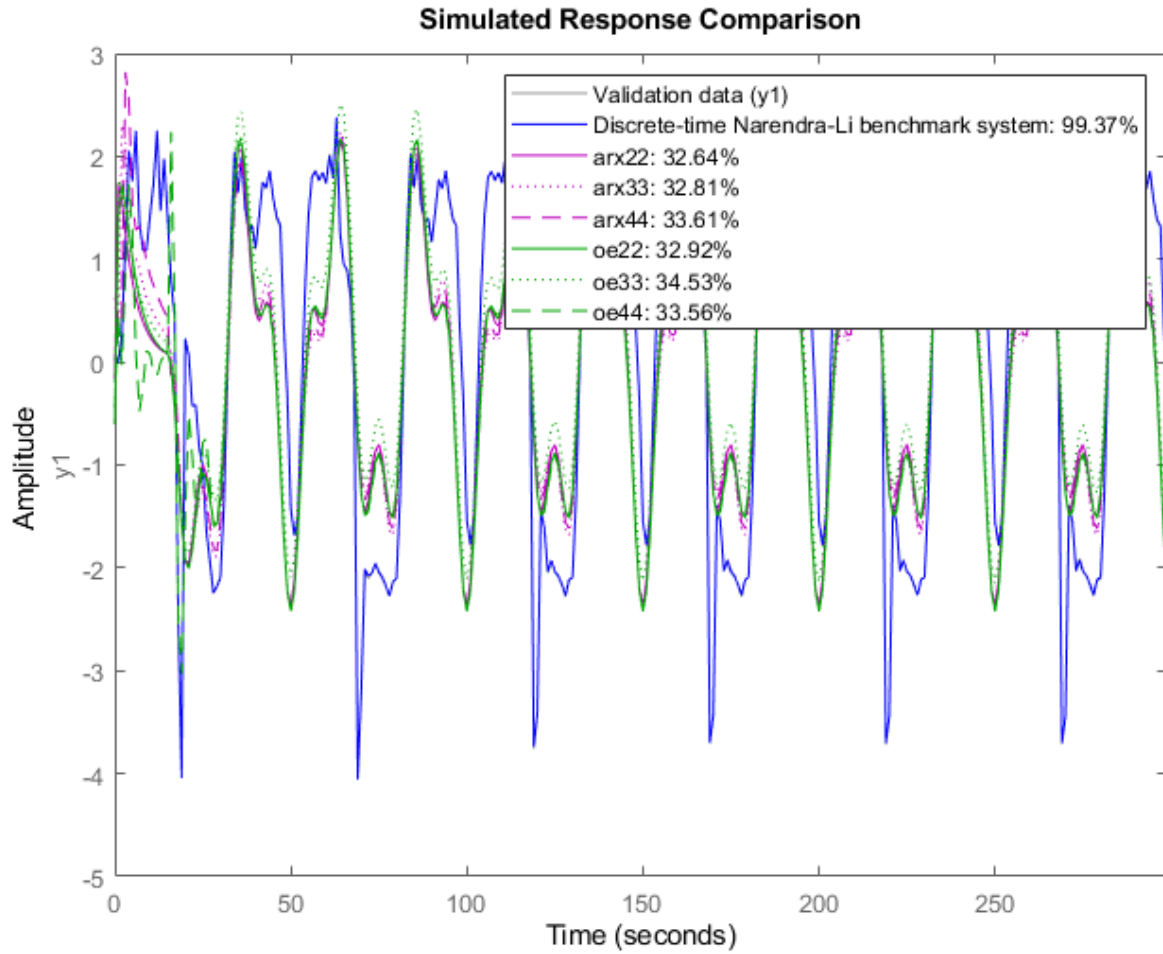
Figure 8: Comparison between true output and the simulated outputs of a number of estimated Narendra-Li models.

Conclusions

In this example we have used a fictive discrete-time Narendra-Li benchmark system to illustrate the basics for performing discrete-time IDNLGREY modeling.

Friction Modeling: MATLAB File Modeling of Static SISO System

This example shows grey-box modeling of a static single-input, single-output system using a MATLAB® function as the ODE file.

System identification normally deals with identifying parameters of dynamic models. However, static models are also of interest, sometimes on their own and sometimes as sub-models of larger more involved models. An example of the latter is discussed in the case study "An Industrial Robot Arm" (idnlgreydemo13.m), where a static friction model is employed as a fixed (pre-estimated) component of a robot arm model.

We discuss how to represent static friction phenomenon as an IDNLGREY model and estimate the corresponding coefficients.

A Continuously Differentiable Friction Model

Discontinuous and piecewise continuous friction models are often problematic for high-performance continuous controllers. This very fact motivated Makkar, Dixon, Sawyer and Hu to suggest a new continuously and differentiable friction model that captures the most common friction phenomena encountered in practice. The new friction model structure was reported in

C. Makkar, W. E. Dixon, W. G. Sawyer, and G.Hu "A New Continuously

Differentiable Friction Model for Control Systems Design", IEEE(R)/ASME

International Conference on Advanced Intelligent Mechatronics,

Monterey, CA, 2005, pages 600-605.

and will serve as a basis for our static identification experiments.

The friction model proposed by Makkar, et al, links the slip speed v(t) of a body in contact with another body to the friction force f(t) via the static relationship

f(t) = g(1)*(tanh(g(2)*v(t) - tanh(g(3)*v(t))

+ g(4)*tanh(g(5)*v(t)) + g(6)*v(t)

where g(1), ..., g(6) are 6 unknown positive parameters. This model structure displays a number of nice properties arising in real-world applications:

1   The friction model is symmetric around the origin

2   The static friction coefficient is approximated by g(1)+g(4)

3   The first term of the equation, tanh(g(2)*v(t) - tanh(g(3)*v(t), captures the so called Striebeck effect, where the friction term shows a rather sudden drop with increasing slip speed near the origin.

4   The Coulombic friction effect is modeled by the term g(4)*tanh(g(5)*v(t))

5   The viscous friction dissipation is reflected by the last term, g(6)*v(t)

Consult the above mentioned paper for many more details about friction in general and the proposed model structure in particular.

IDNLGREY Friction Modeling

Let us now create an IDNLGREY model object describing static friction. As usual, the starting point is to write an IDNLGREY modeling file, and here we construct a MATLAB file, friction_m.m, with contents as follows.

```
function [dx, f] = friction_m(t, x, v, g, varargin)

%FRICTION_M Nonlinear friction model with Stribeck, Coulomb and viscous

% dissipation effects.

% Output equation.

f = g(1)*(tanh(g(2)*v(1))-tanh(g(3)*v(1))) ... % Stribeck effect.

+ g(4)*tanh(g(5)*v(1)) ... % Coulomb effect.

+ g(6)*v(1); % Viscous dissipation term.

% Static system; no states.

dx = [];
```

Notice that a state update dx always must be returned by the model file and that it should be empty ([]) in static modeling cases.

Our next step is to pass the model file, information about model order, guessed parameter vector and so forth as input arguments to the IDNLGREY constructor. We also specify names and units of the input and output and state that all model parameters must be positive.

```
FileName      = 'friction_m';          % File describing the model structure.
Order         = [1 1 0];               % Model orders [ny nu nx].
Parameters    = {[0.20; 90; 11; ...
                  0.12; 110; 0.015]};  % Initial parameters.
InitialStates = [];                    % Initial initial states.
Ts            = 0;                      % Time-continuous system.
nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts,    ...
                'Name', 'Static friction model',                   ...
                'InputName', 'Slip speed', 'InputUnit', 'm/s',     ...
                'OutputName', 'Friction force', 'OutputUnit', 'N', ...
                'TimeUnit', 's');
nlgr = setpar(nlgr, 'Minimum', {zeros(5, 1)});   % All parameters must be >= 0.
```

After these actions we have an initial friction model with properties as follows.

```
present(nlgr);


nlgr =

Continuous-time nonlinear static model defined by 'friction_m' (MATLAB file):

    y(t) = H(t, u(t), p1) + e(t)

 with 1 input(s), 0 state(s), 1 output(s), and 6 free parameter(s) (out of 6).
```

13-179

```
 Inputs:
    u(1)  Slip speed(t) [m/s]
 Outputs:
    y(1)  Friction force(t) [N]
 Parameters:     Value
   p1(1)    p1       0.2      (estimated) in [0, Inf]
   p1(2)             90       (estimated) in [0, Inf]
   p1(3)             11       (estimated) in [0, Inf]
   p1(4)             0.12     (estimated) in [0, Inf]
   p1(5)             110      (estimated) in [0, Inf]
   p1(6)             0.015    (estimated) in [0, Inf]

Name: Static friction model

Status:
Created by direct construction or transformation. Not estimated.
More information in model's "Report" property.

Model Properties
```

In our identification experiments we are not only interested in the full friction model, but also in examining how a reduced friction model would perform. By reduced we here mean a friction model that contains two of the three terms of the full model. To investigate this, three copies of the full model structure are created and in each copy we fix the parameter vector so that only two of the terms will contribute:

```
nlgr1 = nlgr;
nlgr1.Name = 'No Striebeck term';
nlgr1 = setpar(nlgr1, 'Value', {[zeros(3, 1); Parameters{1}(4:6)]});
nlgr1 = setpar(nlgr1, 'Fixed', {[true(3, 1); false(3, 1)]});
nlgr2 = nlgr;
nlgr2.Name = 'No Coulombic term';
nlgr2 = setpar(nlgr2, 'Value', {[Parameters{1}(1:3); 0; 0; Parameters{1}(6)]});
nlgr2 = setpar(nlgr2, 'Fixed', {[false(3, 1); true(2, 1); false]});
nlgr3 = nlgr;
nlgr3.Name = 'No dissipation term';
nlgr3 = setpar(nlgr3, 'Value', {[Parameters{1}(1:5); 0]});
nlgr3 = setpar(nlgr3, 'Fixed', {[false(5, 1); true]});
```

Input-Output Data

At our disposal are 2 different (simulated) data sets where the input slip speed was swept from -10 m/s to 10 m/s in a ramp-type manner. We load the data and create two IDDATA objects for our identification experiments, ze for estimation and zv for validation purposes.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'frictiondata'));
ze = iddata(f1, v, 1, 'Name', 'Static friction system');
set(ze, 'InputName', 'Slip speed', 'InputUnit', 'm/s',  ...
        'OutputName', 'Friction force', 'OutputUnit', 'N', ...
        'Tstart', 0, 'TimeUnit', 's');
zv = iddata(f2, v, 1, 'Name', 'Static friction system');
set(zv, 'InputName', 'Slip speed', 'InputUnit', 'm/s',  ...
        'OutputName', 'Friction force', 'OutputUnit', 'N', ...
        'Tstart', 0, 'TimeUnit', 's');
```

The input-output data that will be used for estimation are shown in a plot window.

```
figure('Name', ze.Name);
plot(ze);
```
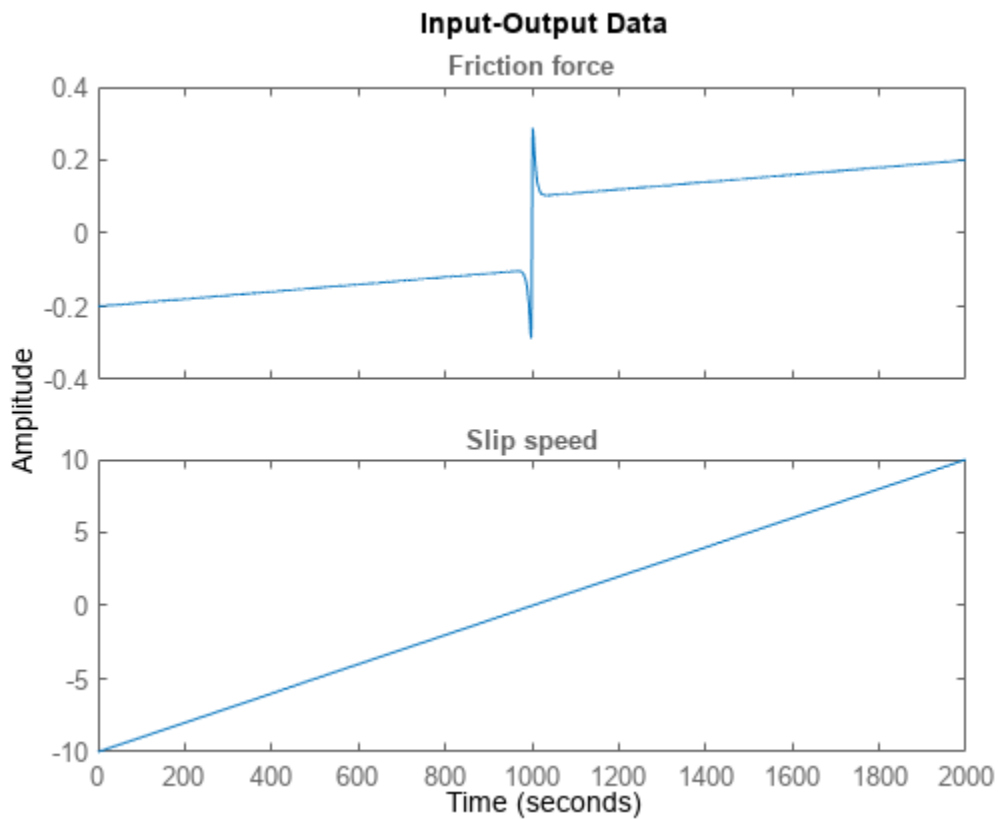

Figure 1: Input-output data from a system exhibiting friction.

Performance of the Initial Friction Models

With access to input-output data and four different initial friction models the obvious question now is how good these models are? Let us investigate this for the estimation data set through simulations carried out by COMPARE:

```
set(gcf,'DefaultLegendLocation','southeast');
compare(ze, nlgr, nlgr1, nlgr2, nlgr3);
```
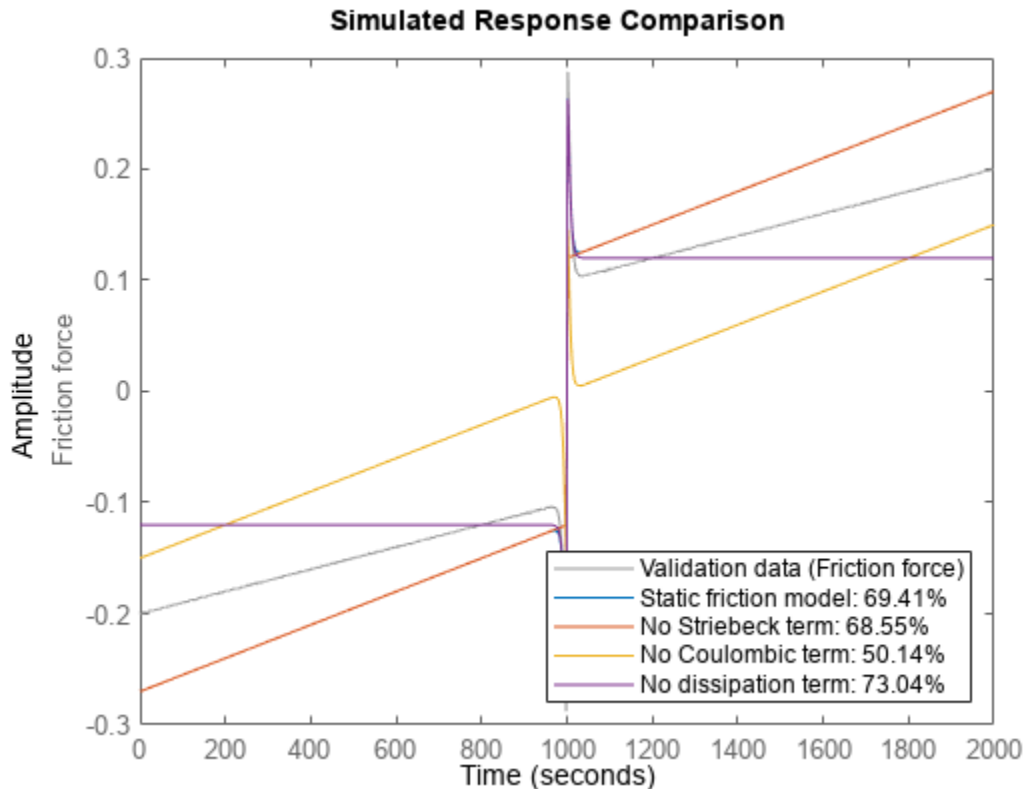
Figure 2: Comparison between true output and the simulated outputs of the four initial friction models.

Parameter Estimation

None of the initial models are able to properly describe the true output. To overcome this we estimate the model parameters of all four model structures. We configure all estimations to perform at most 30 iterations and to stop the iterations only in case the tolerance is almost zero (which it in practice never will be for real-world data). These computations will take some time.

```
opt = nlgreyestOptions('Display', 'on');
opt.SearchOptions.MaxIterations = 30;
opt.SearchOptions.FunctionTolerance = eps;
opt.EstimateCovariance = false;

nlgr  = nlgreyest(nlgr,  ze, opt);
nlgr1 = nlgreyest(nlgr1, ze, opt);
nlgr2 = nlgreyest(nlgr2, ze, opt);
nlgr3 = nlgreyest(nlgr3, ze, opt);
```

Performance of the Estimated Friction Models

The performance of the models are once again investigated by comparing the true output with the simulated outputs of the four models as obtained using COMPARE, but this time the comparison is based on the validation data set zv.

```
compare(zv, nlgr, nlgr1, nlgr2, nlgr3);
```
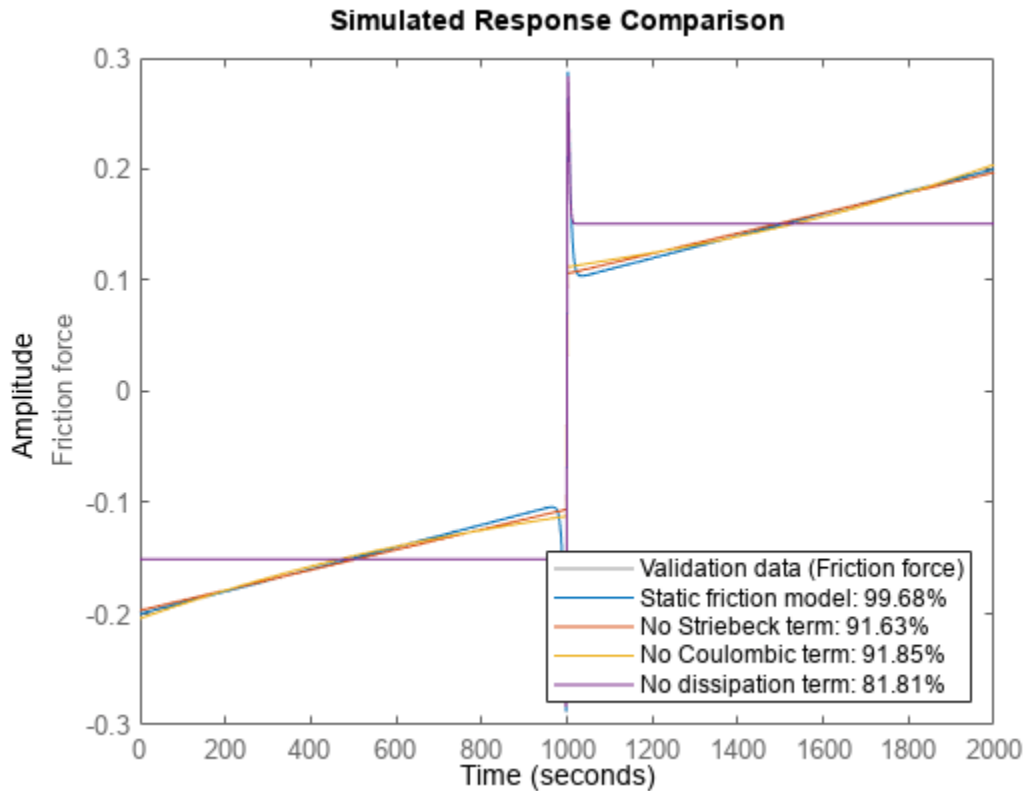
Figure 3: Comparison between true output and the simulated outputs of the four estimated friction models.

For this system we clearly see that the full model outperforms the reduced ones. Nevertheless, the reduced models seem to be able to capture the effects they model rather well, and in each case estimation results in a much better fit. The worst fit is obtained for the model where the viscous dissipation term has been left out. The impressive fit of the full model comes as no big surprise as its model structure coincide with that used to generate the true output data. The parameters of the full model are also close to the ones that were used to generate the true model output:

```
disp('   True        Estimated parameter vector');
```

```
   True        Estimated parameter vector
```

```
ptrue = [0.25; 100; 10; 0.1; 100; 0.01];
fprintf('   %7.3f    %7.3f\n', [ptrue'; getpvec(nlgr)']);
```

```
     0.250       0.249
   100.000     106.637
    10.000       9.978
     0.100       0.100
   100.000      87.699
     0.010       0.010
```

The Final Prediction Error (FPE) criterion (low values are good) applied to all four friction models confirms the superiority of the full friction model:

```
fpe(nlgr, nlgr1, nlgr2, nlgr3);
```

13-183

```
1.0e-03 *

0.0002    0.1665    0.1584    0.7931
```

As for dynamic systems, we can also examine the prediction errors of a static model using PE. We do this for the full friction model and conclude that the residuals seem to have a random nature:

```
pe(zv, nlgr);
```
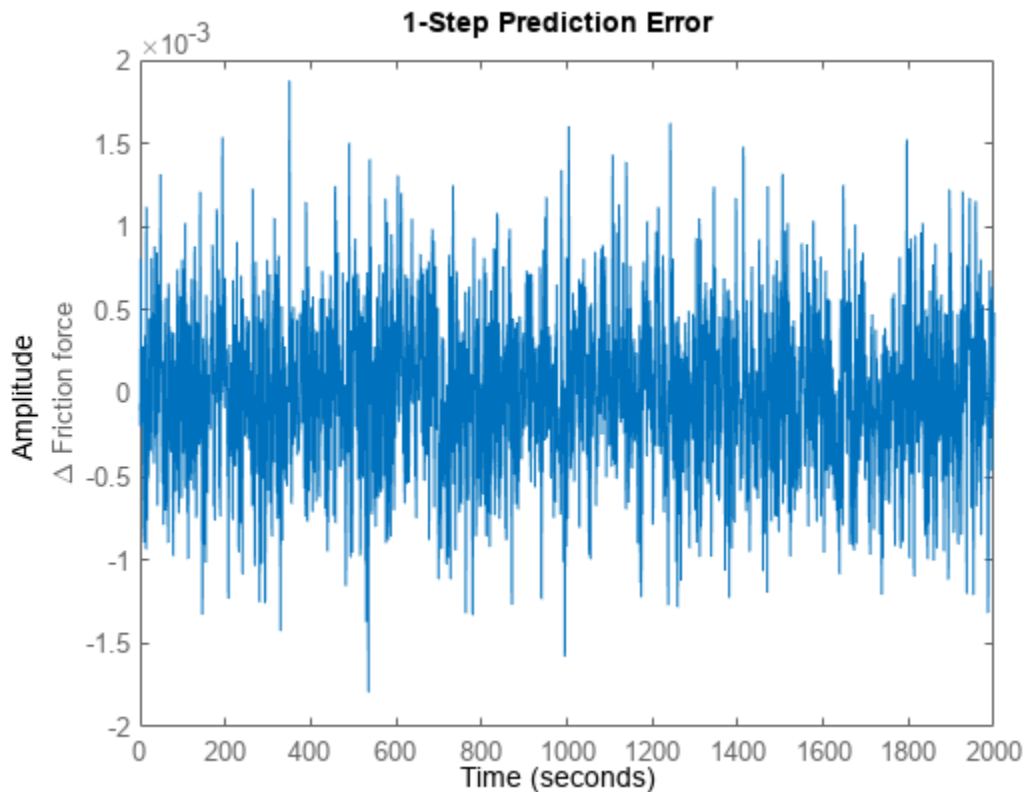

Figure 4: Prediction errors obtained with the estimated full friction model.

We further verify the randomness by looking at the residuals ("leftovers") of the full friction model:

```
figure('Name', [nlgr.Name ': residuals of estimated IDNLGREY model']);
resid(zv, nlgr);
```
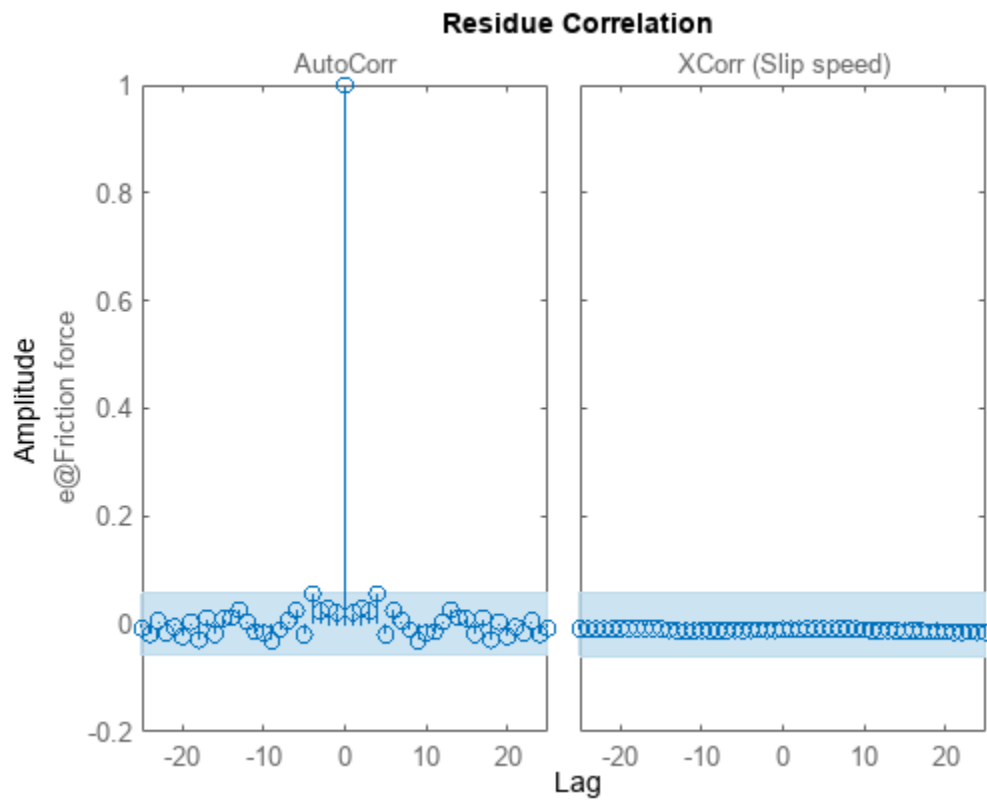
Figure 5: Residuals obtained with the estimated full friction model.

The step response of static models can also be computed and plotted. Let us apply a unit step and do this for all four estimated friction models:

```
figure('Name', [nlgr.Name ': step responses of estimated models']);
step(nlgr, nlgr1, nlgr2, nlgr3);
legend(nlgr.Name, nlgr1.Name, nlgr2.Name, nlgr3.Name, 'location', 'SouthEast');
```
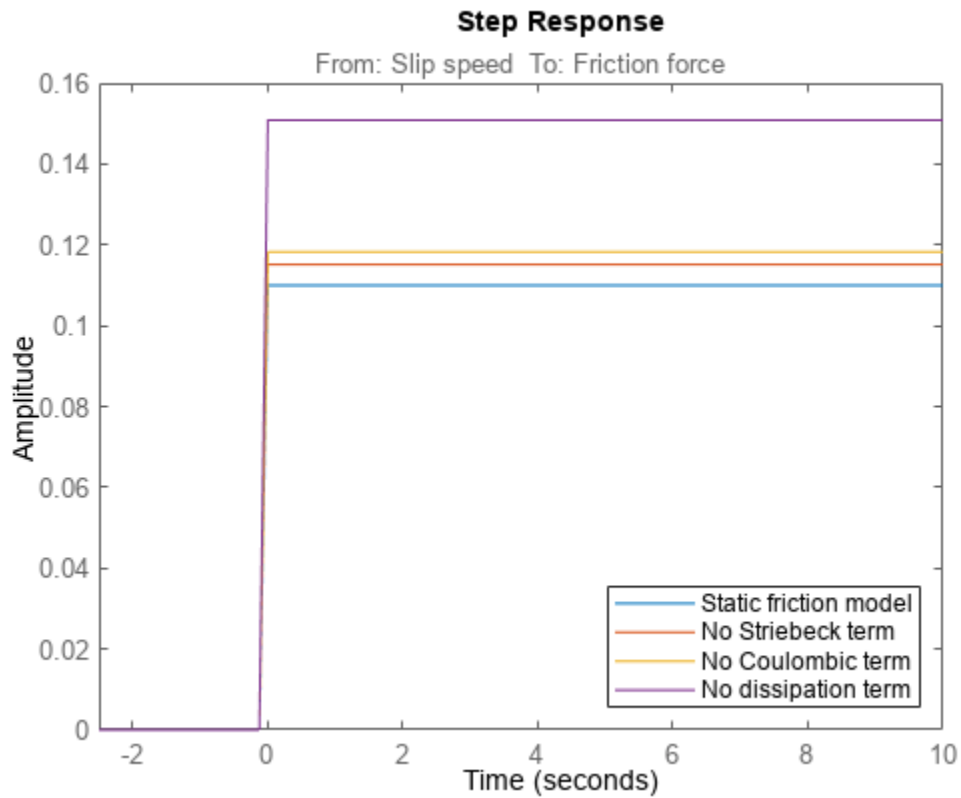
Figure 6: Unit step responses of the four estimated friction models.

We finally display a number of properties, like the estimated standard deviations of the parameters, the loss function, etc., of the full friction model.

```
present(nlgr);
```

```
nlgr =

Continuous-time nonlinear static model defined by 'friction_m' (MATLAB file):

    y(t) = H(t, u(t), p1) + e(t)

 with 1 input(s), 0 state(s), 1 output(s), and 6 free parameter(s) (out of 6).

 Inputs:
    u(1)  Slip speed(t) [m/s]
 Outputs:
    y(1)  Friction force(t) [N]
 Parameters:    Value
    p1(1)   p1     0.249402      (estimated) in [0, Inf]
    p1(2)          106.637       (estimated) in [0, Inf]
    p1(3)          9.97835       (estimated) in [0, Inf]
    p1(4)          0.0999916     (estimated) in [0, Inf]
    p1(5)          87.6992       (estimated) in [0, Inf]
    p1(6)          0.0100019     (estimated) in [0, Inf]
```

```
Name: Static friction model

Status:
Termination condition: Change in parameters was less than the specified tolerance..
Number of iterations: 9, Number of function evaluations: 10

Estimated using Solver: FixedStepDiscrete; Search: lsqnonlin on time domain data "Static friction
Fit to estimation data: 99.68%
FPE: 2.428e-07, MSE: 2.414e-07
More information in model's "Report" property.

Model Properties
```

Conclusions

This example showed how to perform IDNLGREY modeling of a static system. The procedure for doing this is basically the same as for dynamic systems' modeling.

Signal Transmission System: C MEX-File Modeling Using Optional Input Arguments

This example shows how to provide optional input arguments to IDNLGREY models. The discussion concentrates on how to do this for C-MEX types of model files, yet to some minor extent we will also address the most relevant parallels to MATLAB file modeling.

The basis for our discussion will be the signal transmission system (a telegraph wire) schematically depicted in the following figure.
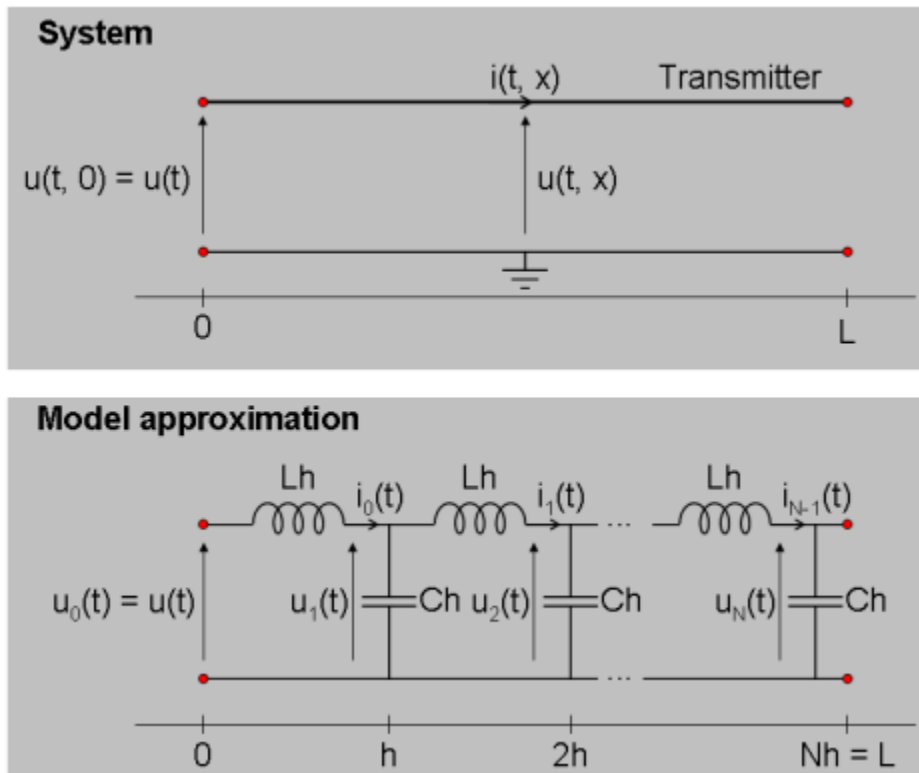
Figure 1: Schematic view of a signal transmission system.

Signal Transmission Modeling

At a distance x (counted from where an input voltage is applied) in a telegraph wire flows at time t the current i(t, x). The corresponding voltage is u(t, x) and the relationship between current and voltage can be described by two coupled Partial Differential Equations (PDEs):

$$-\frac{\partial u(t, x)}{\partial x} = L \frac{\partial i(t, x)}{\partial t}$$

$$-\frac{\partial i(t, x)}{\partial x} = C \frac{\partial u(t, x)}{\partial t}$$

The equations above are often referred to as (a variant of) the so-called "Telegraph equation", with L and C being inductance and capacitance per unit length, respectively.

The telegraph equation can be approximated by a system of ordinary differential equations by only considering the current and the voltage at discrete distance points 0, h, 2h, ..., Nh, where h is the discretization distance and N the total number of such distances. After this approximation, the wire can be thought of as being composed of a number of structurally equal segments connected to each other in a chain. In the literature this type of approximation is commonly referred to as aggregation.

Let the voltage and the current at distance x = kh, for k = 0, 1, ..., N, at time t be denoted u_k(t) and i_k(t), respectively. Approximate d/dx u(t, x) and d/dx i(t, x) through the following simple difference approximations:

```
 d u(t, x)    u_{k+1}(t) - u_k(t)
 --------- ~ -------------------         Forward approximation.
    dx                h

 d i(t, x)    i_k(t) - i_{k-1}(t)
 --------- ~ -------------------         Backward approximation.
    dx                h
```

for x = kh.

The reason for using the forward approximation for d/dx u(t, x) is that i_N(t) = 0 (open wire), so that the remaining N discretized currents can be modeled by the following N differential equations:

```
 d i_k(t)      1
 -------- = - -- (u_{k+1}(t) - u_k(t)) for k = 0, 1, 2, ..., N-1
    dt        Lh
```

Similarly, as u_0(t) is a known input signal and no differential equation is needed to describe it, it is convenient to use the backward approximation scheme to model d/dx i(t, x) at the points h, 2h, ..., N:

```
 d u_k(t)      1
 -------- = - -- (i_k(t) - i_{k-1}(t)) for k = 1, 2, ..., N-1
    dt        Ch

 d u_N(t)      1                        1
 -------- = - -- (i_N(t) - i_{N-1}(t) = -- i_{N-1}(t)
    dt        Ch                       Ch
```

By this we have now arrived at the model approximation shown in Figure 1, in which the equations have been expressed in terms of a number of interconnected coils and capacitors.

Let us now introduce the 2*N states x1(t) = i_0(t), x2(t) = u_1(t), x3(t) = i_1(t), x4(t) = u_2(t), ..., x2N-1(t) = i_{N-1}(t), x2N(t) = u_N(t). Also, denote the input u(t) = u_0(t) and let the output be the voltage at the end of the wire, i.e., y(t) = x2N(t) = u_N(t). Apparent substitutions finally lead to the following state space model structure.

```
    x1(t) = -1/(Lh)*(x2(t) - u(t))
    x2(t) = -1/(Ch)*(x3(t) - x1(t))
    x3(t) = -1/(Lh)*(x4(t) - x2(t))
    x4(t) = -1/(Ch)*(x5(t) - x3(t))
          ...
 x2N-1(t) = -1/(Lh)*(x2N(t) - x2N-2(t))
   x2N(t) =  1/(Ch)*x2N-1(t)

    y(t) = x2N(t)
```

All in all, the above mathematical manipulations have taken us to a standard linear state space model structure, which very well can be handled by IDGREY - the linear counterpart of IDNLGREY. We will

not perform any IDGREY modeling here, but instead show how optional input arguments may be used by IDNLGREY to enhance its modeling flexibility.

IDNLGREY Signal Transmission Model Object

To gain flexibility it is here desirable to have an IDNLGREY model object that immediately is able to deal with a wire of any length L. For modeling quality purposes, it should also be straightforward to vary the number of aggregated blocks N so that a good enough system approximation can be obtained. These requirements can be handled through the passing of N and L in the FileArgument property of an IDNLGREY object. The FileArgument property must be a cell array, but this array may hold any kind of data. In this application, we choose to provide N and L in a structure, and for a wire of length 1000 m we will try three different values of N: 10, 30 and 100. The following three FileArguments will subsequently be used when performing IDNLGREY modeling:

```
FileArgument10  = {struct('N', 10, 'L', 1000)};   % N = 10  --> 20 states.
FileArgument30  = {struct('N', 30, 'L', 1000)};   % N = 30  --> 60 states.
FileArgument100 = {struct('N', 100, 'L', 1000)};  % N = 100 --> 200 states.
```

The parsing and checking of the data contained in FileArgument must be carried out in the IDNLGREY model file, where it is up to the model file designer to implement this functionality. In order to obtain h and N in the function without any error checking the following commands can be used:

```
  N = varargin{1}{1}.N;
  h = varargin{1}{1}.L/N;
```

Notice that FileArgument here corresponds to varargin{1}, which is the last argument passed to the model file. The file signaltransmission_m.m implements the signal transmission model. Type "type signaltransmission_m.m" to see the whole file.

The situation is a bit more involved when using C-MEX model files as will be done further on. In this case we do not beforehand know the number of states Nx, but it is computed in the main interface function and can thus be passed as an input argument to compute_dx and compute_y. The declaration of these functions become:

```
void compute_dx(double *dx, int nx, double *x, double *u, double **p,
            const mxArray *auxvar)
void compute_y(double *y, int nx, double *x)
```

where we have taken away the t-variable (not used in the equations) and instead included an integer nx as a second argument to both these functions. In addition, we have also removed the standard three trailing arguments of compute_y as these are not used to compute the output. With these changes, compute_dx and compute_y are called from the main interface function as follows.

```
  /* Call function for state derivative update. */
  compute_dx(dx, nx, x, u, p, auxvar);

  /* Call function for output update. */
  compute_y(y, nx, x);
```

The first part of compute_dx is shown below. ("type signaltransmission_c.c" displays the whole file in the MATLAB® command window.)

```
  /* State equations. */
  void compute_dx(double *dx, int nx, double *x, double *u, double **p,
                const mxArray *auxvar)
  {
```

```
/* Declaration of model parameters and intermediate variables. */
double *L, *C;      /* Model parameters.                   */
double h, Lh, Ch;   /* Intermediate variables/parameters. */
int    j;           /* Equation counter.                   */

/* Retrieve model parameters. */
L = p[0];   /* Inductance per unit length.  */
C = p[1];   /* Capacitance per unit length. */

/* Get and check FileArgument (auxvar). */
if (mxGetNumberOfElements(auxvar) < 1) {
    mexErrMsgIdAndTxt("IDNLGREY:ODE_FILE:InvalidFileArgument",
                      "FileArgument should at least hold one element.");
} else if (mxIsStruct(mxGetCell(auxvar, 0)) == false) {
    mexErrMsgIdAndTxt("IDNLGREY:ODE_FILE:InvalidFileArgument",
                      "FileArgument should contain a structure.");
} else if (   (mxGetFieldNumber(mxGetCell(auxvar, 0), "N") < 0)
           || (mxGetFieldNumber(mxGetCell(auxvar, 0), "L") < 0)) {
    mexErrMsgIdAndTxt("IDNLGREY:ODE_FILE:InvalidFileArgument",
                      "FileArgument should contain a structure with fields 'N' and 'L'.");
} else {
    /* Skip further error checking to obtain execution speed. */
    h = *mxGetPr(mxGetFieldByNumber(mxGetCell(auxvar, 0), 0,
                mxGetFieldNumber(mxGetCell(auxvar, 0), "L"))) / (0.5*((double) nx));
}
Lh = -1.0/(L[0]*h);
Ch = -1.0/(C[0]*h);
...
```

Worth stressing here is that FileArgument is passed to compute_dx in the variable auxvar. After declaring and retrieving model parameters (and also declaring intermediate variables, like h, auxvar is checked for consistency through a number of external interface routines (so-called mx-routines). These MATLAB routines allow you to create, access, manipulate, and destroy mxArray variables. Consult the MATLAB documentation on External Interfaces for more information about this. The final else-clause is executed if all checks are passed, in which case the value of the auxvar field N is used to determine the value of h. It and the model parameters L and C are finally used to compute the required parameter quantities Lh = -1/(L*h) and Ch = -1/(C*h). See "Tutorials on Nonlinear Grey Box Model Identification: Creating IDNLGREY Model Files" for complimentary information about FileArgument.

With Lh and Ch computed, the second part of compute_dx is as follows. Notice in particular how nx is used in the for-loop of compute_dx to define the number of states of the present model.

```
...
/* x[0]   : Current i_0(t).    */
/* x[1]   : Voltage u_1(t).    */
/* x[2]   : Current i_1(t).    */
/* x[3]   : Voltage u_1(t).    */
/* ...                         */
/* x[Nx-2]: Current i_Nx-1(t). */
/* x[Nx-1]: Voltage u_Nx(t).   */
for (j = 0; j < nx; j = j+2) {
    if (j == 0) {
        /* First transmitter section. */
        dx[j]   = Lh*(x[j+1]-u[0]);
        dx[j+1] = Ch*(x[j+2]-x[j]);
    } else if (j < nx-3) {
```

```
                    /* Intermediate transmitter sections. */
                    dx[j]   = Lh*(x[j+1]-x[j-1]);
                    dx[j+1] = Ch*(x[j+2]-x[j]);
            } else {
                    /* Last transmitter section. */
                    dx[j]   = Lh*(x[j+1]-x[j-1]);
                    dx[j+1] = -Ch*x[j];
            }
        }
    }
```

The output update function compute_dy is much simpler than the state update function:

```
/* Output equation. */
void compute_y(double *y, int nx, double *x)
{
    /* y[0]: Voltage at the end of the transmitter. */
    y[0] = x[nx-1];
}
```

We are now in a position where we straightforwardly can enter the above information into three different IDNLGREY objects, one for N = 10, one for N = 30 and one for N = 100. Notice that the differences when creating these models are the order, the initial state vector and the used optional input argument, but that is it.

```
FileName       = 'signaltransmission_c';            % File describing the model structure.
Parameters     = struct('Name',    {'Inductance per unit length'    ... % Initial parameters.
                                     'Capacitance per unit length'}, ...
                        'Unit',    {'H/m' 'F/m'},                   ...
                        'Value',   {0.99e-3 0.99e-3},               ...
                        'Minimum', {eps(0) eps(0)},                 ... % L, C > 0!
                        'Maximum', {Inf Inf},                       ...
                        'Fixed',   {false false});

% A. Signal transmission model with N = 10;
Order10        = [1 1 2*FileArgument10{1}.N];        % Model orders [ny nu nx].
InitialStates10 = zeros(2*FileArgument10{1}.N, 1);   % Initial intitial states.
nlgr10 = idnlgrey(FileName, Order10, Parameters, InitialStates10, 0, ...
                'FileArgument', FileArgument10,                    ...
                'Name', '10 blocks', 'TimeUnit', 's');

% B. Signal transmission model with N = 30;
Order30        = [1 1 2*FileArgument30{1}.N];        % Model orders [ny nu nx].
InitialStates30 = zeros(2*FileArgument30{1}.N, 1);   % Initial value of intitial states.
nlgr30 = idnlgrey(FileName, Order30, Parameters, InitialStates30, 0, ...
                'FileArgument', FileArgument30,                    ...
                'Name', '30 blocks', 'TimeUnit', 's');

% C. Signal transmission model with N = 100;
Order100        = [1 1 2*FileArgument100{1}.N];      % Model orders [ny nu nx].
InitialStates100 = zeros(2*FileArgument100{1}.N, 1); % Initial value of intitial states.
nlgr100 = idnlgrey(FileName, Order100, Parameters, InitialStates100, 0, ...
                'FileArgument', FileArgument100,                   ...
                'Name', '100 blocks', 'TimeUnit', 's');
```

The names and units of the three signal transmission models are also set, whereupon the sizes of the models are textually confirmed. Input is the voltage applied to the wire and the output is the voltage at the end of the wire.

```
set(nlgr10, 'InputName', 'Vin', 'InputUnit', 'V', ...
            'OutputName', 'Vout', 'OutputUnit', 'V');
set(nlgr30, 'InputName', nlgr10.InputName, 'InputUnit', nlgr10.InputUnit, ...
            'OutputName', nlgr10.OutputName, 'OutputUnit', nlgr10.OutputUnit);
set(nlgr100, 'InputName', nlgr10.InputName, 'InputUnit', nlgr10.InputUnit, ...
            'OutputName', nlgr10.OutputName, 'OutputUnit', nlgr10.OutputUnit);
nlgr10
nlgr30
nlgr100
```

```
nlgr10 =

Continuous-time nonlinear grey-box model defined by 'signaltransmission_c' (MEX-file):

   dx/dt = F(t, u(t), x(t), p1, p2, FileArgument)
    y(t) = H(t, u(t), x(t), p1, p2, FileArgument) + e(t)

 with 1 input(s), 20 state(s), 1 output(s), and 2 free parameter(s) (out of 2).

Name: 10 blocks

Status:
Created by direct construction or transformation. Not estimated.

nlgr30 =

Continuous-time nonlinear grey-box model defined by 'signaltransmission_c' (MEX-file):

   dx/dt = F(t, u(t), x(t), p1, p2, FileArgument)
    y(t) = H(t, u(t), x(t), p1, p2, FileArgument) + e(t)

 with 1 input(s), 60 state(s), 1 output(s), and 2 free parameter(s) (out of 2).

Name: 30 blocks

Status:
Created by direct construction or transformation. Not estimated.

nlgr100 =

Continuous-time nonlinear grey-box model defined by 'signaltransmission_c' (MEX-file):

   dx/dt = F(t, u(t), x(t), p1, p2, FileArgument)
    y(t) = H(t, u(t), x(t), p1, p2, FileArgument) + e(t)

 with 1 input(s), 200 state(s), 1 output(s), and 2 free parameter(s) (out of 2).

Name: 100 blocks

Status:
Created by direct construction or transformation. Not estimated.
```

Input-Output Data

At hand is simulated input-output data from a signal transmission wire of length 1000 m. This data was simulated using the above aggregated model structure, but a much larger N (1500) was employed. The simulation was performed during 20 seconds, using a sampling rate of 0.1 seconds.

13-193

The model parameters used were L = C = 1e-3 and the starting point was a zero voltage wire (all initial states are thus zero). Both model parameters are admittedly much higher than for a typical signal transmission wire, but were so chosen to better illustrate the transport delay involved for this type of systems. We load this data and put it into an IDDATA object z:

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'signaltransmissiondata'));
z = iddata(vout, vin, 0.1, 'Name', 'Signal transmission',                    ...
           'InputName', 'Vin', 'InputUnit', 'V', ...
           'OutputName', 'Vout',         ...
           'OutputUnit', 'V', 'Tstart', 0, 'TimeUnit', 's');
```

A plot of the input-output data clearly unveils the transport delay from applied voltage to the output voltage at the end of the wire. The first input voltage pulse at around 1.4 seconds shows up in the output roughly a second later.

```
figure('Name', [z.Name ': input-output data']);
plot(z);
```
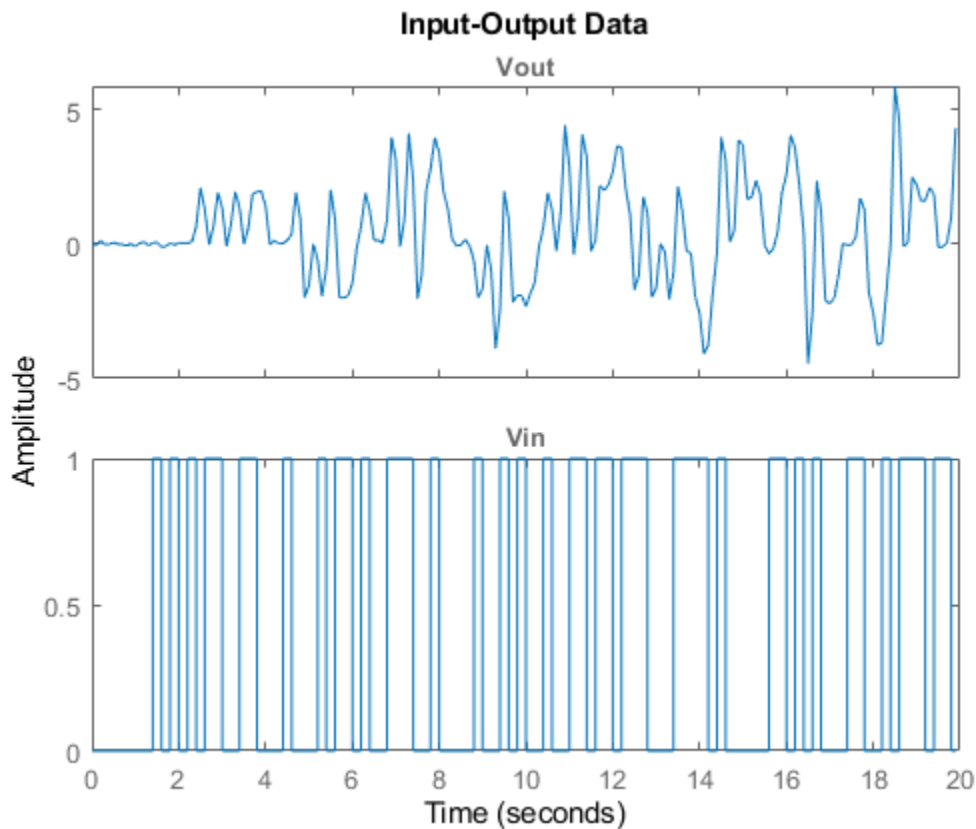

Figure 2: Input-output data from a signal transmission system.

Performance of the Initial Signal Transmission Models

How well do these three initial models perform? Let us investigate this by conducting model simulations using COMPARE. From an execution point of view, it is here vital that the initial states are not estimated, thus the choice of zero initial state vectors. The reason for this is that the number of states is very high, especially for nlgr100 (= 200), and that the estimation of the initial state vectors would then result in really lengthy computations.

```
compare(z, nlgr100, nlgr30, nlgr10, compareOptions('InitialCondition', 'zero'));
```
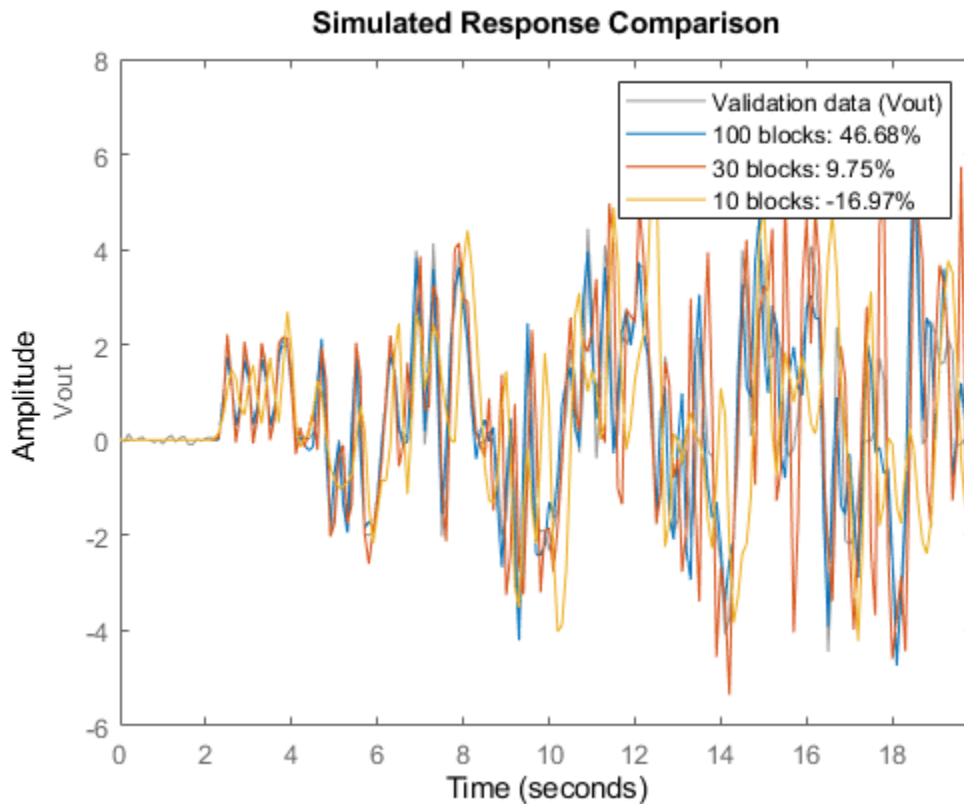

Figure 3: Comparison between true output and the simulated outputs of three initial signal transmission models.

In terms of fit, there is a significant difference between the three models and, as expected, the most complex model outperforms the other two. The difference in modeling power is perhaps best viewed by looking at the prediction errors of each model.

```
peOpt = peOptions('InitialCondition','zero');
e = {pe(z, nlgr100, peOpt) pe(z, nlgr30, peOpt) pe(z, nlgr10, peOpt)};
figtitle = {'nlgr100 : e@' 'nlgr30 : e@' 'nlgr10 : e@'};
figure('Name', [z.Name ': prediction errors']);
for i = 1:3
    subplot(3, 1, i);
    plot(e{i}.SamplingInstants, e{i}.OutputData, 'r');
    title(['Initial ' figtitle{i} z.OutputName{1}]);
    axis('tight');
end
```
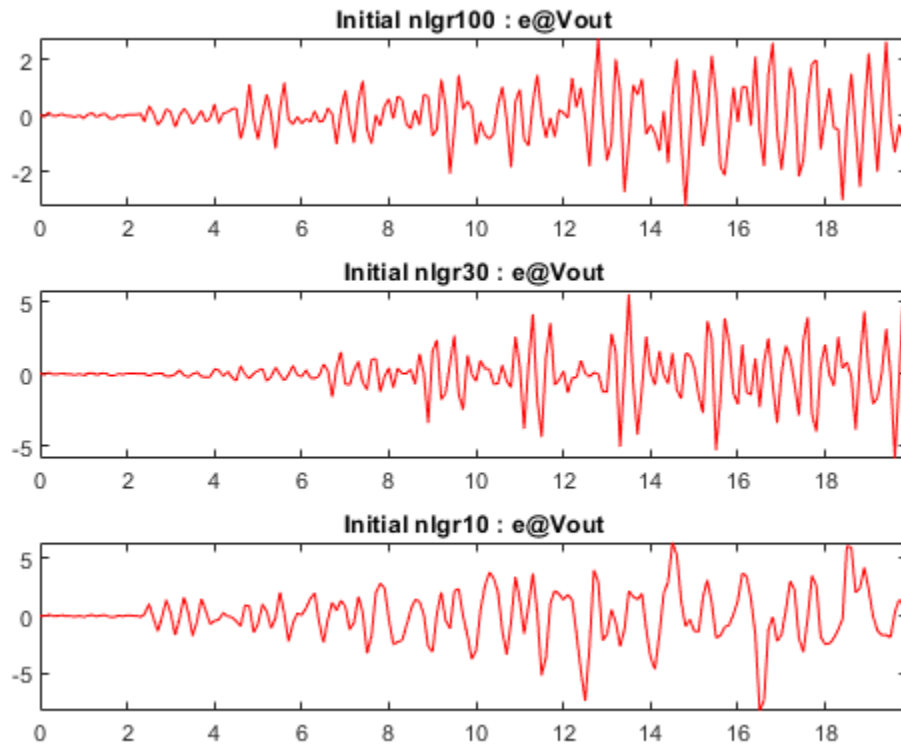
Figure 4: Prediction errors obtained with the three initial IDNLGREY signal transmission models.

Parameter Estimation

Although we have here started off with almost the correct parameter values of the true system, this does not necessarily mean that these values are the ones producing the best model fit. Let us investigate this by estimating the two model parameters of the three signal transmission models using NLGREYEST. These computations may take some time.

```
opt = nlgreyestOptions('Display', 'on', 'EstimateCovariance', false);

nlgr100 = nlgreyest(z, nlgr100, opt);
nlgr30  = nlgreyest(z, nlgr30, opt);
nlgr10  = nlgreyest(z, nlgr10, opt);
```

```
Nonlinear Grey Box Model Estimation
Data has 1 outputs, 1 inputs and 200 samples.
ODE Function: signaltransmission_c
Number of parameters: 2
```

ESTIMATION PROGRESS

```
Algorithm: Trust-Region Reflective Newton
-----------------------------------------------------------------
                            Norm of      First-order
Iteration      Cost          step         optimality
-----------------------------------------------------------------
    0        4.70582           -               -
    1        4.70357        2.17e-05          564
    2        4.69933        1.97e-05          532
    3        4.69289        1.85e-05          493
    4        4.68933         1.7e-05          482
    5         4.6806        1.66e-05          485
    6         4.6806        1.66e-05          485
    7         4.6806        4.16e-06          485
    8        4.68038        1.04e-06          484
    9        4.68038        2.08e-06          484
```

RESULT

```
Termination condition: Change in parameters was less than the specified tolerance..
Number of iterations: 10, Number of function evaluations: 11

Status: Estimated using NLGREYEST
Fit to estimation data: -16.65%, FPE: 4.77494
```

■ Stop   Close

Performance of the Estimated Signal Transmission Models

The Final Prediction Error (FPE) criterion applied to the estimated models continues to indicate that the most complex model is superior:

```
fpe(nlgr100, nlgr30, nlgr10)

   0.5228    0.6771    4.7749
```

The estimated parameter values are not changed that much as compared to the initial values. As can be seen below, an interesting point is that the parameter values of the two least involved models

actually move further away from the true parameter values, whereas the opposite occurs for the most complex model. There are several reasons for why this might happen, e.g., a too coarse aggregation or that the minimization procedure ended up with parameter values corresponding to a local minimum.

```
disp('    True         nlgr100      nlgr30       nlgr10');
ptrue = [1e-3; 1e-3];
fprintf('    %1.7f   %1.7f   %1.7f   %1.7f\n', ...
        [ptrue'; getpvec(nlgr100)'; getpvec(nlgr30)'; getpvec(nlgr10)']);

    True         nlgr100      nlgr30       nlgr10
  0.0010000    0.0009952    0.0009759    0.0009879
  0.0010000    0.0009952    0.0009759    0.0009879
```

Next, we again utilize COMPARE to perform simulations of the three estimated signal transmission models.

```
figure
compare(z, nlgr100, nlgr30, nlgr10, compareOptions('InitialCondition', 'zero'));
```
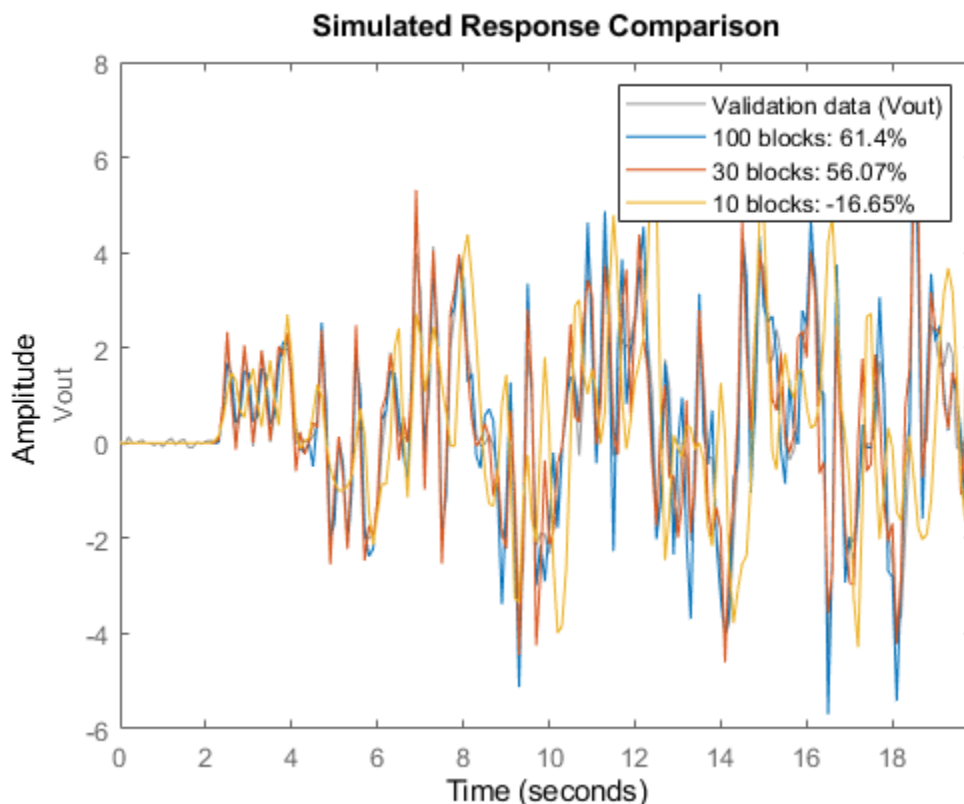

Figure 5: Comparison between true output and the simulated outputs of three estimated signal transmission models.

In all three cases we get a slightly improved fit. Although small, the improvement can be recognized by comparing the prediction errors obtained after and before parameter estimation.

```
peOpt = peOptions('InitialCondition','zero');
e = {pe(z, nlgr100, peOpt) pe(z, nlgr30, peOpt) pe(z, nlgr10, peOpt)};
```

```matlab
figtitle = {'nlgr100 : e@' 'nlgr30 : e@' 'nlgr10 : e@'};
figure('Name', [z.Name ': prediction errors']);
for i = 1:3
    subplot(3, 1, i);
    plot(e{i}.SamplingInstants, e{i}.OutputData, 'r');
    title(['Estimated ' figtitle{i} z.OutputName{1}]);
    axis('tight');
end
```
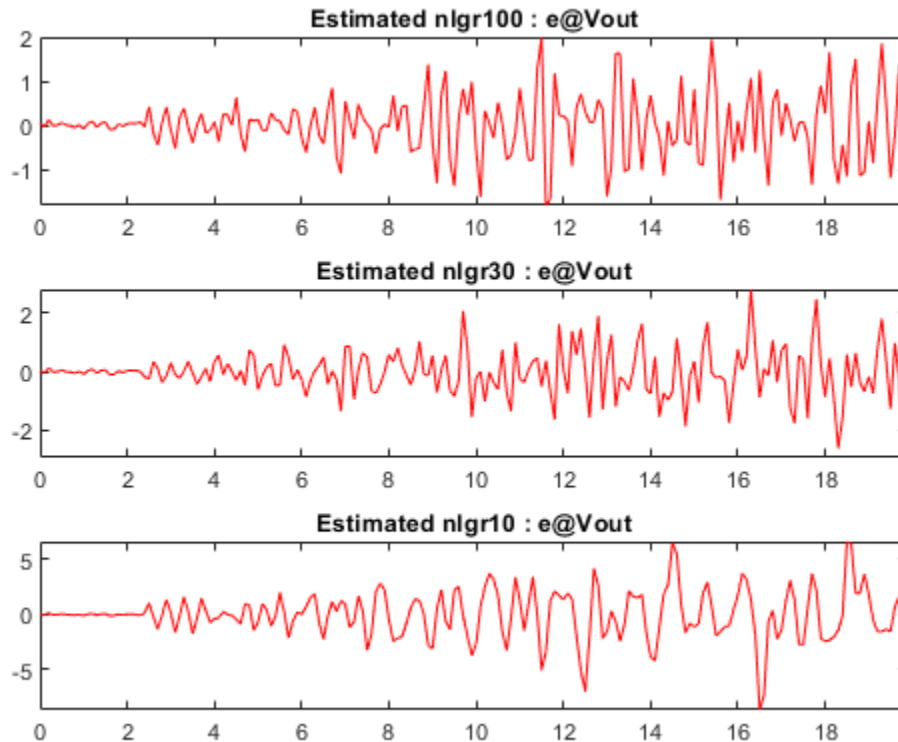

Figure 6: Prediction errors obtained with the three estimated IDNLGREY signal transmission models.

We conclude this tutorial by looking at the unit step responses of the estimated signal transmission models. As can be seen, all these models seem to be able to pick up the transport delay. However, the poles of these linear models are all located on the imaginary axes, i.e., on the stability boundary. It is well-known that simulations of such models are delicate to perform and might result in oscillating step responses of the kind shown in this figure.

The main problem here is that this variant of the telegraph equation does not model any losses; such effects can be included by adding "resistors" (in series with the coils) and "resistors of conductance" (in parallel with the capacitors) to each Lh-Ch sub-block of Figure 1. By this, the models will become more "stable" and simulations will be "easier".

```matlab
figure('Name', [z.Name ': step responses']);
step(nlgr100, nlgr30, nlgr10);
grid on;
legend('nlgr100', 'nlgr30', 'nlgr10', 'Location', 'NorthWest');
```
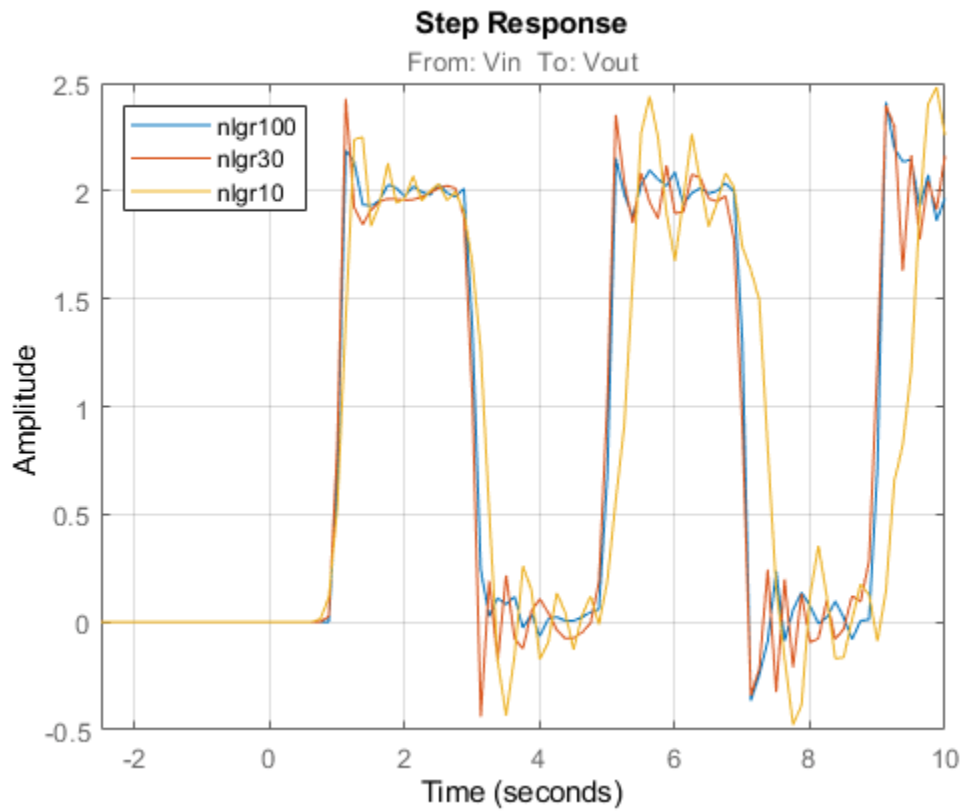
Figure 7: Step responses of three estimated IDNLGREY signal transmission models.

Conclusions

This tutorial has first and foremost addressed how to use optional input file arguments when doing IDNLGREY modeling. We focused on C-MEX type of modeling, but indicated how to do this with MATLAB files too. For certain kinds of systems, it turns out to be beneficial (model reuse) to design a model file with more flexibility than normal, e.g., in terms of the number of states to use. The signal transmission system treated in this tutorial is one example in this category.

Dry Friction Between Two Bodies: Parameter Estimation Using Multiple Experiment Data

This example shows how to estimate parameters of a nonlinear grey box model using multiple experiment data. A system exhibiting dry friction between two solid bodies will be used as the basis for the discussion. In this system, one body is fixed, while the other body moves forward and backward over the fixed body due to an exogenous force according to Figure 1.

Figure 1: Schematic view of a two body system.

Modeling Dry Friction Between Two Bodies

Using Newton's third law of motion, the movement of the moving body is described by:

```
F_tot(t) = m*a(t) = m*d/dt v(t) = m*d^2/dt^2 s(t)
```

where F_tot(t) equals the exogenous force F(t) minus the friction force caused by the contact between the two bodies. The friction force is assumed to be the sum of a sliding friction force F_s(t) and a dry friction force F_d(t). The former is normally modeled as a linear function of the velocity, i.e., F_s(t) = k*v(t), where k is an unknown sliding friction parameter. Dry friction, on the other hand, is a rather complex phenomenon. In the paper:

A. Clavel, M. Sorine and Q. Zhang. "Modeling and identification of a leaf spring system". In third IFAC Workshop on Advances in Automotive Control, 2001.

it is modeled by an ordinary differential equation:

```
d/dt F_d(t) = -1/e*|v(t)|*F_d(t) + f/e*v(t)
```

where e and f are two unknown parameters with dimensions distance and force, respectively.

Denoting the input signal u(t) = F(t) [N], introducing states as:

```
x1(t) = s(t)    Position of the moving body [m].
x2(t) = v(t)    Velocity of the moving body [m/s].
x3(t) = F_d(t)  Dry friction force between the bodies [N].
```

and model parameters as:

```
m        Mass of the moving body [m].
k        Sliding friction force coefficient [kg/s].
e        Distance-related dry friction parameter [m].
f        Force-related dry friction parameter [N].
```

we arrive at the following state space model structure:

```
d/dt x1(t) = x2(t)
d/dt x2(t) = 1/m*(u(t) - k*x2(t) - x3(t))
d/dt x3(t) = 1/e*(-|x2(t)|*x3(t) + f*x2(t))

     y(t) = x1(t)
```

These equations are entered into a C-MEX model file, twobodies_c.c. Its state and output update equations, compute_dx and compute_y, are as follows:

```
/* State equations. */
void compute_dx(double *dx, double t, double *x, double *u, double **p,
                const mxArray *auxvar)
{
    /* Retrieve model parameters. */
    double *m, *k, *e, *f;
    m = p[0];    /* Mass of the moving body.                  */
    k = p[1];    /* Sliding friction force coefficient.       */
    e = p[2];    /* Distance-related dry friction parameter. */
    f = p[3];    /* Force-related dry friction parameter.     */

    /* x[0]: Position. */
    /* x[1]: Velocity. */
    /* x[2]: Dry friction force. */
    dx[0] = x[1];
    dx[1] = (u[0]-k[0]*x[1]-x[2])/m[0];
    dx[2] = (-fabs(x[1])*x[2]+f[0]*x[1])/e[0];
}

/* Output equation. */
void compute_y(double *y, double t, double *x, double *u, double **p,
                const mxArray *auxvar)
{
    /* y[0]: Position. */
    y[0] = x[0];
}
```

Having written the file describing the model structure, the next step is to create an IDNLGREY object reflecting the modeling situation. We also add information about the names and units of the inputs, outputs, states and model parameters of the model structure. Notice that the Parameters and InitialStates are here specified as vectors, which by default means that all model parameters and no initial state vector will be estimated when NLGREYEST is called.

```
FileName      = 'twobodies_c';             % File describing the model structure.
Order         = [1 1 3];                   % Model orders [ny nu nx].
Parameters    = [380; 2200; 0.00012; 1900]; % Initial parameter vector.
InitialStates = [0; 0; 0];                 % Initial states.
Ts            = 0;                         % Time-continuous system.
nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts, ...
                'Name', 'Two body system',                     ...
```

```
                    'InputName', 'Exogenous force',                  ...
                    'InputUnit', 'N',                                ...
                    'OutputName', 'Position of moving body',         ...
                    'OutputUnit', 'm',                               ...
                    'TimeUnit', 's');
nlgr = setinit(nlgr, 'Name', {'Position of moving body' ...
                        'Velocity of moving body' ...
                        'Dry friction force between the bodies'});
nlgr = setinit(nlgr, 'Unit', {'m' 'm/s' 'N'});
nlgr = setpar(nlgr, 'Name', {'Mass of the moving body'              ...
                        'Sliding friction force coefficient',      ...
                        'Distance-related dry friction parameter' ...
                        'Force-related dry friction parameter'});
nlgr = setpar(nlgr, 'Unit', {'m' 'kg/s' 'm' 'N'});
```

Input-Output Data

At this point, we load the available (simulated) input-output data. The file contains data from three different (simulated) test runs each holding 1000 noise-corrupted input-output samples generated using a sampling rate (Ts) of 0.005 seconds. The input u(t) is the exogenous force [N] acting upon the moving body. In the experiments, the input was a symmetric saw-tooth formed signal, where the waveform repetition frequency was the same for all experiments, but where the maximum signal amplitude varied between the test runs. The output y(t) is the position [m] of the moving body (relative to the fixed one). For our modeling purposes, we create one IDDATA object holding three different experiments:

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'twobodiesdata'));
z = merge(iddata(y1, u1, 0.005), iddata(y2, u2, 0.005), iddata(y3, u3, 0.005));
z.Name = 'Two body system';
z.InputName = nlgr.InputName;
z.InputUnit = nlgr.InputUnit;
z.OutputName = nlgr.OutputName;
z.OutputUnit = nlgr.OutputUnit;
z.Tstart = 0;
z.TimeUnit = nlgr.TimeUnit;
```

The input-output data used for the onward identification experiments are shown in a plot window.

```
figure('Name', [z.Name ': input-output data']);
for i = 1:z.Ne
    zi = getexp(z, i);
    subplot(z.Ne, 2, 2*i-1);   % Input.
    plot(zi.SamplingInstants, zi.InputData);
    title([z.ExperimentName{i} ': ' zi.InputName{1}]);
    if (i < z.Ne)
        xlabel('');
    else
        xlabel([z.Domain ' (' zi.TimeUnit ')']);
    end
    axis('tight');
    subplot(z.Ne, 2, 2*i);     % Output.
    plot(zi.SamplingInstants, zi.OutputData);
    title([z.ExperimentName{i} ': ' zi.OutputName{1}]);
    if (i < z.Ne)
        xlabel('');
    else
        xlabel([z.Domain ' (' zi.TimeUnit ')']);
    end
```

```
      axis('tight');
end
```
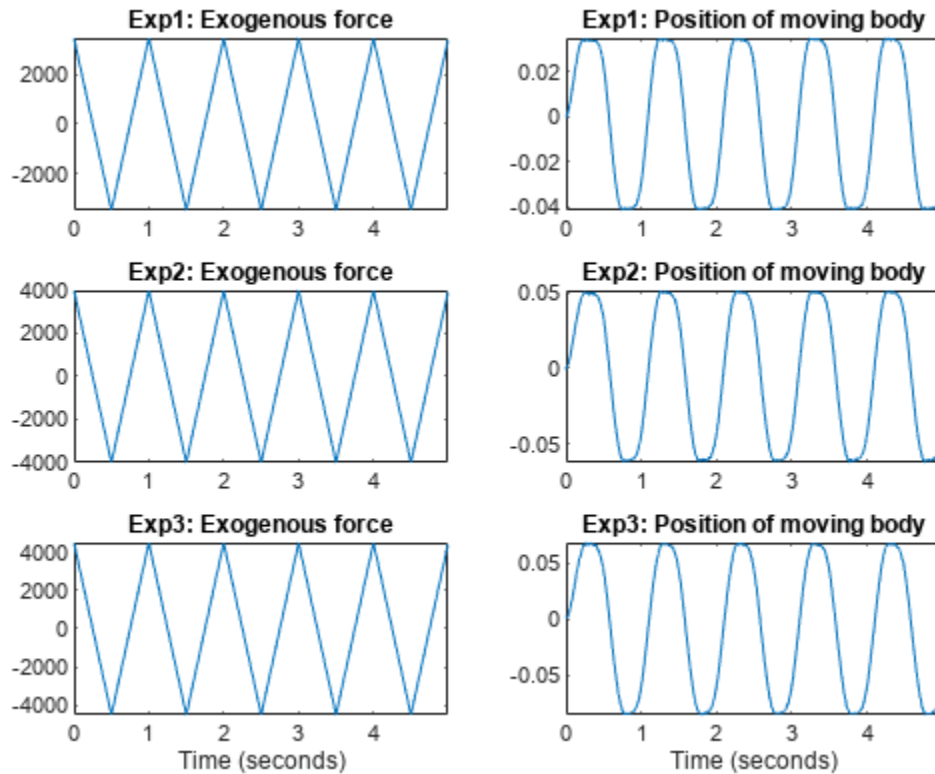

Figure 2: Input-output data from a two body system.

Performance of the Initial Two Body Model

Before estimating the four unknown parameters we simulate the system using the initial parameter values. We use the default differential equation solver (ode45) with the default solver options. When called with only two input arguments, COMPARE will estimate the full initial state vectors, in this case one per experiment, i.e., 3 experiments each with a 3-by-1 state vector implies 9 estimated initial states in total. The simulated and true outputs are shown in a plot window, and as can be seen the fit is decent but not as good as desired.

```
clf
compare(z, nlgr);
```
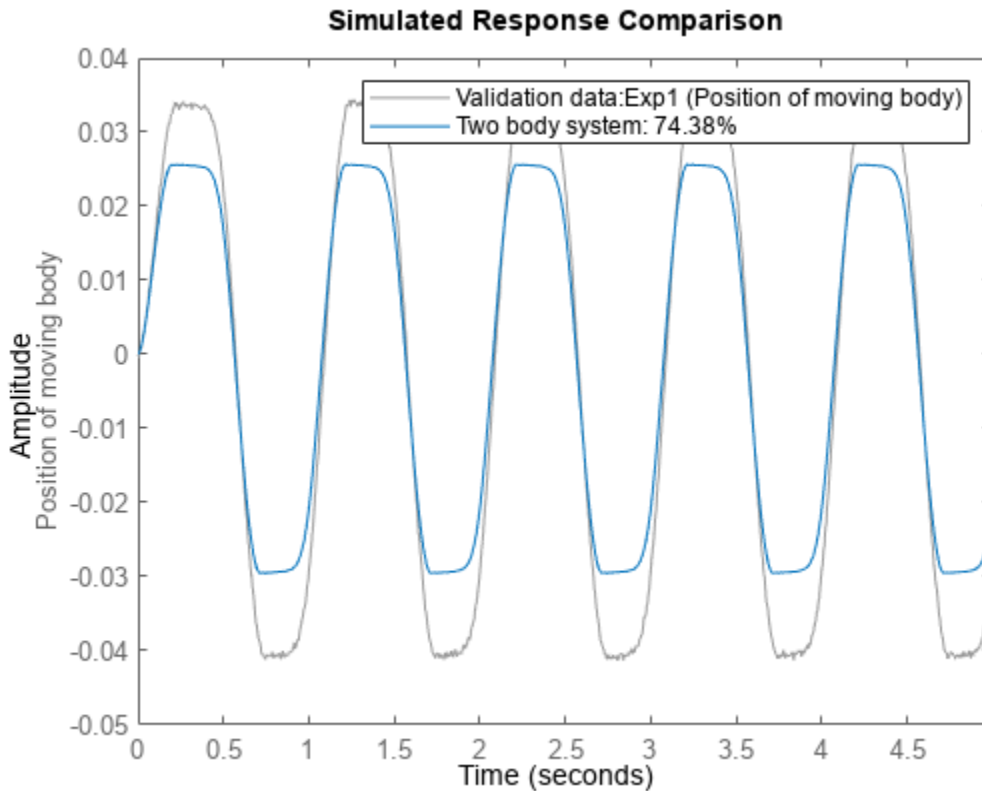
Figure 3: Comparison between true outputs and the simulated outputs of the initial two body model.

Parameter Estimation

In order to improve the fit, the four parameters are next estimated. We use data from the first and last experiments in the estimation phase and keep the data from the second experiment for pure validation purposes.

```
opt  = nlgreyestOptions('Display', 'on');
nlgr = nlgreyest(getexp(z, [1 3]), nlgr, opt);
```

Performance of the Estimated Two Body Model

In order to investigate the performance of the estimated model, a simulation of it is finally performed. By tailoring the initial state structure array of nlgr it is possible to fully specify which states to estimate per experiment in, e.g., COMPARE. Let us here define and use a structure where initial states x1(0) and x2(0) are estimated for experiment 1, x2(0) for experiment 2, and x3(0) for experiment 3. With this modification, a comparison between measured and model outputs is shown in a plot window.

```
nlgr.InitialStates = struct('Name', getinit(nlgr, 'Name'),              ...
                            'Unit', getinit(nlgr, 'Unit'),              ...
                            'Value' , zeros(1, 3), 'Minimum', -Inf(1, 3), ...
                            'Maximum', Inf(1, 3), 'Fixed',              ...
                            {[false false true]; [true false true]; [true true false]});
compare(z, nlgr, compareOptions('InitialCondition', 'model'));
```
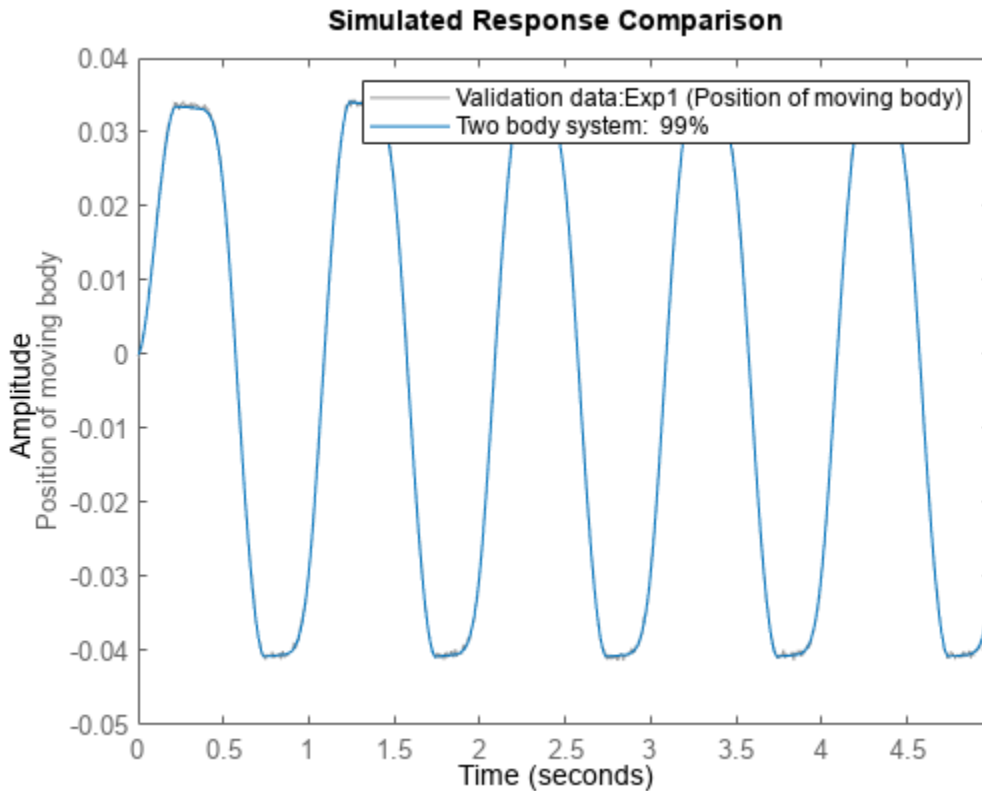
Figure 4: Comparison between true outputs and the simulated outputs of the estimated two body model.

Of special interest is the result with data from the second experiment, which were not used for the parameter estimation. The dynamics of the true system is clearly modeled quite well for all experiments. The estimated parameters are also rather close to the ones used to generate the experimental data:

```
disp('   True            Estimated parameter vector');

   True            Estimated parameter vector

ptrue = [400; 2e3; 0.0001; 1700];
fprintf('   %10.5f    %10.5f\n', [ptrue'; getpvec(nlgr)']);

    400.00000     402.11783
   2000.00000    1986.02824
      0.00010       0.00011
   1700.00000    1705.29327
```

By finally using the PRESENT command, we can get additional information about the estimated model:

```
present(nlgr);


nlgr =
```

```
Continuous-time nonlinear grey-box model defined by 'twobodies_c' (MEX-file):

   dx/dt = F(t, u(t), x(t), p1, ..., p4)
    y(t) = H(t, u(t), x(t), p1, ..., p4) + e(t)

 with 1 input(s), 3 state(s), 1 output(s), and 4 free parameter(s) (out of 4).

 Inputs:
    u(1)  Exogenous force(t) [N]
 States:                                                 Initial value
    x(1)  Position of moving body(t) [m]                   xinit@exp1   0   (estimated) in [-Inf, ]
                                                           xinit@exp2   0   (estimated) in [-Inf, ]
                                                           xinit@exp3   0   (fixed) in [-Inf, Inf]
    x(2)  Velocity of moving body(t) [m/s]                 xinit@exp1   0   (fixed) in [-Inf, Inf]
                                                           xinit@exp2   0   (estimated) in [-Inf, ]
                                                           xinit@exp3   0   (fixed) in [-Inf, Inf]
    x(3)  Dry friction force between the bodies(t) [N]     xinit@exp1   0   (fixed) in [-Inf, Inf]
                                                           xinit@exp2   0   (fixed) in [-Inf, Inf]
                                                           xinit@exp3   0   (estimated) in [-Inf, ]
 Outputs:
    y(1)  Position of moving body(t) [m]
 Parameters:                                       Value
    p1   Mass of the moving body [m]                       402.118     (estimated) in [-Inf, ]
    p2   Sliding friction force coefficient [kg/s]         1986.03     (estimated) in [-Inf, ]
    p3   Distance-related dry friction parameter [m]    0.000105164    (estimated) in [-Inf, ]
    p4   Force-related dry friction parameter [N]          1705.29     (estimated) in [-Inf, ]

Name: Two body system

Status:
Model modified after estimation.
More information in model's "Report" property.

Model Properties
```

Conclusions

This example described how to use multiple experiment data when performing IDNLGREY modeling. Any number of experiments can be employed, and for each such experiment it is possible to fully specify which initial state or states to estimate in NLGREYEST, COMPARE, PREDICT, and so on.

Additional Information

For more information on identification of dynamic systems with System Identification Toolbox™ visit the System Identification Toolbox product information page.

Industrial Three-Degrees-of-Freedom Robot: C MEX-File Modeling of MIMO System Using Vector/Matrix Parameters

This example shows how to design C-MEX model files that involve scalar, vector as well as matrix parameters. As a modeling basis, we will use a somewhat idealized industrial robot, where the left-hand sides of the derived state space equations are not explicitly given. To make it a little bit more illustrative, we will also employ multiple experiment data in the identification part.

Modeling of the Manutec R3 Robot

The considered robot, the Manutec r3, was originally manufactured by Manutec, a Siemens subsidiary company. In reality, the robot comprises six different links, three for positioning a tool center and three for orienting the tool itself. Here we will just consider the modeling of a robot with the three degrees of freedom related to the movement of the tool center. The components of the robot will be modeled as rigid bodies connected by rotational joints with one degree of freedom. Friction and other complex phenomena in the gear-boxes as well as the dynamics of the motors and the sensors are neglected. Even with these simplifications, the resulting model structure is, as we shall see, rather intricate.

The model structure used for the identification experiments conducted below was described in detail in the document:

M. Otter and S. Turk. The DFVLR Models 1 and 2 of the Manutec r3 Robot. Institute for Robotics and System Dynamics, German Aerospace Research Establishment (DLR), Oberpfaffenhofen, May 1988.

and parameter estimation based on the simplified Manutec r3 robot has earlier been considered in the book:

K. Schittkowski. Numerical Data Fitting in Dynamical Systems. Kluwer Academic Publishers, pages 239-242, 2002.

Figure 1 shows a schematic picture of the Manutec r3 robot.
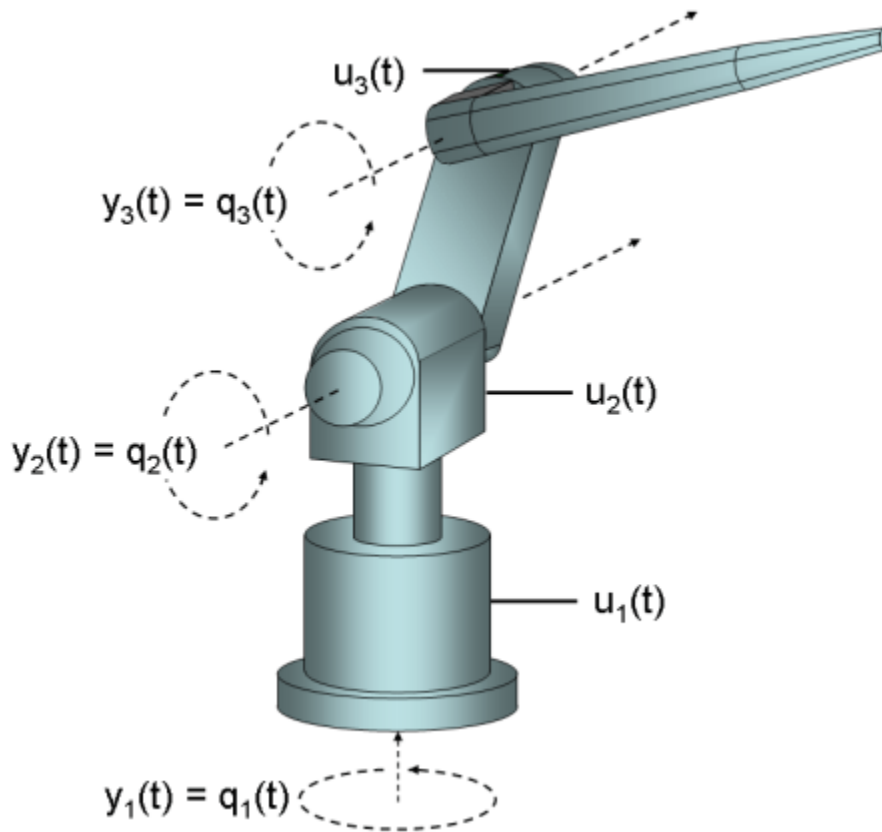
Figure 1: Schematic view of a Manutec r3 robot.

The dynamics of the simplified Manutec r3 robot is given by the vector equation

```
        d^2                                    d
 M(q(t)) ---- q(t) = F(u(t)) + G(q(t)) + H(-- q(t), q(t))
         dt^2                                 dt
```

where the column vector q(t) = [q_1(t), q_2(t), q_3(t)]' describes the relative angle between arm i-1 and arm i for i = 1, 2, 3, with arm 0 corresponding to the coordinate of the fundament. The torque controls u(t) = [u_1(t) u_2(t) u_3(t)]^T applied to the three motors represent the exogenous force moving the robot. These signals are individually scaled (via the force coefficients Fc(1), Fc(2) and Fc(3))) to provide the driving force:

```
F(u(t))  =  [Fc(1)*u_1(t)  Fc(2)*u_2(t)  Fc(3)*u_3(t)]'
```

The mass matrix M(q(t)) is a rather complex symmetric and positive definite 3-by-3 matrix with elements as follows.

```
  M(1, 1) = c_1(p) + c_2(p)*cos(q_2(t))^2 + c_3(p)*sin(q_2(t))^2 +
            c_4(p)*cos(q_2(t)+q_3(t)) + c_5(p)*sin(q_2(t)+q_3(t)) +
            c_6(p)*sin(q_2(t))*sin(q_2(t)+q_3(t))
  M(1, 2) = c_7(p)*cos(q_2(t)) + c_8(p)*cos(q_2(t)+q_3(t))
  M(1, 3) = c_9(p)*cos(q_2(t)+q_3(t))
  M(2, 1) = M(1, 2)
  M(2, 2) = c_10(p) + c_11(p)*cos(q_3(t))
  M(2, 3) = c_12(p) + c_13(p)*cos(q_3(t))
```

```
M(3, 1) = M(1, 3)
M(3, 2) = M(2, 3)
M(3, 3) = c_14(p)
```

where c_1(p), ..., c_14(p) are 14 functions of the robot parameters p.

The robot is also affected by two additional forces. The first one, G(q(t)), is caused by gravity and has elements

```
          G_1(p) = 0
G(p) =    G_2(p) = b_1(p)*sin(q_2(t)) + b_2(p)*sin(q_2(t)+q_3(t))
          G_3(p) = b_3(p)*sin(q_2(t)+q_3(t))
```

where b_1(p), ..., b_3(p) are three functions of the parameters p. The second force, h(d/dt q(t), q(t)), is caused by coriolis and centrifugal forces, which are computed via the so-called Christoffel symbols

```
                 d                  d                  d
  g_ijk = -0.5*(------- M(i, j) + -------- M(i, k) - -------- M(j, k))
                d q_k(t)          d q_j(t)           d q_k(t)
```

as

```
     d                     3   3        d          d
  H_i(-- q(t), q(t)) = sum(sum(g_ijk*-- q_k(t))*-- q_j(t))
     dt                   j=1 k=1       dt         dt
```

for i = 1, 2, 3.

The mass matrix M(q(t)) is invertible (for physically interesting angles), which means that the dynamics of the robot can be written

```
  d^2                -1                          d
  ---- q(t) = M(q(t))   (F(u(t)) + G(q(t)) + H(-- q(t), q(t)))
  dt^2                                          dt
```

By introducing the states

x_1(t) = q_1(t), relative angle between fundament and arm 1.

x_2(t) = q_2(t), relative angle between arm 1 and arm 2.

x_3(t) = q_3(t), relative angle between arm 2 and arm 3.

x_4(t) = d/dt q_1(t), relative velocity between fundament and arm 1.

x_5(t) = d/dt q_2(t), relative velocity between arm 1 and arm 2.

x_6(t) = d/dt q_3(t), relative velocity between arm 2 and arm 3.

we end up with a state space model structure suitable for IDNLGREY modeling. In summary, this model involves 3 inputs, 6 states, 3 outputs, and 28 different model parameters or constants.

IDNLGREY Manutec R3 Robot Model Object

The C-MEX model file developed for this application is quite intricate. We will leave out many of its details and only provide an outline of it; the interested reader is referred to robot_c.c for the complete picture. The above equations result in a robot model with 28 (= Np) different parameters or

constants, that for logical reasons are put into 10 (= Npo) unique parameter objects: 3 scalar ones, 5 vectors, and 2 matrices. The state update function, compute_dx, has the following reduced input argument list:

```
void compute_dx(double *dx, double *x, double *u, double **p)
```

where p holds the 10 parameter objects:

- g = p[0], pl = p[5] and Ia1 = p[8] are scalars.
- Fc = p[1], r = p[2], Im = p[3], m = p[4] and L = p[6] are column vectors with two or three entries.
- com = p[7] is a 2-by-2 matrix and Ia = p[9] a 4-by-2 matrix.

The scalars are as usual referenced as p[0] (p(1) 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 com(1, 1) is referred as com[0], com(2, 1) as com[1], com(1, 2) as com[3], and com(2, 2) as com[3]. Similarly, the eight elements of Ia are obtained via Ia[i] for i = 0, 1, ..., 7.

With this, compute_dx involves the following computational steps (note that many assignments are left out here). The purpose of steps A-E is to restructure the equations so that the states can be explicitly computed.

```
void compute_dx(double *dx, double *x, double *u, double **p)
{
    /* Declaration of model parameters and intermediate variables. */
    double *g, *Fc, *r, *Im, *m, *pl, *L, *com, *Ia1, *Ia;
    double M[3][3];      /* Mass matrix.                          */
    ...

    /* Retrieve model parameters. */
    ...

    /* A. Components of the symmetric and positive definite mass matrix M(x, p), a 3x3 matrix.
    M[0][0] = Ia1[0] + r[0]*r[0]*Im[0] + com[2]*com[2]*m[1] ...
    ...
    M[2][2] = Ia[4] + r[2]*r[2]*Im[2] + com[3]*com[3]*m[1] + L[1]*L[1]*pl[0];

    /* B. Inputs. */
    F[0] = Fc[0]*u[0]; ...

    /* C. Gravitational forces G. */
    G[0] = 0; ...

    /* D. Coriolis and centrifugal force components Gamma and forces H. */
    Gamma[1] = (Ia[6] - Ia[5] - com[3]*com[3]*m[1] ...

    /* E. Compute inverse of M. */
    Det = M[0][0]*M[1][1]*M[2][2] + 2*M[0][1]*M[1][2]*M[0][2] ...

    /* State equations. */
    /* x[0]: Relative angle between fundament and arm 1. */
    /* x[1]: Relative angle between arm 1 and arm 2. */
    /* x[2]: Relative angle between arm 2 and arm 3. */
    /* x[3]: Relative velocity between fundament and arm 1. */
    /* x[4]: Relative velocity between arm 1 and arm 2. */
    /* x[5]: Relative velocity between arm 2 and arm 3. */
    dx[0] = x[3];
```

```
        dx[1] = x[4];
        dx[2] = x[5];
        dx[3] = Minv[0][0]*(F[0]+G[0]+H[0]) + Minv[0][1]*(F[1]+G[1]+H[1]) + Minv[0][2]*(F[2]+G[2]+H
        dx[4] = Minv[0][1]*(F[0]+G[0]+H[0]) + Minv[1][1]*(F[1]+G[1]+H[1]) + Minv[1][2]*(F[2]+G[2]+H
        dx[5] = Minv[0][2]*(F[0]+G[0]+H[0]) + Minv[1][2]*(F[1]+G[1]+H[1]) + Minv[2][2]*(F[2]+G[2]+H
    }
```

The output update function, compute_y, is appreciably simpler:

```
    /* Output equations. */
    void compute_y(double y[], double x[])
    {
        /* y[0]: Relative angle between fundament and arm 1. */
        /* y[1]: Relative angle between arm 1 and arm 2. */
        /* y[2]: Relative angle between arm 2 and arm 3. */
        y[0] = x[0];
        y[1] = x[1];
        y[2] = x[2];
    }
```

Consult the "Creating IDNLGREY Model Files" example for further details about C-MEX model files.

We now have sufficient knowledge to create an IDNLGREY object reflecting the movement of the simplified Manutec r3 robot. We start by describing the inputs:

```
InputName = {'Voltage applied to motor moving arm 1'; ...
             'Voltage applied to motor moving arm 2'; ...
             'Voltage applied to motor moving arm 3'};
InputUnit = {'V'; 'V'; 'V'};
```

Next, we define the six states, the first three being the outputs:

```
StateName  = {'\Theta_1'; ...    % Relative angle between fundament and arm 1
              '\Theta_2'; ...    % Relative angle between arm 1 and arm 2
              '\Theta_3'; ...    % Relative angle between arm 2 and arm 3
              'Vel_1'; ...       % Relative velocity between fundament and arm 1
              'Vel_2'; ...       % Relative velocity between arm 1 and arm 2
              'Vel_3'...         % Relative velocity between arm 2 and arm 3
             };
StateUnit  = {'rad'; 'rad'; 'rad'; 'rad/s'; 'rad/s'; 'rad/s'};
OutputName = StateName(1:3);
OutputUnit = StateUnit(1:3);
```

As mentioned earlier, the model involves 28 different parameters or constants that are lumped into 10 different parameter objects as follows. Notice that some of the parameters, by physical reasoning, are specified to have distinct minimum values. These minimum parameter values are defined in a cell array with elements of the same size as are used for specifying the parameter values.

```
ParName  = {'Gravity constant';                       ... % g, scalar.
            'Voltage-force constant of motor';         ... % Fc, 3-by-1 vector, for motor 1, 2
            'Gear ratio of motor';                     ... % r, 3-by-1 vector, for motor 1, 2,
            'Moment of inertia of motor';              ... % Im, 3-by-1 vector, for motor 1, 2
            'Mass of arm 2 and 3 (incl. tool)';        ... % m, 2-by-1 vector, for arm 2 and 3
            'Point mass of payload';                   ... % pl, scalar.
            'Length of arm 2 and 3 (incl. tool)';      ... % L, 2-by-1 vector, for arm 2 and 3
            'Center of mass coordinates of arm 2 and 3'; ... % com, 2-by-2 matrix, 1:st column f
            'Moment of inertia arm 1, element (3,3)';  ... % Ia1, scalar.
            'Moment of inertia arm 2 and 3'            ... % Ia, 4-by-2 matrix. 1:st column for
```

```
                                                    ... % column elements: 1: (1,1); 2: (2,2
            };
ParUnit   = {'m/s^2'; 'N*m/V'; ''; 'kg*m^2'; 'kg'; 'kg'; 'm'; 'm'; 'kg*m^2'; 'kg*m^2'};
ParValue  = {9.81; [-126; 252; 72]; [-105; 210; 60]; [1.3e-3; 1.3e-3; 1.3e-3]; ...
            [56.5; 60.3]; 10; [0.5; 0.98]; [0.172 0.028; 0.205 0.202];        ...
            1.16; [2.58 11.0; 2.73 8.0; 0.64 0.80; -0.46 0.50]};
ParMin    = {eps(0); -Inf(3, 1); -Inf(3, 1); eps(0)*ones(3, 1); [40; 40]; ...
            eps(0); eps(0)*ones(2, 1); eps(0)*ones(2); -Inf; -Inf(4, 2)};
ParMax    = Inf;   % No maximum constraint.
```

After having specified the model file, the initial state, and so on, a Manutec r3 IDNLGREY model object is created as follows:

```
FileName      = 'robot_c';              % File describing the model structure.
Order         = [3 3 6];                % Model orders [ny nu nx].
Parameters    = struct('Name', ParName, 'Unit', ParUnit, 'Value', ParValue, ...
                       'Minimum', ParMin, 'Maximum', ParMax, 'Fixed', false);
InitialStates = struct('Name', StateName, 'Unit', StateUnit, 'Value', 0, ...
                       'Minimum', -Inf, 'Maximum', Inf, 'Fixed', true);
Ts            = 0;                       % Time-continuous system.
nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts,      ...
                'Name', 'Manutec r3 robot', 'InputName', InputName, ...
                'InputUnit', InputUnit, 'OutputName', OutputName,   ...
                'OutputUnit', OutputUnit,  'TimeUnit', 's');
```

Input-Output Data

Next we load the available input-output data. The outputs were here simulated using the above IDNLGREY model structure. Before storage, the outputs were corrupted by some additive noise.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'robotdata'));
```

The file holds data from two different (simulation) experiments each holding 101 input-output samples generated using a sampling rate (Ts) of 0.02 seconds. Starting with a zero initial state, the inputs to the three motors [V] used in the experiments were all kept constant:

```
  u(t) = [u_1(t) u_2(t) u_3(t)]^T =  [0.20 0.05 0.03]^T   % Experiment 1.
  u(t) = [u_1(t) u_2(t) u_3(t)]^T = -[0.20 0.05 0.03]^T   % Experiment 2.
```

The generated outputs hold data according to the above description. For our modeling purposes, we create one IDDATA object z containing data from the two experiments:

```
z = merge(iddata(y1, u1, 0.02), iddata(y2, u2, 0.02));
z.Name = 'Manutec r3 robot';
z.InputName = nlgr.InputName;
z.InputUnit = nlgr.InputUnit;
z.OutputName = nlgr.OutputName;
z.OutputUnit = nlgr.OutputUnit;
z.Tstart = 0;
z.TimeUnit = 's';
```

Performance of the Initial Manutec R3 Robot Model

Before proceeding with parameter estimation we simulate the model using the initial parameter values. We use the default differential equation solver (ode45) with a somewhat higher requirement on the relative accuracy than what is used by default. The simulated and true outputs are shown in a plot window, and as is indicated the fit is already now decent (maybe except for the fit between true and simulated relative angle between arm 2 and arm 3, i.e., the third output).

```
nlgr.SimulationOptions.RelTol = 1e-5;
compare(z, nlgr);
```
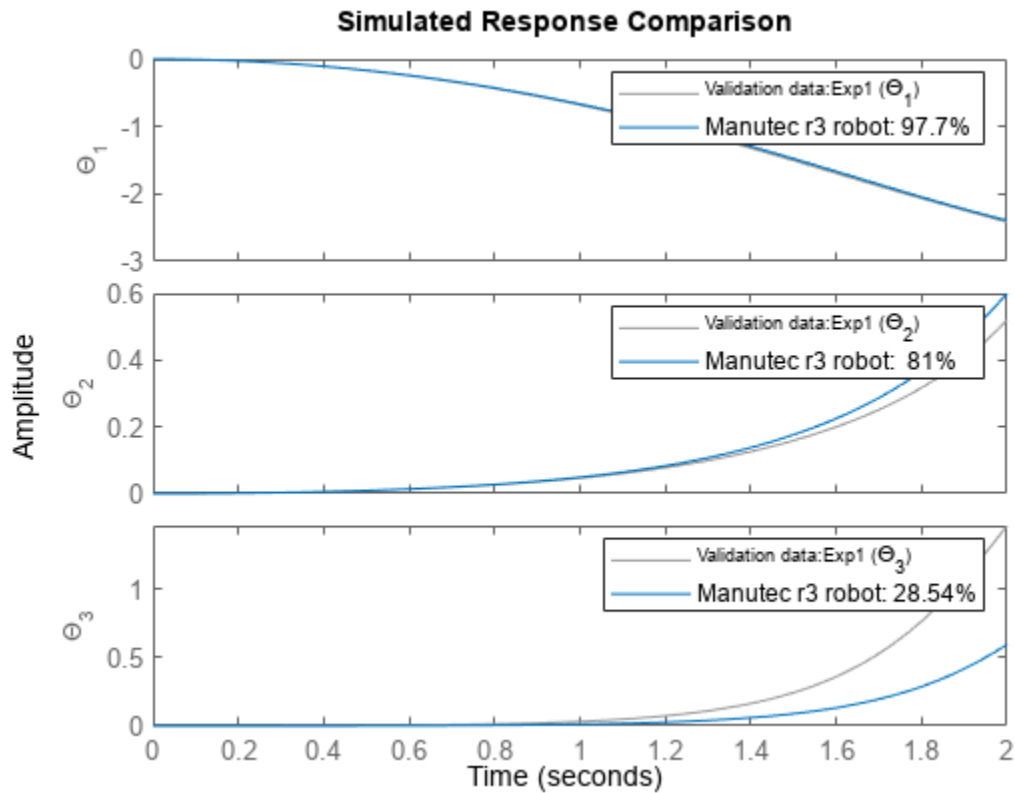

Figure 2: Comparison between true outputs and the simulated outputs of the initial Manutec r3 robot model.

Parameter Estimation

Identification of the Manutec r3 parameters is quite demanding, partly because the available data is rather limited in terms of excitation, and partly because of the highly nonlinear nature of the robot dynamics. In order to simplify the task, we only estimate the last four parameters, i.e., the moments of inertia related to arm 3 and the tool:

```
for k = 1:size(nlgr, 'Npo')-1    % Fix all parameters of the first 9 parameter objects.
    nlgr.Parameters(k).Fixed = true;
end
nlgr.Parameters(end).Fixed(:, 1) = true;    % Fix the moment of inertia parameters for arm 2.
```
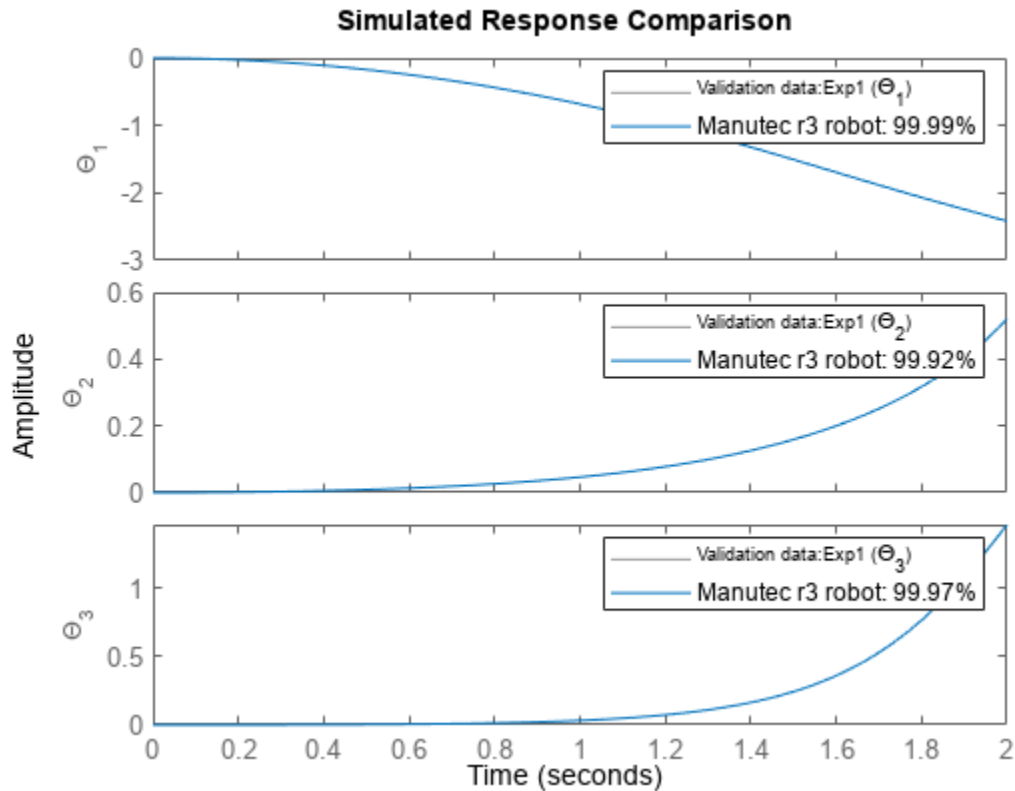
This time we use a Levenberg-Marquardt search algorithm.

```
opt = nlgreyestOptions('SearchMethod', 'lm', 'Display', 'on');
nlgr = nlgreyest(z, nlgr, opt);
```

Performance of the Estimated Manutec R3 Robot Model

The performance of the estimated Manutec r3 robot is next investigated through a simulation.

```
compare(z, nlgr);
```

Figure 3: Comparison between true outputs and the simulated outputs of the estimated Manutec r3 robot model.

As can be seen in the figure, the fit between simulated and true outputs have improved considerably, especially when it comes to the third output (the relative angle between arm 2 and arm 3). The true and the estimated parameters are quite close to each other:

```
disp('   True      Estimated parameter vector');

   True      Estimated parameter vector

ptrue = [9.81; -126; 252; 72; -105; 210; 60; 1.3e-3; 1.3e-3; 1.3e-3; ...
         56.5; 60.3; 10; 0.5; 0.98; 0.172; 0.205; 0.028; 0.202; 1.16; ...
         2.58; 2.73; 0.64; -0.46; 5.41; 5.60; 0.39; 0.33];
fprintf('   %1.3f    %1.3f\n', ptrue(25), nlgr.Parameters(end).Value(1, 2));

   5.410    5.414

fprintf('   %1.3f    %1.3f\n', ptrue(26), nlgr.Parameters(end).Value(2, 2));

   5.600    5.609

fprintf('   %1.3f    %1.3f\n', ptrue(27), nlgr.Parameters(end).Value(3, 2));

   0.390    0.390

fprintf('   %1.3f    %1.3f\n', ptrue(28), nlgr.Parameters(end).Value(4, 2));

   0.330    0.331
```

Let us further investigate the estimated Manutec r3 robot model via the PRESENT command:

```
present(nlgr);
```

```
nlgr =

Continuous-time nonlinear grey-box model defined by 'robot_c' (MEX-file):

   dx/dt = F(t, u(t), x(t), p1, ..., p10)
    y(t) = H(t, u(t), x(t), p1, ..., p10) + e(t)

 with 3 input(s), 6 state(s), 3 output(s), and 4 free parameter(s) (out of 28).

 Inputs:
   u(1)   Voltage applied to motor moving arm 1(t) [V]
   u(2)   Voltage applied to motor moving arm 2(t) [V]
   u(3)   Voltage applied to motor moving arm 3(t) [V]
 States:                        Initial value
   x(1)   \Theta_1(t) [rad]    xinit@exp1   0   (fixed) in [-Inf, Inf]
                               xinit@exp2   0   (fixed) in [-Inf, Inf]
   x(2)   \Theta_2(t) [rad]    xinit@exp1   0   (fixed) in [-Inf, Inf]
                               xinit@exp2   0   (fixed) in [-Inf, Inf]
   x(3)   \Theta_3(t) [rad]    xinit@exp1   0   (fixed) in [-Inf, Inf]
                               xinit@exp2   0   (fixed) in [-Inf, Inf]
   x(4)   Vel_1(t) [rad/s]     xinit@exp1   0   (fixed) in [-Inf, Inf]
                               xinit@exp2   0   (fixed) in [-Inf, Inf]
   x(5)   Vel_2(t) [rad/s]     xinit@exp1   0   (fixed) in [-Inf, Inf]
                               xinit@exp2   0   (fixed) in [-Inf, Inf]
   x(6)   Vel_3(t) [rad/s]     xinit@exp1   0   (fixed) in [-Inf, Inf]
                               xinit@exp2   0   (fixed) in [-Inf, Inf]
 Outputs:
   y(1)   \Theta_1(t) [rad]
   y(2)   \Theta_2(t) [rad]
   y(3)   \Theta_3(t) [rad]
 Parameters:                                             Value Standard Deviation
   p1        Gravity constant [m/s^2]                     9.81            0    (f:
   p2(1)     Voltage-force constant of motor [N*m/V]       -126           0    (f:
   p2(2)                                                   252            0    (f:
   p2(3)                                                    72            0    (f:
   p3(1)     Gear ratio of motor                          -105           0    (f:
   p3(2)                                                   210            0    (f:
   p3(3)                                                    60            0    (f:
   p4(1)     Moment of inertia of motor [kg*m^2]         0.0013          0    (f:
   p4(2)                                                 0.0013          0    (f:
   p4(3)                                                 0.0013          0    (f:
   p5(1)     Mass of arm 2 and 3 (incl. tool) [kg]        56.5           0    (f:
   p5(2)                                                   60.3          0    (f:
   p6        Point mass of payload [kg]                     10           0    (f:
   p7(1)     Length of arm 2 and 3 (incl. tool) [m]         0.5          0    (f:
   p7(2)                                                   0.98          0    (f:
   p8(1,1)   Center of mass coordinates of arm 2 and 3 [m]  0.172        0    (f:
   p8(2,1)                                                 0.205         0    (f:
   p8(1,2)                                                 0.028         0    (f:
   p8(2,2)                                                 0.202         0    (f:
   p9        Moment of inertia arm 1, element (3,3) [kg*m^2]  1.16       0    (f:
   p10(1,1)  Moment of inertia arm 2 and 3 [kg*m^2]          2.58        0    (f:
   p10(2,1)                                                  2.73        0    (f:
```

```
        p10(3,1)                                                        0.64                  0    (f:
        p10(4,1)                                                       -0.46                  0    (f:
        p10(1,2)                                                     5.41371        1.80292e-11    (es
        p10(2,2)                                                     5.60905        7.67168e-11    (es
        p10(3,2)                                                    0.389978        1.58707e-12    (es
        p10(4,2)                                                    0.330506        2.25628e-12    (es

Name: Manutec r3 robot

Status:
Termination condition: Near (local) minimum, (norm(g) < tol)..
Number of iterations: 10, Number of function evaluations: 49

Estimated using Solver: ode45; Search: lm on time domain data "Manutec r3 robot".
Fit to estimation data: [99.99 99.99;99.92 99.93;99.97 99.97]%
FPE: 9.788e-25, MSE: [2.821e-08 3.086e-08]
More information in model's "Report" property.

Model Properties
```

Some Identification Remarks

In this case we obtain parameters very close to the true ones. However, generally speaking there are a number of reasons for why the true parameters might not be found:

1   The available data is not "informative enough" for identifying the parameters (the data is not persistently exciting).

2   The data is corrupted by so much noise that it is virtually impossible to find the true parameters.

3   A local and not the searched-for global minimum is found. This risk is always present when using iterative search algorithms.

4   The parameters of the model structure are not uniquely identifiable. Generally speaking, this risk becomes larger when more parameters are estimated. (Estimating all 28 parameters of the Manutec r3 robot, e.g., typically leads to some physically impossible parameter values.)

5   The model structure is just "too complex" or contains nonlinearities that are not "sufficiently smooth".

Conclusions

The main purpose of this tutorial has been to illustrate how to include parameters that are vectors or matrices in an IDNLGREY C-MEX modeling framework. In addition, we did this for a robot modeling example, where the equations had to be manipulated in order to fit into the required explicit state space form.

Non-Adiabatic Continuous Stirred Tank Reactor: MATLAB File Modeling with Simulations in Simulink

This example shows how to include and simulate an IDNLGREY model in Simulink®. We use a chemical reaction system as a modeling basis. The first modeling and identification part of the example can be run without Simulink.

Modeling A Non-Adiabatic Continuous Stirred Tank Reactor

A rather common chemical system encountered in the process industry is the Continuously Stirred Tank Reactor (CSTR). Here we will study a jacketed diabatic (i.e., non-adiabatic) tank reactor described extensively in Bequette's book "Process Dynamics: Modeling, Analysis and Simulation", published by Prentice-Hall, 1998. The vessel is assumed to be perfectly mixed, and a single first-order exothermic and irreversible reaction, A --> B, takes place. A schematic diagram of the vessel and the surrounding cooling jacket is shown in a plot window. Notice that this is a sketch; in reality the coolant flow is, e.g., normally surrounding the whole reactor jacket, and not just the bottom of it.
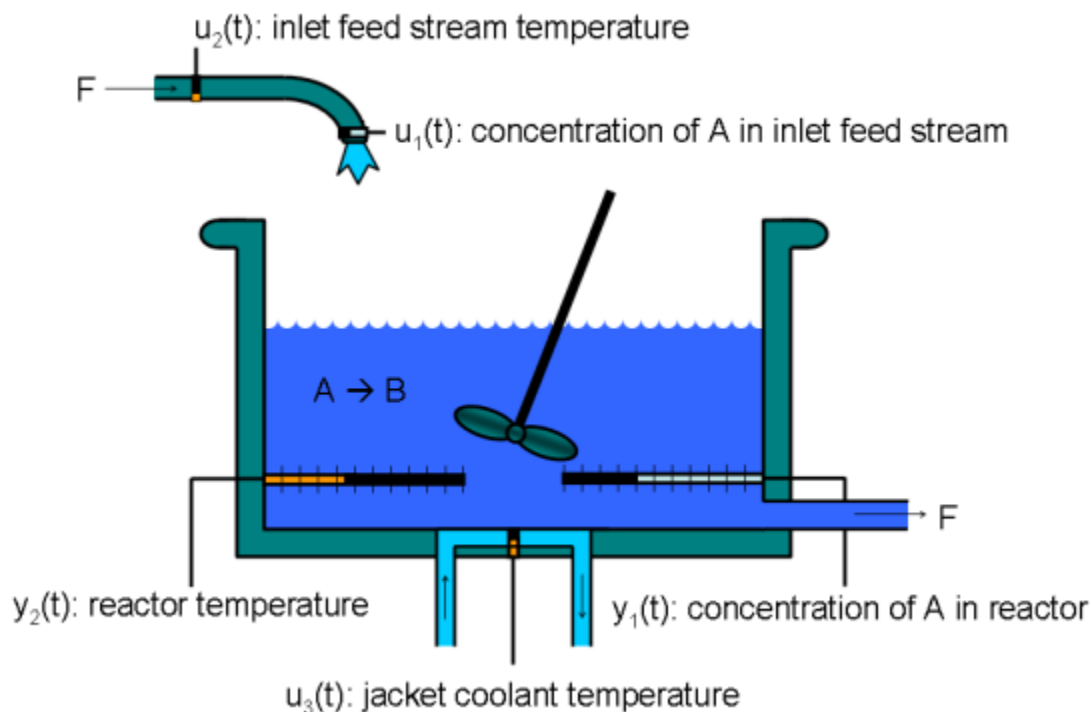
Figure 1: Schematic diagram of a CSTR.

A model of the CSTR is required for more advanced control approaches. The inlet stream of reagent A is fed at a constant rate F. After stirring, the end product streams out of the vessel at the same rate as reagent A is fed into the tank (the volume V in the reactor tank is thus kept constant). The control strategy requires that the jacket temperature u_3(t) is manipulated in order to keep the concentration of reagent A y_1(t) at the desired level, in spite of disturbances arising from the inlet feed stream concentration and temperature (inputs u_1(t) and u_2(t), respectively). As the temperature in the tank y_2(t) can vary significantly during operation of the reactor, it is also desirable to ensure that this process variable is kept within reasonable limits.

Modeling the CSTR

The CSTR system is modeled using basic accounting and energy conservation principles. The change of the concentration of reagent A in the vessel per time unit d C_A(t)/dt (= d y_1(t)/dt) can be modeled as:

```
  d C_A(t)
  -------- = F/V*(C_Af(t)-C_A(t)) - r(t)
     dt
```

where the first term expresses concentration changes due to differences between the concentration of reagent A in the inlet stream and in the vessel, and the second term expresses concentration changes (reaction rate) that occurs due to the chemical reaction in the vessel. The reaction rate per unit volume is described by Arrhenius rate law:

```
  r(t) = k_0*exp(-E/(R*T(t)))*C_A(t)
```

which states that the rate of a chemical reaction increases exponentially with the absolute temperature. k_0 is here an unknown nonthermal constant, E is the activation energy, R Boltzmann's ideal gas constant and T(t) (= y_2(t)) the temperature in the reactor.

Similarly, using the energy balance principle (assuming constant volume in the reactor), the temperature change per time unit d T(t)/dt in the reactor can be modeled as:

```
  d T(t)
  ------ = F/V(T_f(t)-T(t)) - (H/c_p*rho)*r(t) - (U*A)/(c_p*rho*V)*(T(t)-T_j(t))
     dt
```

where the first and third terms describe changes due to that the feed stream temperature T_f(t) and the jacket coolant temperature T_j(t) differ from the reactor temperature, respectively. The second term is the influence on the reactor temperature caused by the chemical reaction in the vessel. In this equation, H is a heat of reaction parameter, c_p a heat capacity term, rho a density term, U an overall heat transfer coefficient and A the area for the heat exchange (coolant/vessel area).

Put together, the CSTR has three input signals:

```
  u_1(t) = C_Af(t)  Concentration of A in inlet feed stream [kgmol/m^3].
  u_2(t) = T_f(t)   Inlet feed stream temperature [K].
  u_3(t) = T_j(t)   Jacket coolant temperature [K].
```

and two output signals:

```
  y_1(t) = C_A(t)   Concentration of A in reactor tank [kgmol/m^3].
  y_2(t) = T(t)     Reactor temperature [K].
```

which also are the natural model states, i.e., y_1(t) = x_1(t) and y_2(t) = x_2(t).

After lumping together some of the original parameters we end up with eight different model parameters:

```
  F             Volumetric flow rate (volume/time) [m^3/h].   Fixed.
  V             Volume in reactor [m^3].                       Fixed.
  k_0           Pre-exponential nonthermal factor [1/h].       Free.
  E             Activation energy [kcal/kgmol].                Free.
  R             Boltzmann's gas constant [kcal/(kgmol*K)].     Fixed.
  H             Heat of reaction [kcal/kgmol].                 Fixed.
  HD = c_p*rho  Heat capacity times density [kcal/(m^3*K)].    Free.
```

```
HA = U*A           Overall heat transfer coefficient times tank
                   area [kcal/(K*h)]                          Free.
```

Four of the parameters are here specified to be free (i.e., to be estimated). In practice, one could probably also determine the pre-exponential nonthermal factor k_0 and the activation energy E from bench scale experiments. This would then simplify the identification task to only consider two unknowns: the heat capacity c_p and the overall heat transfer coefficient U (which are inferred from HD and HA, respectively, as rho and A are known).

With the above introduced notation, the following state-space representation is obtained for the CSTR.

```
d x_1(t)
-------- =  F/V*(u_1(t)-x_1(t)) - k_0*exp(-E/(R*x_2(t)))*x_1(t)
   dt

d x_2(t)
-------- =   F/V(u_2(t)-x_2(t)) - (H/HD)*k_0*exp(-E/(R*x_2(t)))*x_1(t)
   dt       - (HA/(HD*V))*(x_2(t)-u_3(t))

  y_1(t) = x_1(t)
  y_2(t) = x_2(t)
```

Creating A Non-Adiabatic Continuous Stirred Tank Reactor IDNLGREY Object

This information is entered into a MATLAB® file named cstr_m.m with the following content.

```
function [dx, y] = cstr_m(t, x, u, F, V, k_0, E, R, H, HD, HA, varargin)
%CSTR_M  A non-adiabatic Continuous Stirred Tank Reactor (CSTR).

% Output equations.
y = [x(1);                ... % Concentration of substance A in the reactor.
     x(2)                 ... % Reactor temperature.
    ];

% State equations.
dx = [F/V*(u(1)-x(1))-k_0*exp(-E/(R*x(2)))*x(1); ...
      F/V*(u(2)-x(2))-(H/HD)*k_0*exp(-E/(R*x(2)))*x(1)-(HA/(HD*V))*(x(2)-u(3)) ...
     ];
```

An IDNLGREY object reflecting the modeling situation is next created.

```
FileName      = 'cstr_m';                    % File describing the model structure.
Order         = [2 3 2];                     % Model orders [ny nu nx].
Parameters    = [1; 1; 35e6; 11850; ...      % Initial parameters.
                 1.98589; -5960; 480; 145];
InitialStates = [8.5695; 311.267];           % Initial value of the initial states.
Ts            = 0;                           % Time-continuous system.
nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts, 'Name', ...
                'Stirred tank reactor',  ...
                'TimeUnit', 'hours');
```

The inputs, states and outputs of the CSTR model structure are specified using the methods SET and SETINIT. We also specify that the initial states by default should be estimated.

```
nlgr.InputName = {'Concentration of A in inlet feed stream'  ...   % u(1).
                  'Inlet feed stream temperature'            ...   % u(2).
                  'Jacket coolant temperature'};                   % u(3).
nlgr.InputUnit = {'kgmol/m^3' 'K' 'K'};
```

```
nlgr = setinit(nlgr, 'Name', {'Concentration of A in reactor tank'        ...    % x(1).
                     'Reactor temperature'});                          ...    % x(2).
nlgr = setinit(nlgr, 'Unit', {'kgmol/m^3' 'K'});
nlgr = setinit(nlgr, 'Fixed', {false false});
nlgr.OutputName = {'A Concentration' ...    % y(1); Concentration of A in reactor tank
                    'Reactor temp.'};    % y(2).
nglr.OutputUnit = {'kgmol/m^3' 'K'};
```

The parameters of the CSTR model structure are defined and F, V, R and H are specified to be fixed.

```
nlgr = setpar(nlgr, 'Name', {'Volumetric flow rate (volume/time)'              ...    % F.
                     'Volume in reactor'                            ...    % V.
                     'Pre-exponential nonthermal factor'            ...    % k_0.
                     'Activation energy'                            ...    % E.
                     'Boltzmann''s ideal gas constant'              ...     % R.
                     'Heat of reaction'                             ...    % H.
                     'Heat capacity times density'                  ...    % HD.
                     'Overall heat transfer coefficient times tank area'}); ...    % HA.
nlgr = setpar(nlgr, 'Unit', {'m^3/h' 'm^3' '1/h' 'kcal/kgmol' 'kcal/(kgmol*K)'      ...
                     'kcal/kgmol' 'kcal/(m^3*K)' 'kcal/(K*h)'});
nlgr.Parameters(1).Fixed = true;    % Fix F.
nlgr.Parameters(2).Fixed = true;    % Fix V.
nlgr.Parameters(5).Fixed = true;    % Fix R.
nlgr.Parameters(6).Fixed = true;    % Fix H.
```

Through physical reasoning we also know that all but the heat of reaction parameter (always negative because the reaction is exothermic) are positive. Let us also incorporate this (somewhat crude) knowledge into our CSTR model structure:

```
nlgr.Parameters(1).Minimum = 0;    % F.
nlgr.Parameters(2).Minimum = 0;    % V.
nlgr.Parameters(3).Minimum = 0;    % k_0.
nlgr.Parameters(4).Minimum = 0;    % E.
nlgr.Parameters(5).Minimum = 0;    % R.
nlgr.Parameters(6).Maximum = 0;    % H.
nlgr.Parameters(7).Minimum = 0;    % HD.
nlgr.Parameters(8).Minimum = 0;    % HA.
```

A summary of the entered CSTR model structure is next obtained through the PRESENT command:

```
present(nlgr);
```

```
nlgr =

Continuous-time nonlinear grey-box model defined by 'cstr_m' (MATLAB file):

   dx/dt = F(t, u(t), x(t), p1, ..., p8)
    y(t) = H(t, u(t), x(t), p1, ..., p8) + e(t)

 with 3 input(s), 2 state(s), 2 output(s), and 4 free parameter(s) (out of 8).

 Inputs:
    u(1)  Concentration of A in inlet feed stream(t) [kgmol/m^3]
    u(2)  Inlet feed stream temperature(t) [K]
    u(3)  Jacket coolant temperature(t) [K]
 States:                                         Initial value
    x(1)  Concentration of A in reactor tank(t) [kgmol/m^3]   xinit@exp1    8.5695   (estimated)
```

```
    x(2)  Reactor temperature(t) [K]                          xinit@exp1   311.267   (estimated)
 Outputs:
    y(1)  A Concentration(t)
    y(2)  Reactor temp.(t)
 Parameters:                                              Value
    p1   Volumetric flow rate (volume/time) [m^3/h]                    1      (fixed)
    p2   Volume in reactor [m^3]                                       1      (fixed)
    p3   Pre-exponential nonthermal factor [1/h]                   3.5e+07    (estima
    p4   Activation energy [kcal/kgmol]                             11850     (estima
    p5   Boltzmann's ideal gas constant [kcal/(kgmo..]            1.98589     (fixed)
    p6   Heat of reaction [kcal/kgmol]                              -5960     (fixed)
    p7   Heat capacity times density [kcal/(m^3*K)]                   480     (estima
    p8   Overall heat transfer coefficient times tank area [kcal/(K*h)]   145      (estima

Name: Stirred tank reactor

Status:
Created by direct construction or transformation. Not estimated.
More information in model's "Report" property.

<a href="matlab:if exist('nlgr','var'), if isa(nlgr,'idnlgrey'), disp(' '); disp(get(nlgr)), else
```

Input-Output Data

System identification experiment design for many nonlinear systems is typically much more involved than for linear systems. This is also true for the CSTR, where on one hand it is desired that the controllable input u_3 is such that it excites the system sufficiently and on the other hand it must be chosen to be "plant-friendly" (the chemical process must be kept stable, the duration of the test should be as short as possible so as to influence the production least, and so forth). The article in [1] discusses the choice of input signals to the CSTR. There it is argued that a multi-sinusoidal input u_3 is advantageous to a multi-level pseudo random input signal for several reasons. In the identification experiments below we will use two such input signals, one for estimation and one for validation purposes, that were generated through a MATLAB input data generation tool (a GUI) kindly provided by the authors of the mentioned article.

We load this CSTR data and place it in two different IDDATA objects, ze for estimation and zv for validation purposes:

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'cstrdata'));
Ts = 0.1;    % 10 samples per hour!
ze = iddata(y1, u1, 0.1, 'Name', 'Estimation data');
ze.InputName = nlgr.InputName;
ze.InputUnit = nlgr.InputUnit;
ze.OutputName = nlgr.OutputName;
ze.OutputUnit = nlgr.OutputUnit;
ze.Tstart = 0;
ze.TimeUnit = 'hour';
ze.ExperimentName = 'Estimation';
zv = iddata(y2, u2, 0.1, 'Name', 'Validation data');
zv.InputName = nlgr.InputName;
zv.InputUnit = nlgr.InputUnit;
zv.OutputName = nlgr.OutputName;
zv.OutputUnit = nlgr.OutputUnit;
zv.Tstart = 0;
zv.TimeUnit = 'hour';
zv.ExperimentName = 'Validation';
```
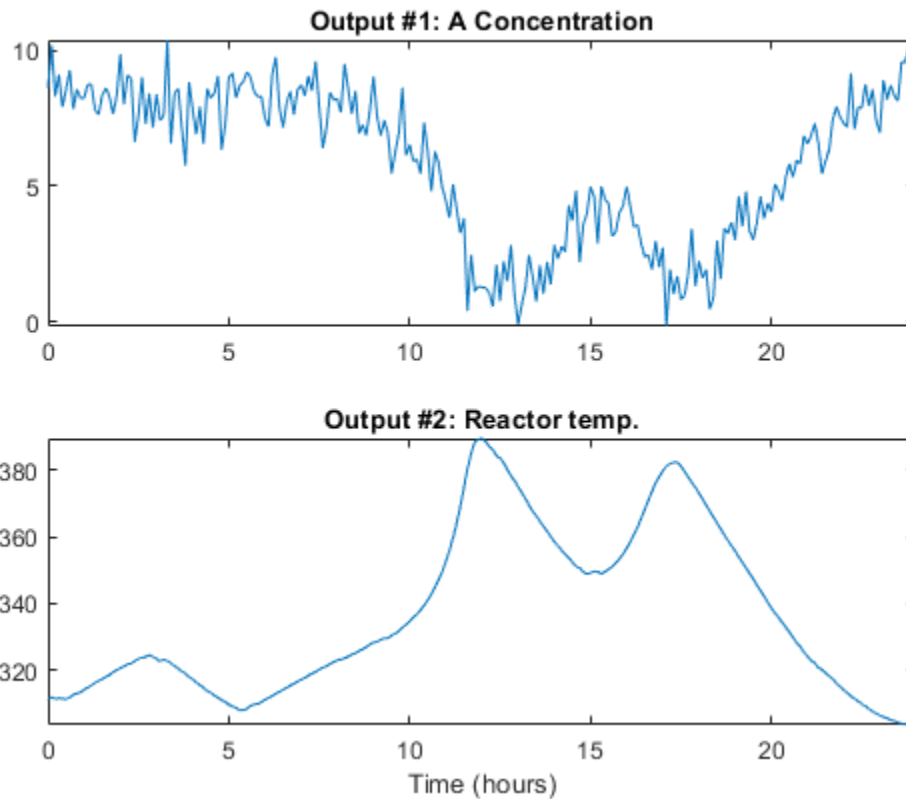
The inputs and outputs of the estimation data set ze are shown in two plots.

```
figure('Name', [ze.Name ': input data']);
for i = 1:ze.Nu
    subplot(ze.Nu, 1, i);
    plot(ze.SamplingInstants, ze.InputData(:,i));
    title(['Input #' num2str(i) ': ' ze.InputName{i}]);
    xlabel('');
    axis tight;
end
xlabel([ze.Domain ' (' ze.TimeUnit ')']);
```
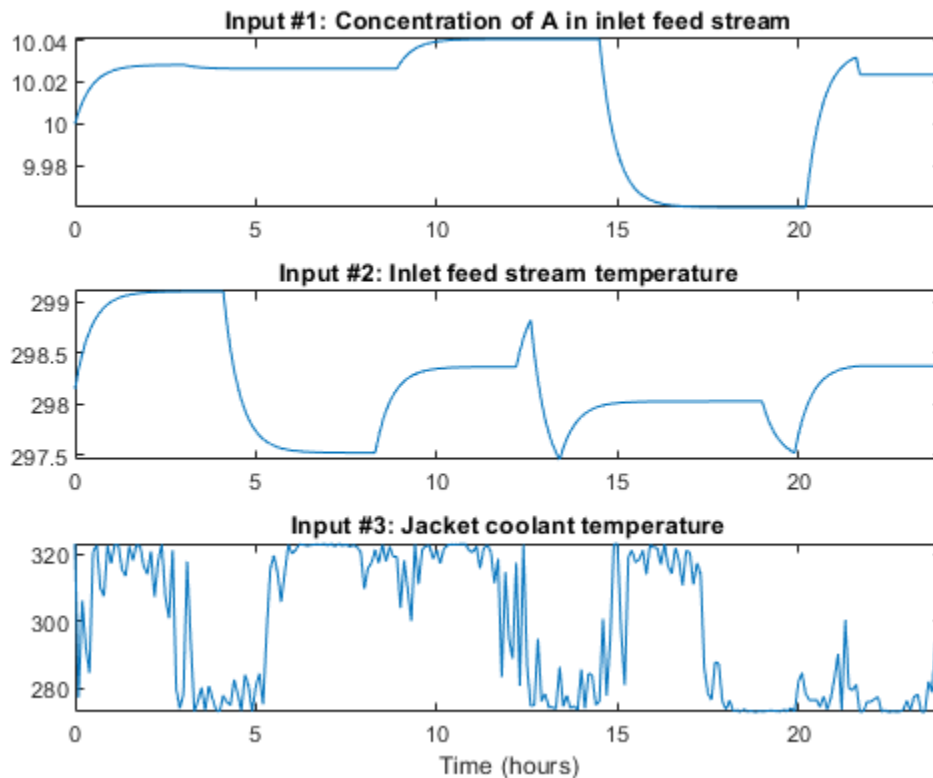

Figure 2: Estimation data set inputs to a CSTR.

```
figure('Name', [ze.Name ': output data']);
for i = 1:ze.Ny
    subplot(ze.Ny, 1, i);
    plot(ze.SamplingInstants, ze.OutputData(:,i));
    title(['Output #' num2str(i) ': ' ze.OutputName{i}]);
    xlabel('');
    axis tight;
end
xlabel([ze.Domain ' (' ze.TimeUnit ')']);
```

Figure 3: Estimation data set outputs from a CSTR.

Before we proceed with the identification experiments, we should mention that the generated input signals force the outputs of the CSTR to display a lot of the reactor nonlinearities (with temperature changes of around 80 degrees overall and causing some of the "ignition" phenomena of the reactor to be evident). Whereas this excites the reactor (typically good from an identification point of view), it is probably not the way in which engineers would like to operate a real-world reactor, especially not one that is as exothermic as this one. Using the guidelines described in Braun et al. (2002), one could then redesign the experiment before it is actually carried out. In this case, it would be interesting to try to reduce the duration of the experiment and use multi-sinusoidal input signals with shorter cycle lengths. The aim is to reduce the low-frequency content of the controllable input signal so as to reduce the variations in the reactor outputs.

Performance of the Initial CSTR Model

How good is the initial CSTR model? Let us investigate this by simulating the initial model using the input signals of ze and zv and compare the computed outputs with the true outputs (obtained by simulating the above IDNLGREY model using other parameters and adding some noise) contained in ze and zv, respectively.

```
clf
compare(ze, nlgr);
figure; compare(zv, nlgr);
```
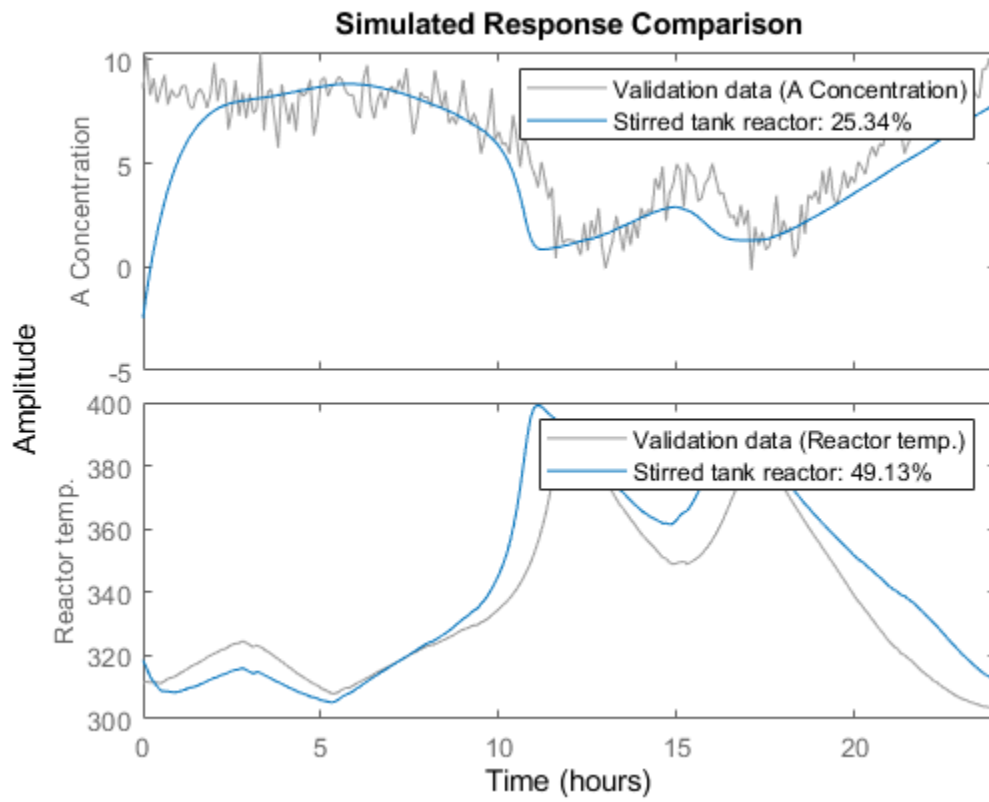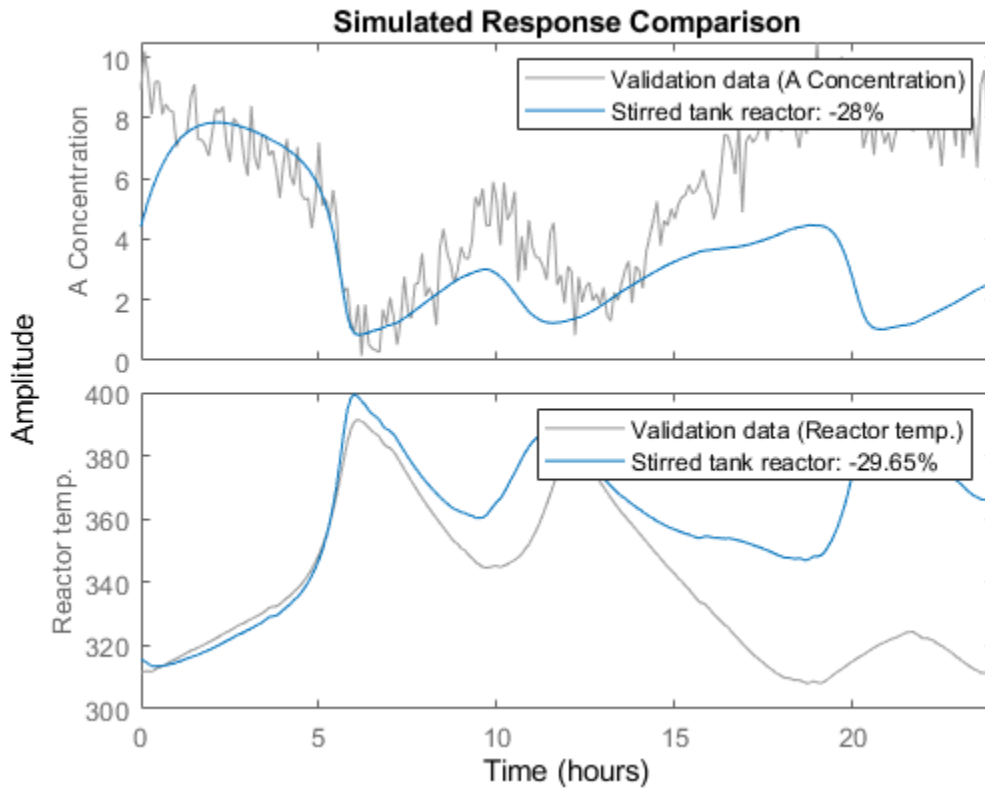
Figure 4: Comparison between the true outputs and the simulated outputs of the initial CSTR model.

Parameter Estimation

The agreement between true and simulated outputs of the initial CSTR model is decent. To further improve it, we estimate the four free model parameters as well as the initial state vector of the model by using the estimation data set ze. We instruct NLGREYEST to display iteration information and to perform at most 25 search iterations.

```
opt = nlgreyestOptions('Display', 'on');
opt.SearchOptions.MaxIterations = 25;
nlgr = nlgreyest(ze, nlgr, opt);
```

```
Nonlinear Grey Box Model Estimation
Data has 2 outputs, 3 inputs and 241 samples.
ODE Function: cstr_m
Number of parameters: 8
```

ESTIMATION PROGRESS

```
Algorithm: Trust-Region Reflective Newton
--------------------------------------------------------------
                              Norm of      First-order
Iteration      Cost           step         optimality
--------------------------------------------------------------
     0        112.577          -              -
     1        65.3797          10           5.55e+05
     2        15.8619          20           2.41e+05
     3        1.93491          40             197
     4        0.718559         80           5.51e+04
     5        0.341237         36.9         4.04e+04
     6        0.334771         1.99         5.96e+04
     7        0.334454         3.09         5.71e+04
     8        0.334454         0.772        5.71e+04
     9        0.334454         0.193        5.71e+04
```

RESULT

```
Termination condition: Change in parameters was less than the specified tolerance..
Number of iterations: 18, Number of function evaluations: 19

Status: Estimated using NLGREYEST
Fit to estimation data: [71.36;99.17]%, FPE: 0.0276874
```

■ Stop   Close

If desired, the search can always be continued via a second call to NLGREYEST (or PEM). This time NLGREYEST is instructed to not display any iteration information and to only carry out at most 5 more iterations.
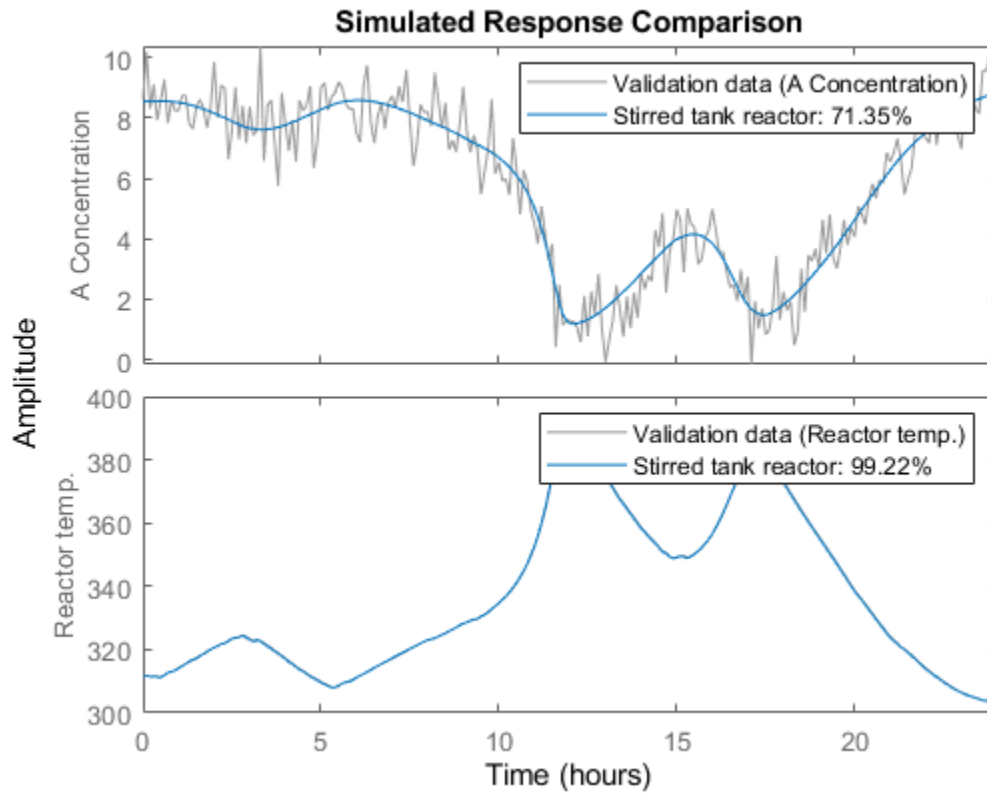
```
opt.Display = 'off';
opt.SearchOptions.MaxIterations = 5;
nlgr = nlgreyest(ze, nlgr, opt);
```
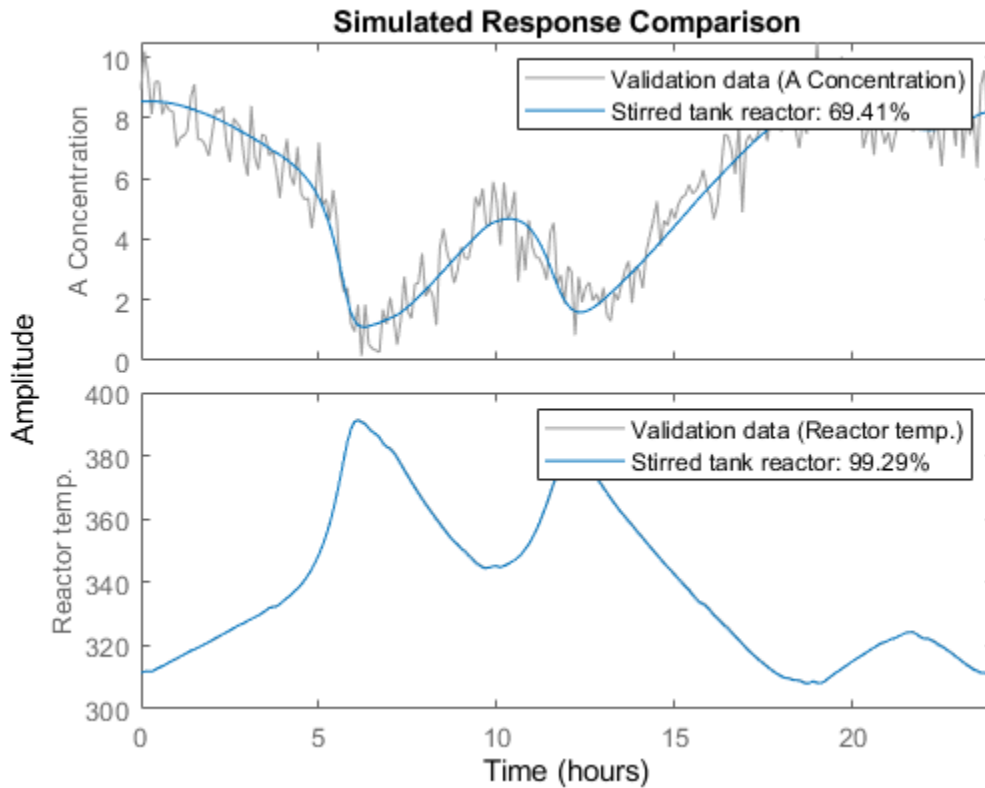
Performance of the Estimated CSTR Model

To evaluate the performance of the estimated model we once again use COMPARE:

```
compare(ze, nlgr);
figure; compare(zv, nlgr);
```

Figure 5: Comparison between the true outputs and the simulated outputs of the estimated CSTR model.

Visual inspection immediately reveals that the outputs of the estimated model are close to the true outputs, both for ze and zv. The improvement is especially significant for the validation data set, where the model fits have increased from negative values to around 70% and 99%, respectively, for the two model outputs.

Further information about the estimated CSTR model is next returned by PRESENT:

```
present(nlgr);
```

```
nlgr =

Continuous-time nonlinear grey-box model defined by 'cstr_m' (MATLAB file):

    dx/dt = F(t, u(t), x(t), p1, ..., p8)
     y(t) = H(t, u(t), x(t), p1, ..., p8) + e(t)

 with 3 input(s), 2 state(s), 2 output(s), and 4 free parameter(s) (out of 8).

 Inputs:
    u(1)  Concentration of A in inlet feed stream(t) [kgmol/m^3]
    u(2)  Inlet feed stream temperature(t) [K]
    u(3)  Jacket coolant temperature(t) [K]
 States:                                             Initial value
```

```
    x(1)  Concentration of A in reactor tank(t) [kgmol/m^3]   xinit@exp1   8.62914   (estimated)
    x(2)  Reactor temperature(t) [K]                          xinit@exp1   311.215   (estimated)
 Outputs:
    y(1)  A Concentration(t)
    y(2)  Reactor temp.(t)
 Parameters:                                                  Value      Standard Dev:
    p1   Volumetric flow rate (volume/time) [m^3/h]                           1
    p2   Volume in reactor [m^3]                                              1
    p3   Pre-exponential nonthermal factor [1/h]              3.55889e+07            1
    p4   Activation energy [kcal/kgmol]                          11853.9         0.07
    p5   Boltzmann's ideal gas constant [kcal/(kgmo..]           1.98589
    p6   Heat of reaction [kcal/kgmol]                             -5960
    p7   Heat capacity times density [kcal/(m^3*K)]               500.71          0.1
    p8   Overall heat transfer coefficient times tank area [kcal/(K*h)]  150.127    0.06

Name: Stirred tank reactor

Status:
Termination condition: Maximum number of iterations or number of function evaluations reached..
Number of iterations: 6, Number of function evaluations: 7

Estimated using Solver: ode45; Search: lsqnonlin on time domain data "Estimation data".
Fit to estimation data: [71.36;99.18]%
FPE: 0.02736, MSE: 0.6684
More information in model's "Report" property.

<a href="matlab:if exist('nlgr','var'), if isa(nlgr,'idnlgrey'), disp(' '); disp(get(nlgr)), else
```

Simulation of the Estimated CSTR Model in Simulink

An IDNLGREY model can also be imported and used within Simulink. The Simulink model "cstr_sim" imports the validation data set zv and passes its data to a Simulink IDNLGREY model block, which when simulated produces outputs that together with the used input signals are stored in the MATLAB workspace in the IDDATA object zsim. (The last five lines of code below are used to ensure that the proper model inputs are fed to the Simulink model. This is only needed if idnlgreydemo9 is run within a function, where access to zv and nlgr cannot be guaranteed.)

```
open_system('cstr_sim');
if ~evalin('base', 'exist(''zv'', ''var'')')
    cstrws = get_param(bdroot, 'modelworkspace');
    cstrws.assignin('zv', zv);
    cstrws.assignin('nlgr', nlgr);
end
```
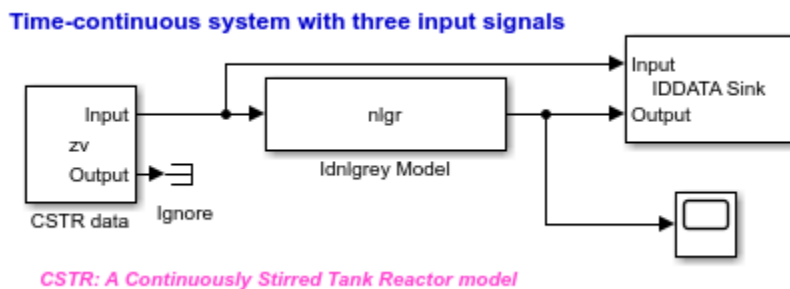

Figure 6: Simulink model containing the estimated CSTR model.

The generic IDNLGREY Simulink library block is found in the standard system identification Simulink library and can be copied to and used in any Simulink model. For example, in the CSTR case it could very well be used in a closed-loop control arrangement.

The IDNLGREY block must be configured before it is simulated. This is done by entering the MATLAB workspace variable holding the IDNLGREY model (here nlgr) or by defining a proper IDNLGREY model object using the idnlgrey constructor in the parameter edit box labeled "IDNLGREY model". Here it is also possible to specify the initial state vector to use (default is the internally stored initial state vector of the specified IDNLGREY object).

An IDNLGREY model object stores the properties specifying the setup of the differential/difference equation solver used by SIM, PREDICT, NLGREYEST and so on (see nlgr.SimulationOptions for options controlling the simulation of the model). In Simulink, these settings are always overridden so that the options of the Simulink specified solver are used. If the IDNLGREY model object specifies and uses a different solver, then the simulation result obtained in Simulink might be different to that obtained with IDNLGREY/SIM in MATLAB. An example illustrating this is provided in the example "idnlgreydemo10".

With the solver options settled, we can next perform a command prompt simulation of the cstr_sim Simulink model (which here will be conducted for the estimated CSTR model). (The evalin call is needed to retrieve zsim in case idnlgreydemo9 is run within a function.)

```matlab
sim('cstr_sim');
if ~exist('zsim', 'var')
    zsim = evalin('base', 'zsim');
end
zsim.InputName = nlgr.InputName;
zsim.InputUnit = nlgr.InputUnit;
zsim.OutputName = nlgr.OutputName;
zsim.OutputUnit = nlgr.OutputUnit;
zsim.TimeUnit = 'hour';
```

Let us finally conclude the example by plotting the Simulink obtained simulation result.

```matlab
figure('Name', 'Simulink simulation result of estimated CSTR model');
for i = 1:zsim.Ny
   subplot(zsim.Ny, 1, i);
   plot(zsim.SamplingInstants, ze.OutputData(:,i));
   title(['Output #' num2str(i) ': ' zsim.OutputName{i}]);
   xlabel('');
   axis tight;
end
xlabel([zsim.Domain ' (' zsim.TimeUnit ')']);
```

Figure 7: Outputs obtained by simulating the estimated CSTR model in Simulink.

Conclusions

This tutorial has covered modeling and identification of a non-adiabatic continuous stirred tank reactor. In particular, it was illustrated how to import and use an IDNLGREY model within Simulink.

References

[1] Braun, M.W., R. Ortiz-Mojica and D.E. Rivera, "Application of minimum crest factor multisinusoidal signals for 'plant-friendly' identification of nonlinear process systems," in *Control Engineering Practice*, Vol. 10, No. 3, 2002, pp. 301-313.

Classical Pendulum: Some Algorithm-Related Issues

This example shows how the estimation algorithm choices may impact the results for a nonlinear grey box model estimation. We use data produced by a nonlinear pendulum system, which is schematically shown in Figure 1. We show in particular how the choice of differential equation solver impacts the results.
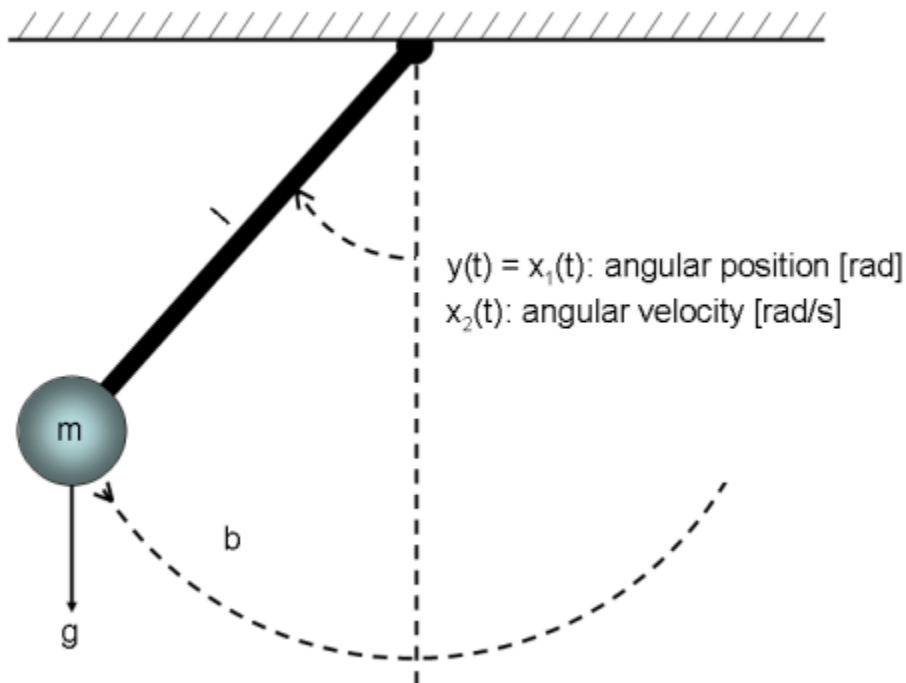
Figure 1: Schematic view of a classical pendulum.

Output Data

We start our modeling tour by loading the output data (time-series data). The data contains one output, y, which is the angular position of the pendulum [rad]. The angle is zero when the pendulum is pointing downwards, and it is increasing anticlockwise. There are 1001 (simulated) samples of data points and the sample time is 0.1 seconds. The pendulum is affected by a constant gravity force, but no other exogenous force affects the motion of the pendulum. To investigate this situation, we create an IDDATA object:

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'pendulumdata'));
z = iddata(y, [], 0.1, 'Name', 'Pendulum');
z.OutputName = 'Pendulum position';
z.OutputUnit = 'rad';
z.Tstart = 0;
z.TimeUnit = 's';
```

The angular position of the pendulum (the output) is shown in a plot window.

```
figure('Name', [z.Name ': output data']);
plot(z);
```

Figure 2: Angular position of the pendulum (output).

13-233

Pendulum Modeling

The next step is to specify a model structure for the pendulum system. The dynamics of it has been studied in numerous books and articles and are well understood. If we choose x1(t) as the angular position [rad] and x2(t) as the angular velocity [rad/s] of the pendulum, then it is rather straightforward to set up a state-space structure of the following kind:

```
d/dt x1(t) = x2(t)
d/dt x2(t) = -(g/l)*sin(x1(t)) - (b/(m*l^2))*x2(t)

       y(t) = x1(t)
```

having parameters (or constants)

```
g - the gravity constant [m/s^2]
l - the length of the rod of the pendulum [m]
b - viscous friction coefficient [Nms/rad]
m - the mass of the bob of the pendulum [kg]
```

We enter this information into the MATLAB file pendulum_m.m, with contents as follows:

```
function [dx, y] = pendulum_m(t, x, u, g, l, b, m, varargin)
%PENDULUM_M  A pendulum system.

% 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.
     ];
```

The next step is to create a generic IDNLGREY object for describing the pendulum. We also enter information about the names and the units of the inputs, states, outputs and parameters. Owing to the physical reality all model parameters must be positive and this is imposed by setting the 'Minimum' property of all parameters to the smallest recognizable positive value in MATLAB®, eps(0).

```
FileName      = 'pendulum_m';        % File describing the model structure.
Order         = [1 0 2];             % Model orders [ny nu nx].
Parameters    = [9.81; 1; 0.2; 1];   % Initial parameters.
InitialStates = [1; 0];              % Initial initial states.
Ts            = 0;                   % Time-continuous system.
nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts);
nlgr.OutputName = 'Pendulum position';
nlgr.OutputUnit = 'rad';
nlgr.TimeUnit = 's';
nlgr = setinit(nlgr, 'Name', {'Pendulum position' 'Pendulum velocity'});
nlgr = setinit(nlgr, 'Unit', {'rad' 'rad/s'});
nlgr = setpar(nlgr, 'Name', {'Gravity constant' 'Length' ...
                    'Friction coefficient' 'Mass'});
nlgr = setpar(nlgr, 'Unit', {'m/s^2' 'm' 'Nms/rad' 'kg'});
nlgr = setpar(nlgr, 'Minimum', {eps(0) eps(0) eps(0) eps(0)});   % All parameters > 0.
```

Performance of Three Initial Pendulum Models

Before trying to estimate any parameter we simulate the system with the guessed parameter values. We do this for three of the available solvers, Euler forward with fixed step length (ode1), Runge-Kutta

23 with adaptive step length (ode23), and Runge-Kutta 45 with adaptive step length (ode45). The outputs obtained when using these three solvers are shown in a plot window.

```
% A. Model computed with first-order Euler forward ODE solver.
nlgref = nlgr;
nlgref.SimulationOptions.Solver = 'ode1';        % Euler forward.
nlgref.SimulationOptions.FixedStep = z.Ts*0.1;   % Step size.

% B. Model computed with adaptive Runge-Kutta 23 ODE solver.
nlgrrk23 = nlgr;
nlgrrk23.SimulationOptions.Solver = 'ode23';     % Runge-Kutta 23.

% C. Model computed with adaptive Runge-Kutta 45 ODE solver.
nlgrrk45 = nlgr;
nlgrrk45.SimulationOptions.Solver = 'ode45';     % Runge-Kutta 45.

compare(z, nlgref, nlgrrk23, nlgrrk45, 1, ...
    compareOptions('InitialCondition', 'model'));
```

Figure 3: Comparison between true output and the simulated outputs of the three initial pendulum models.

As can be seen, the result with the Euler forward method (with a finer grid than what is used by default) differs significantly from the results obtained with the Runge-Kutta solvers. In this case, it turns out that the Euler forward solver produces a rather good result (in terms of model fit), whereas the outputs obtained with the Runge-Kutta solvers are a bit limited. However, this is somewhat deceiving, as will be evident later on, because the initial value of b, the viscous friction coefficient, is twice as large as in reality.

Parameter Estimation

The gravitational constant, g, the length of the rod, l, and the mass of the bob, m, can easily be measured or taken from a table without estimation. However, this is typically not the case with friction coefficients, which often must be estimated. We therefore fix the former three parameters to g = 9.81, l = 1, and m = 1, and consider only b as a free parameter:

```
nlgref = setpar(nlgref, 'Fixed', {true true false true});
nlgrrk23 = setpar(nlgrrk23, 'Fixed', {true true false true});
nlgrrk45 = setpar(nlgrrk45, 'Fixed', {true true false true});
```

Next we estimate b using the three differential equation solvers. We start with the Euler forward based model structure.

```
opt = nlgreyestOptions('Display', 'on');
tef = clock;
nlgref = nlgreyest(z, nlgref, opt);   % Perform parameter estimation.
tef = etime(clock, tef);
```

Then we continue with the model based on the Runge-Kutta 23 solver (ode23).

```
trk23 = clock;
nlgrrk23 = nlgreyest(z, nlgrrk23, opt);   % Perform parameter estimation.
trk23 = etime(clock, trk23);
```

We finally use the Runge-Kutta 45 solver (ode45).

```
trk45 = clock;
nlgrrk45 = nlgreyest(z, nlgrrk45, opt);   % Perform parameter estimation.
trk45 = etime(clock, trk45);
```

Performance of Three Estimated Pendulum Models

The results when using these three solvers are summarized below. The true value of b is 0.1, which is obtained with ode45. ode23 returns a value quite close to 0.1, while ode1 returns a value quite a distance from 0.1.

```
disp('           Estimation time   Estimated b value');
fprintf('   ode1 :  %3.1f                 %1.3f\n', tef, nlgref.Parameters(3).Value);
fprintf('   ode23:  %3.1f                 %1.3f\n', trk23, nlgrrk23.Parameters(3).Value);
fprintf('   ode45:  %3.1f                 %1.3f\n', trk45, nlgrrk45.Parameters(3).Value);
```

However, this does not mean that the simulated outputs differ that much from the actual output, because the errors produced by the different differential equation solvers are typically accounted for in the estimation procedure. This is a fact that readily can be seen by simulating the three estimated pendulum models.

```
compare(z, nlgref, nlgrrk23, nlgrrk45, 1, ...
   compareOptions('InitialCondition', 'model'));
```

Figure 4: Comparison between true output and the simulated outputs of the three estimated pendulum models.

As expected given this figure, the Final Prediction Error (FPE) criterion values of these models are also rather close to each other:

```
fpe(nlgref, nlgrrk23, nlgrrk45);
```

Based on this we conclude that all three models are able to capture the pendulum dynamics, but the Euler forward based model reflects the underlying physics quite poorly. The only way to increase its "physics" performance is to decrease the step length of the solver, but this also means that the solution time increases significantly. Our experiments indicate that the Runge-Kutta 45 solver is the best solver for non-stiff problems when taking both accuracy and computational speed into account.

Conclusions

The Runge-Kutta 45 (ode45) solver often returns high quality results relatively fast, and is therefore chosen as the default differential equation solver in IDNLGREY. Typing "help idnlgrey.SimulationOptions" provides some more details about the available solvers and typing "help nlgreyestOptions" provides details on the various estimation options that can be used for IDNLGREY modeling.

Additional Information

For more information on identification of dynamic systems with System Identification Toolbox™ visit the System Identification Toolbox product information page.

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 λ as its virtual input. The true measured input size of such models is zero, and their governing equation takes the form:

$$y(t) = He(t)$$

Here, $y(t)$ is the signal being modeled and H is the transfer function that represents the relationship between $y(t)$ and $e(t)$.

The multivariate power spectrum Φ of the time series $y(t)$ is given by:

$$\Phi = H(\Lambda Ts)H'$$

Here Λ is the noise variance matrix 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`), state-space (`idss`, or `idgrey`) model. Some particular types of models are parametric autoregressive (AR), autoregressive and moving average (ARMA), and autoregressive models with integrated moving average (ARIMA). For nonlinear time series models, the toolbox supports nonlinear ARX models.

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.

To represent a time series vector or a matrix `s` as an `iddata` object, use the following syntax:

```
y = iddata(s,[],Ts);
```

The following example illustrates a 4th order autoregressive model estimation for the time series data `z9` that is stored in file `iddata9`.

```
load iddata9 z9
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`.

See Also

More About

- "Identify Time Series Models at the Command Line" on page 14-12
- "Estimate Time-Series Power Spectra" on page 14-4
- "Analyze Time-Series Models" on page 14-30
- "Identifying Nonlinear ARX Models" on page 11-17
- "Estimate Nonlinear Grey-Box Models" on page 13-25
- "Time Series Analysis"

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 2-9.

For more information about preparing data for modeling, see "Ways to Prepare Data for System Identification" on page 2-5.

If your data is already in the MATLAB workspace, you can import it directly into the System Identification app. 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 app — When you import scalar or multiple-output time series data into the app, leave the **Input** field empty. For more information about importing data, see "Represent Data".

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.

Estimate Time-Series Power Spectra

A frequency-response model encapsulates the frequency response of a linear system evaluated over a range of frequency values. When the data contains both input and output channels, the frequency-response model describes the steady-state response of the system to sinusoidal inputs. Time-series data contains no input channel. The frequency response of a time series model reduces to a spectral representation of the output data. This output data implicitly includes the effects of the unmeasured input noise.

For a discrete-time system sampled for both inputs and outputs with a time interval T, the transfer function $G(z)$ relates the Z-transforms of the input $U(z)$ and output $Y(z)$:

$$Y(z) = G(z)U(z) + H(z)E(z)$$

$H(z)$ represents the noise transfer functions for each output and $E(z)$ is the Z-transform of the additive disturbance $e(t)$ with variance Λ.

For a time-series model, this equation reduces to:

$$Y(z) = H(z)E(z)$$

In this case, $E(z)$ represents the assumed, but unmeasured, white-noise input disturbance. The single-output noise spectrum Φ in the presence of disturbance noise with scalar variance λ is defined as:

$$\Phi_v(\omega) = \lambda T |H(e^{i\omega T})|^2$$

The equivalent multi-output noise power spectrum can be given as:

$$\Phi_v(\omega) = T H(e^{i\omega T}) \Lambda H'(e^{-i\omega T})$$

Here, Λ is the variance vector with a length equal to the number of outputs.

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. These functions return estimated models that are represented by `idfrd` model objects, which contain the spectral data in the property `SpectrumData` and the spectral variance in the property `NoiseCovariance`. For multiple-output data, `SpectrumData` contains power spectra of each output and the cross-spectra between each output pair.

Commands for Estimating and Comparing 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.
`spectrum`	Estimates and plots the output power spectrum of time series models.

For example, suppose y is time series data. Estimate the power spectrum g and the periodogram p using spa and etfe. Plot the models together with three standard deviation confidence intervals by using spectrum.

```
g = spa(y);
p = etfe(y);
spectrum(g,p);
```

For a more detailed example of spectral estimation, see "Identify Time Series Models at the Command Line" on page 14-12. For more information about the individual commands, see the corresponding reference pages.

Estimate Time-Series Power Spectra Using the App

You must have already imported your data into the app.

To estimate time series spectral models in the System Identification app:

1   In the System Identification app, select **Estimate > Spectral Models** to open the Spectral Model dialog box.

2   In the **Method** list, select the spectral analysis method you want to use. For information about each method, see "Selecting the Method for Computing Spectral Models" on page 9-7.

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 9-8. 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 app.

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 app. For more information about working with this plot, see "Noise Spectrum Plots" on page 17-64.

To export the model to the MATLAB workspace, drag it to the **To Workspace** rectangle in the System Identification app. You can view the power spectrum and the confidence intervals of the resulting `idfrd` model object using the `spectrum` command.

See Also
`idfrd | spa | spafdr | etfe`

More About

Estimate AR and ARMA Models

AR and ARMA models are autoregressive parametric models that have no measured inputs. These models operate on time series data.

- The AR model contains a single polynomial A that operates on the measured output. For a single-output signal $y(t)$, the AR model is given by the following equation:

    $A(q)y(t) = e(t)$

- The ARMA model adds a second polynomial C that calculates the moving average of the noise error. 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 AR and ARMA model structures are special cases of the more general ARX and ARMAX model structures, which do provide for measured inputs. You can estimate AR and ARMA models at the command line and in the app.

For information about:

- Time series models, see "What Are Time Series Models?" on page 14-2
- Polynomial models, see "What Are Polynomial Models?" on page 6-2
- Autoregressive time series models containing noise integration, see "Estimate ARIMA Models" on page 14-18

Estimate AR and ARMA Models at the Command Line

Estimate AR and ARMA models at the command line by using `ar`, `arx`, `ivar`, or `armax` with estimation data that contains only output measurements. These functions return estimated models that are represented by `idpoly` model objects.

Selected Commands for Estimating Polynomial AR and ARMA Time-Series Models

Function	Description
`ar`	Noniterative, least-squares method to estimate linear, discrete-time, single-output AR models. Provides algorithmic options including lattice-based approaches and the Yule-Walker covariance approach. Example: `sys = ar(y,na)` estimates an AR model `sys` of polynomial order `na` from the scalar time series `y`.
`arx`	Noniterative, least-squares method for estimating linear AR models. Supports multiple outputs. Assumes white noise. Example: `sys = arx(y,na)` estimates an AR model from the multiple-output time series `y`.
`ivar`	Noniterative, instrumental variable method for estimating single-output AR models. Insensitive to noise color. Example: `sys = ivar(y,na)` estimates an AR model using the instrumental variable method for the scalar time series `y`.
`armax`	Iterative prediction-error method to estimate linear ARMA models. Example: `sys = armax(y,[na nc])` of estimates an ARMA model of polynomial orders `na` and `nc` from the time series `y`.

For more detailed usage information and examples, as well as information on other models that these functions can estimate, see `ar`, `arx`, `ivar`, and `armax`.

Estimate AR and ARMA Time Series Models in the App

Before you begin, complete the following steps:

- Prepare the data, as described in "What Are Time Series Models?" on page 14-2.
- Estimate model order, as described in "Preliminary Step – Estimating Model Orders and Input Delays" on page 6-8.
- For multiple-output AR models only, specify the model-order matrix in the MATLAB workspace before estimation, as described in "Polynomial Sizes and Orders of Multi-Output Polynomial Models" on page 6-21.

Estimate AR and ARMA models using the System Identification app by following these steps.

1 In the System Identification app, select **Estimate** > **Polynomial Models** to open the Polynomial 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 Polynomial Models dialog box to correspond with this model structure.

3   In the **Orders** field, specify the model orders.

  - 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 6-21. Alternatively, enter the name of the matrix `NA` in the MATLAB Workspace browser that stores model orders, which is `Ny`-by-`Ny`.

  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 6-25.

5   Select **Add noise integration** if you want to include an integrator in noise source $e(t)$. This selection changes an AR model into an ARI model ($Ay = \dfrac{e}{1 - q^{-1}}$) and an ARMA model into an ARIMA model ($Ay = \dfrac{C}{1 - q^{-1}}e(t)$).

6   In the **Name** field, edit the name of the model or keep the default. The name of the model must 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 6-24.

  If you get an inaccurate fit, try setting a specific method for handling initial states rather than specifying automatic selection.

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.

  If you do not want the algorithm to estimate uncertainty, select `None`. Skipping uncertainty computation can reduce computation time for complex models and large data sets.

9   Click **Regularization** to obtain regularized estimates of model parameters. Specify regularization constants in the Regularization Options dialog box. For more information, see "Regularized Estimates of Model Parameters" on page 1-34.

10   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 — 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 — Actual versus expected improvements in the fit.

11   Click **Estimate** to add this model to the Model Board in the System Identification app.

12   For the prediction-error method, only, to stop the search and save the results after the current iteration completes, 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.

13   To plot the model, select the appropriate check box in the **Model Views** area of the System Identification app.

You can export the model to the MATLAB workspace for further analysis by dragging it to the **To Workspace** rectangle in the System Identification app.

See Also

ar | arx | armax | ivar

More About

- "What Are Model Objects?" on page 1-3
- "What Are Time Series Models?" on page 14-2
- "What Are Polynomial Models?" on page 6-2
- "Identify Time Series Models at the Command Line" on page 14-12

Estimate State-Space Time Series Models

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 sample time 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 7-2.

Estimate 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 the Command Line

This example shows how to simulate a time series and use parametric and nonparametric methods to estimate and compare time series models.

Generate Model and Simulate Model Output

Generate time series data by creating and simulating an autoregressive (AR) polynomial model ts_orig of the form $y_k = a_1 y_{k-1} + a_2 y_{k-2} + e_k$, where e_k is random Gaussian noise. This noise represents an unmeasured input to the model. Since the model is a time series, there are no measured inputs.

Before calculating e_k, initialize the random number generator seed so that the noise values are repeatable.

```
ts_orig = idpoly([1 -1.75 0.9]);
rng('default')
e = idinput(300,'rgs');
```

Simulate the observed output y_obs of this model and convert y_obs to an iddata object y with the default sample time of one second. Plot the model output together with the input noise.

```
y_obs = sim(ts_orig,e);
y = iddata(y_obs);

plot(e)
hold on
plot(y_obs)
title('Input Noise and Original Model Output')
legend('RGS Noise','Model Output')
hold off
```
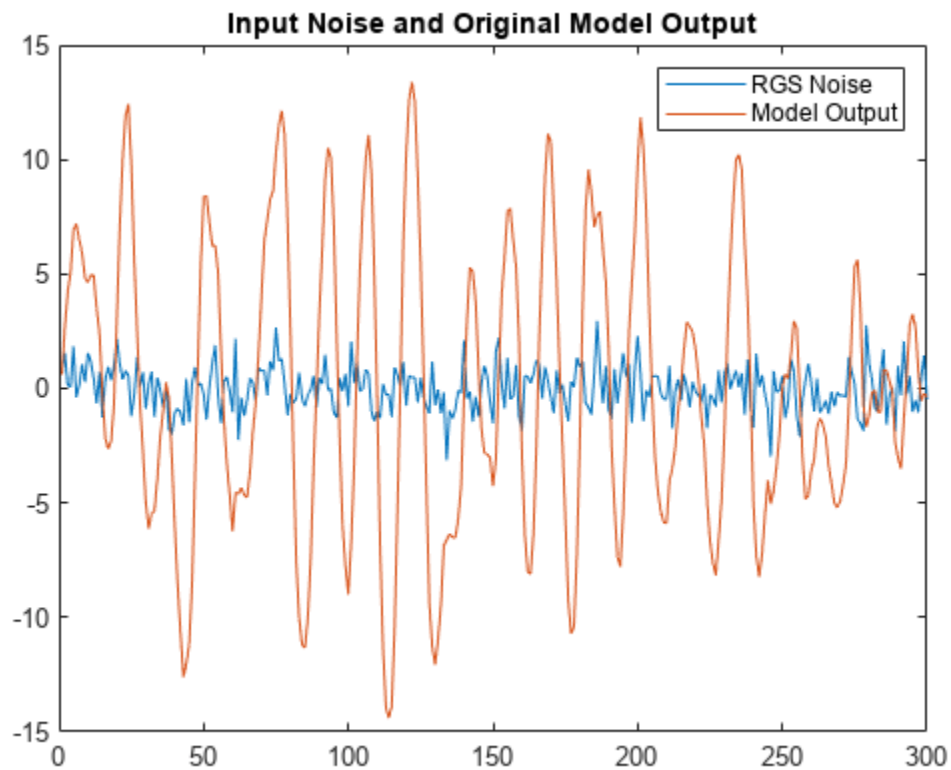
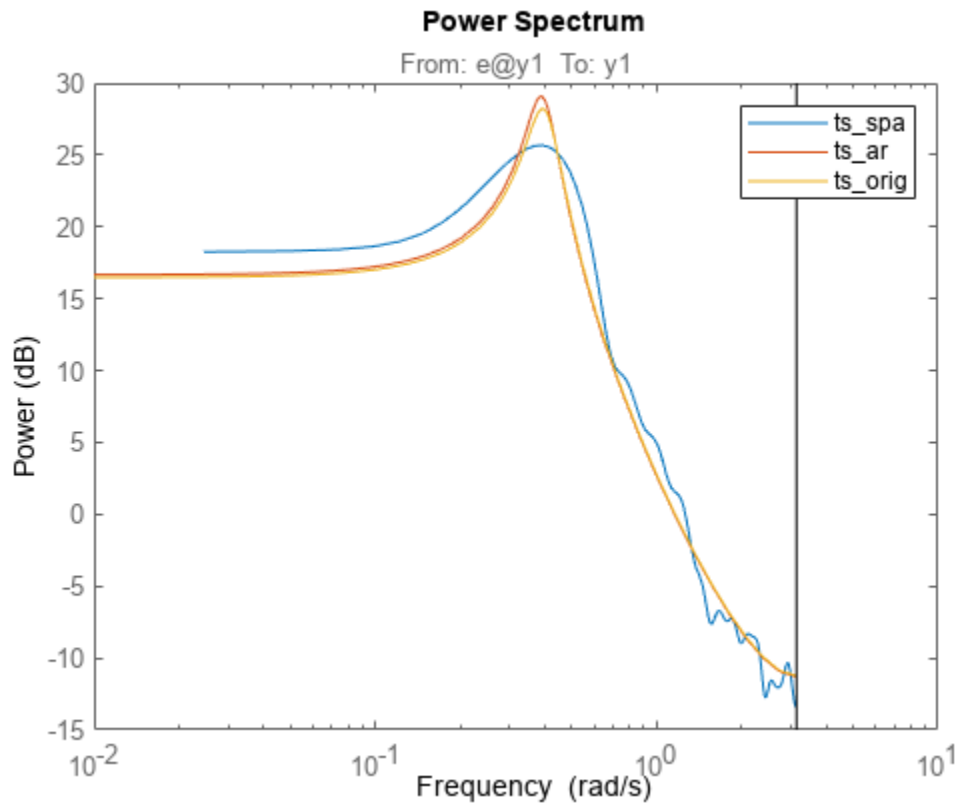**Estimate Model and Compare Spectra**

The functions `etfe` and `spa` provide two nonparametric techniques for performing spectral analysis. Compare the estimated power spectra from `etfe` and `spa` to the original model.

```
ts_etfe = etfe(y);
ts_spa = spa(y);
spectrum(ts_etfe,ts_spa,ts_orig);
legend('ts_{etfe}','ts_{spa}','ts_{orig}')
```
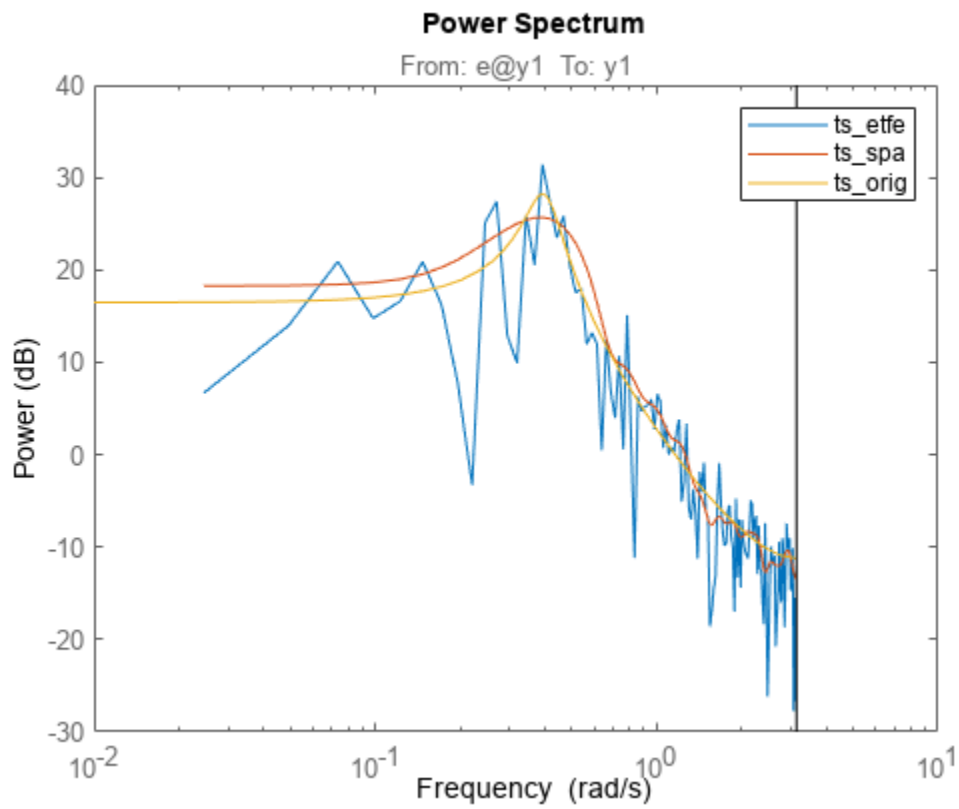
Now estimate a parametric model using the AR structure. Estimate a second-order AR model and compare its spectrum with the original model and the `spa` estimate.

```
ts_ar = ar(y,2);
spectrum(ts_spa,ts_ar,ts_orig);
legend('ts_{spa}', 'ts_{ar}', 'ts_{orig}')
```

The AR model spectrum fits the original model spectrum more closely than the nonparametric models.

Estimate and Compare Covariance

Calculate the covariance function for the original model and the AR model by convolving each model output with itself.

```
ir_orig = sim(ts_orig,[1;zeros(24,1)]);
Ry_orig = conv(ir_orig,ir_orig(25:-1:1));
ir_ar = sim(ts_ar,[1;zeros(24,1)]);
Ry_ar = conv(ir_ar,ir_ar(25:-1:1));
```

Also estimate the covariance Ry directly from the observed outputs y using `xcorr`.

```
Ry = xcorr(y.y,24,'biased');
```

Plot and compare the original and estimated covariances.

```
plot(-24:24'*ones(1,3),[Ry_orig,Ry_ar,Ry]);
legend('Ry_{orig}', 'Ry_{ar}', 'Ry')
```
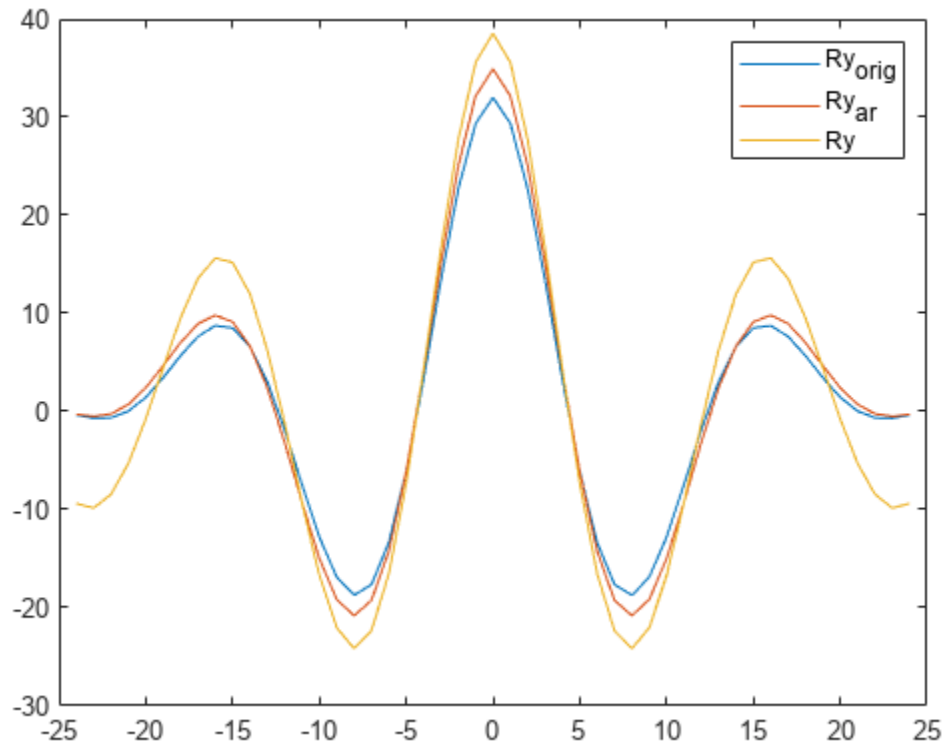
The covariance of the estimated AR model, `Ry_ar`, is closer to the original covariance `Ry_orig`.

Predict and Compare Model Outputs

Compare the three-step prediction accuracy, or fit percentage, for the original model and the AR model, using the function `compare`. Here, `compare` computes the predicted response of the `ts_orig` and `ts_ar` models with the original model output data `y`, assuming unmeasured input e_k is zero. The fourth argument, 3, is the number of steps to predict.

```
compare(y,ts_orig,ts_ar,3);
```

3-Step Predicted Response Comparison
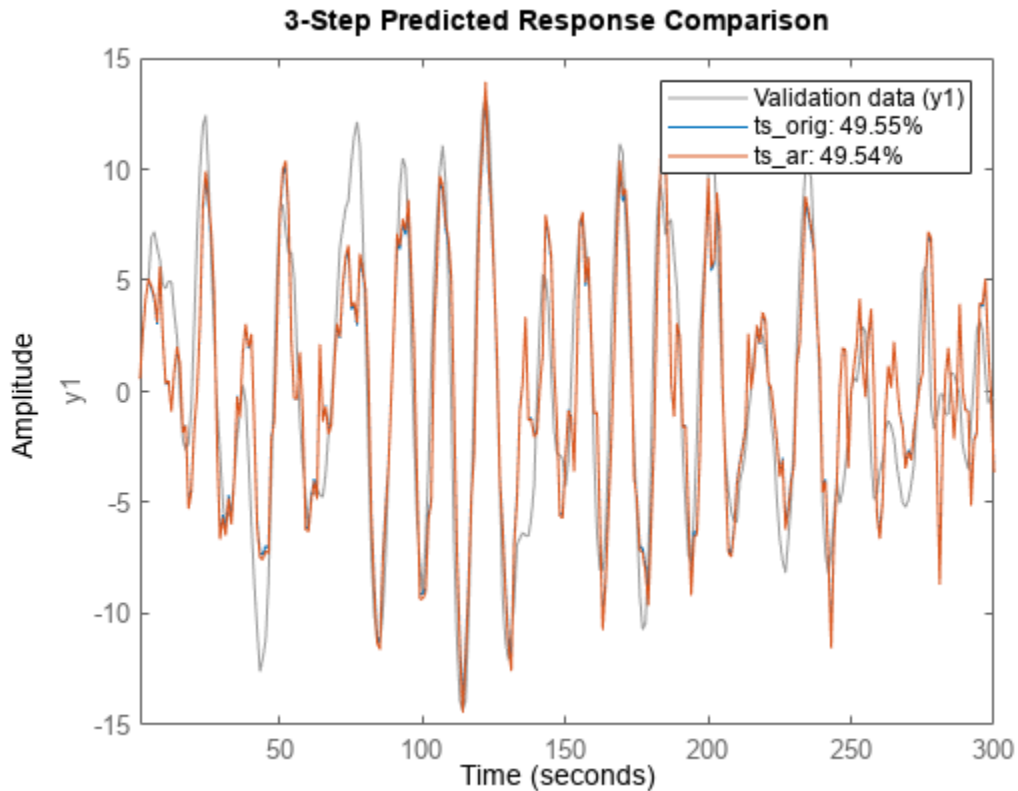
The percentages in the legend are the fit percentages, which represent goodness of fit. The prediction accuracy is far from 100% even for the original model because the unmeasured model input e_k is not accounted for in the prediction process. The fit value for the estimated AR model is close to the original model, indicating that the AR model is a good estimate.

See Also

`spectrum | etfe | spa | ar`

Related Examples

- "Estimate Time-Series Power Spectra" on page 14-4

More About

- "What Are Time Series Models?" on page 14-2

Estimate ARIMA Models

This example shows how to estimate autoregressive integrated moving average (ARIMA) models.

Models of time series containing non-stationary trends (seasonality) are sometimes required. One category of such models are the 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 is expressed as

$$A(q)y(t) = \frac{C(q)}{(1 - q^{-1})}e(t)$$

where the term $\dfrac{1}{1 - q^{-1}}$ 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 introduce integrators into the noise source `e(t)`. You do this by using the `IntegrateNoise` parameter in the estimation command.

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 for 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);
% 5 step ahead prediction
compare(y,model,5)
```

5-Step Predicted Response Comparison
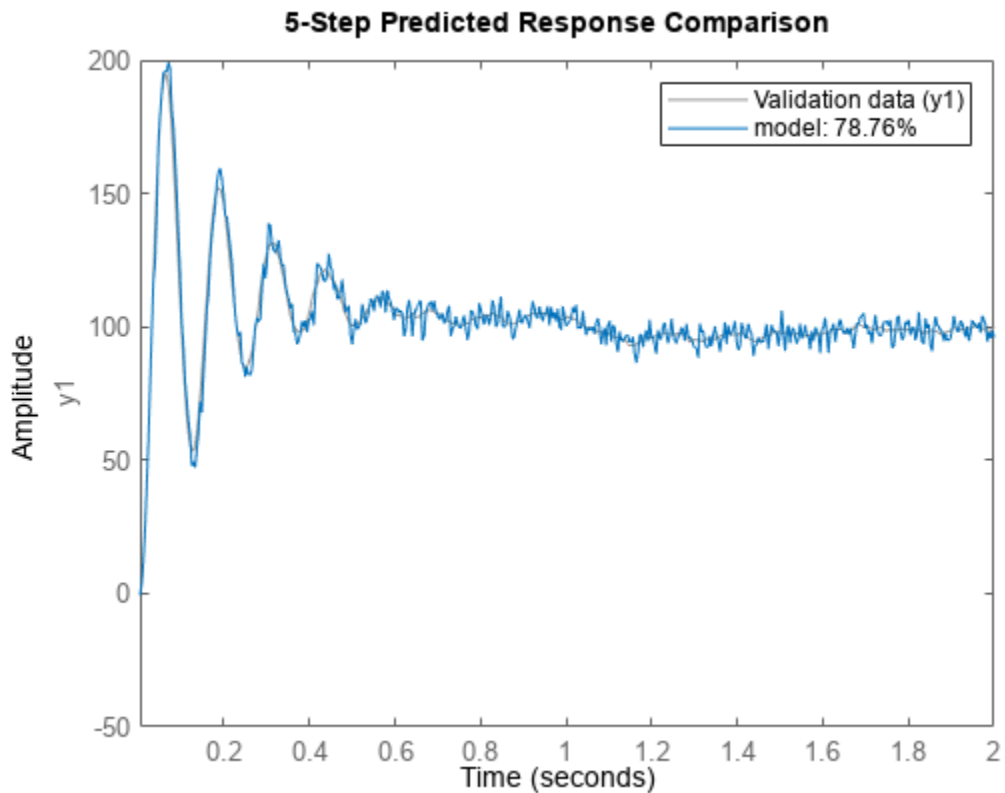
Estimate a multivariate time-series model such that the noise integration is present in only one of the two time series.

```
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)
```
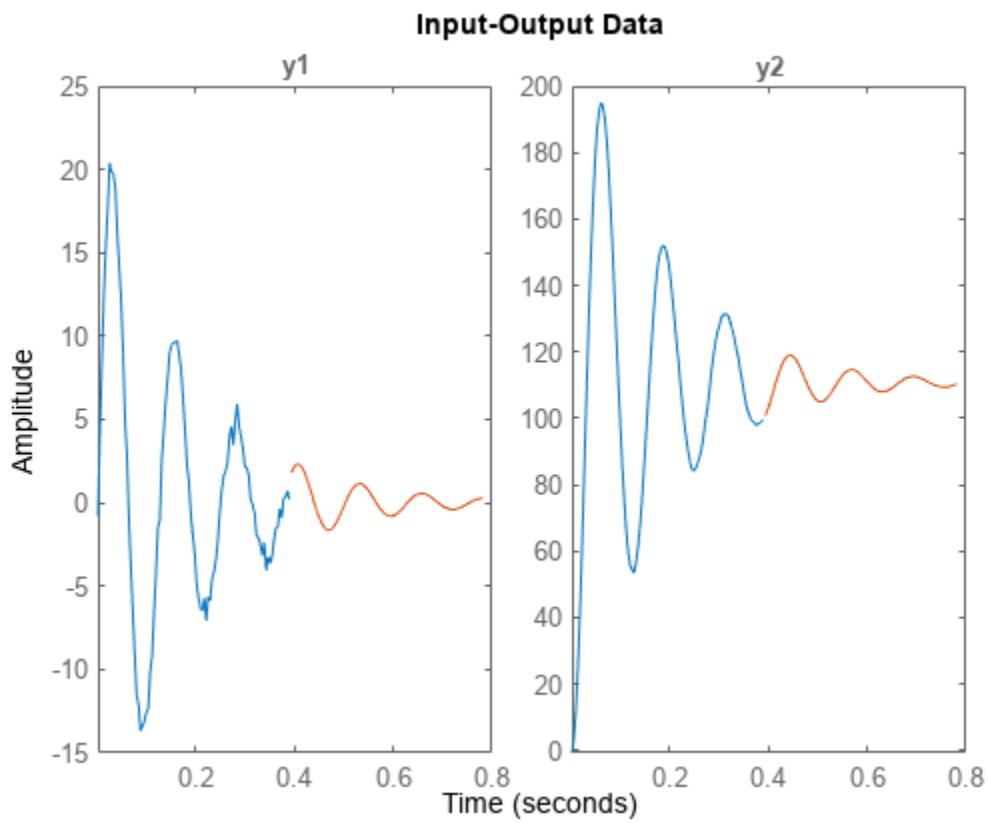
If the outputs were coupled (na was not a diagonal matrix), the situation will be more complex and simply adding an integrator to the second noise channel will not work.

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.')
```
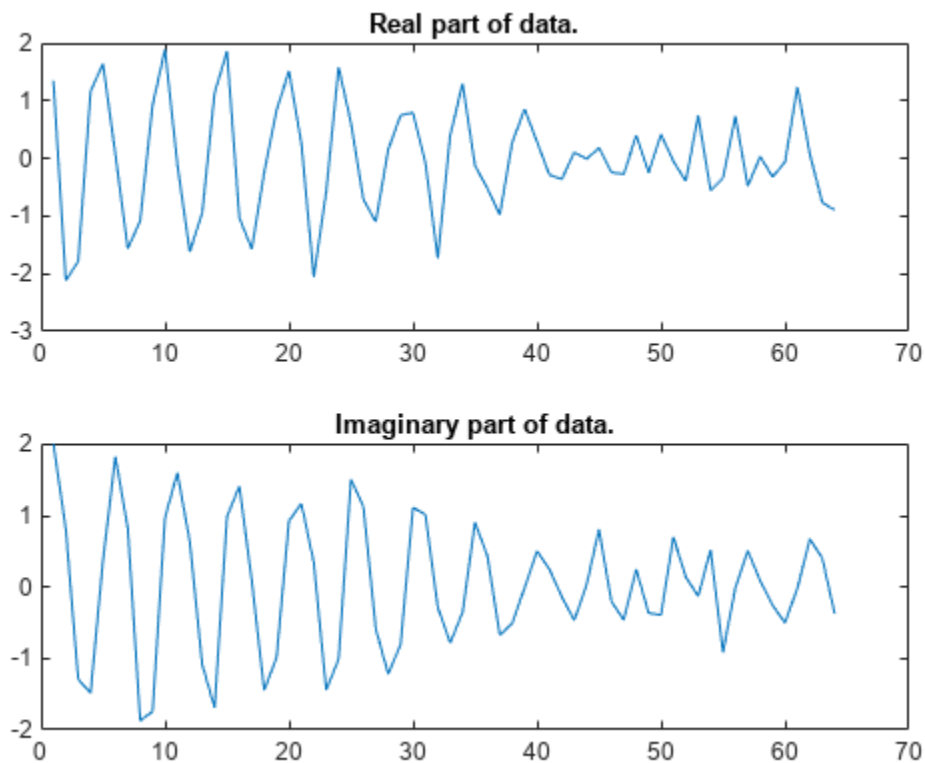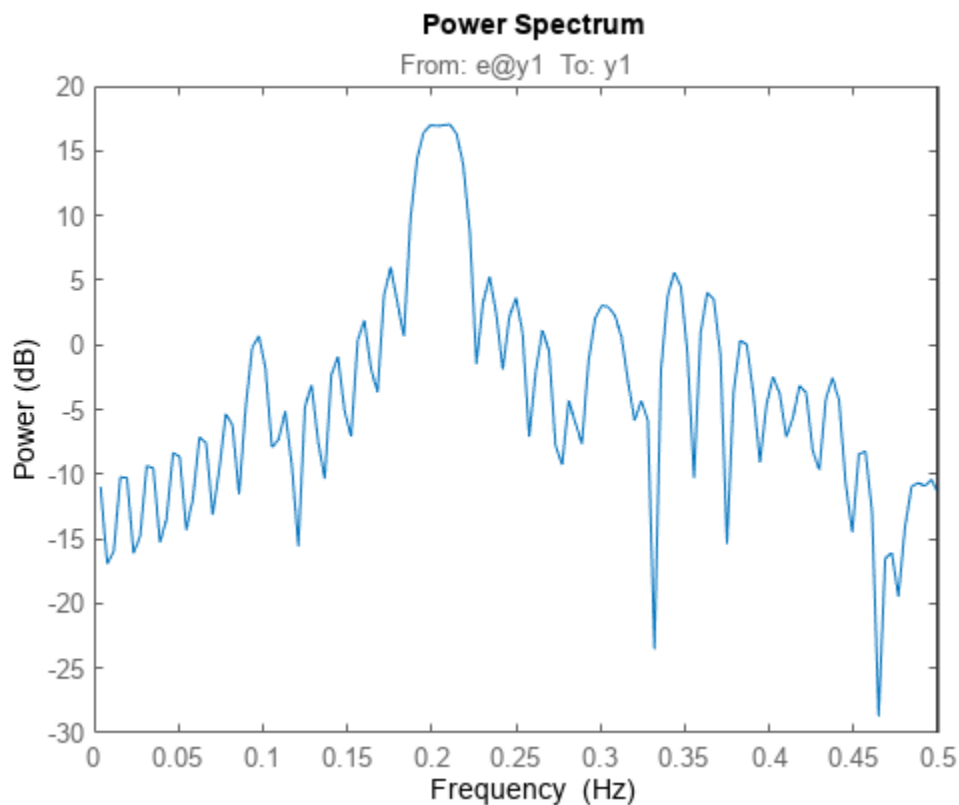
As a preliminary analysis step, let us check the periodogram of the data:

```
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);
```
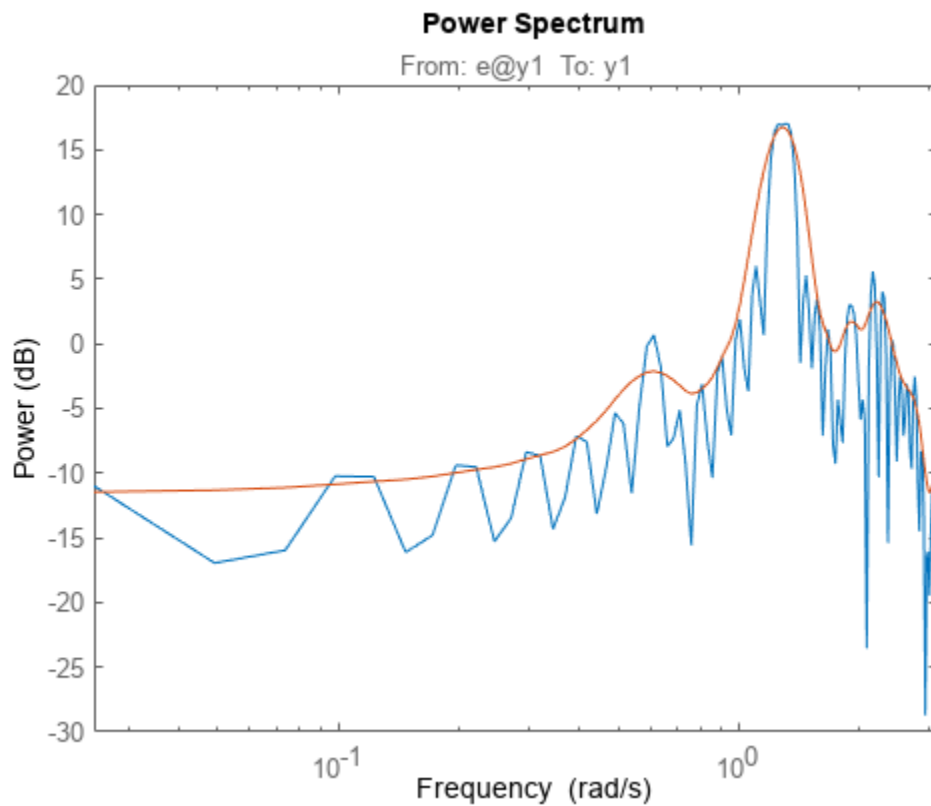
Let us now try the Blackman-Tukey approach to spectrum estimation:

```
ssm = spa(marple); % Function spa performs spectral estimation
spectrumplot(sp,'b',ssm,'g',w,opt);
legend({'Smoothed periodogram','Blackman-Tukey estimate'});
```
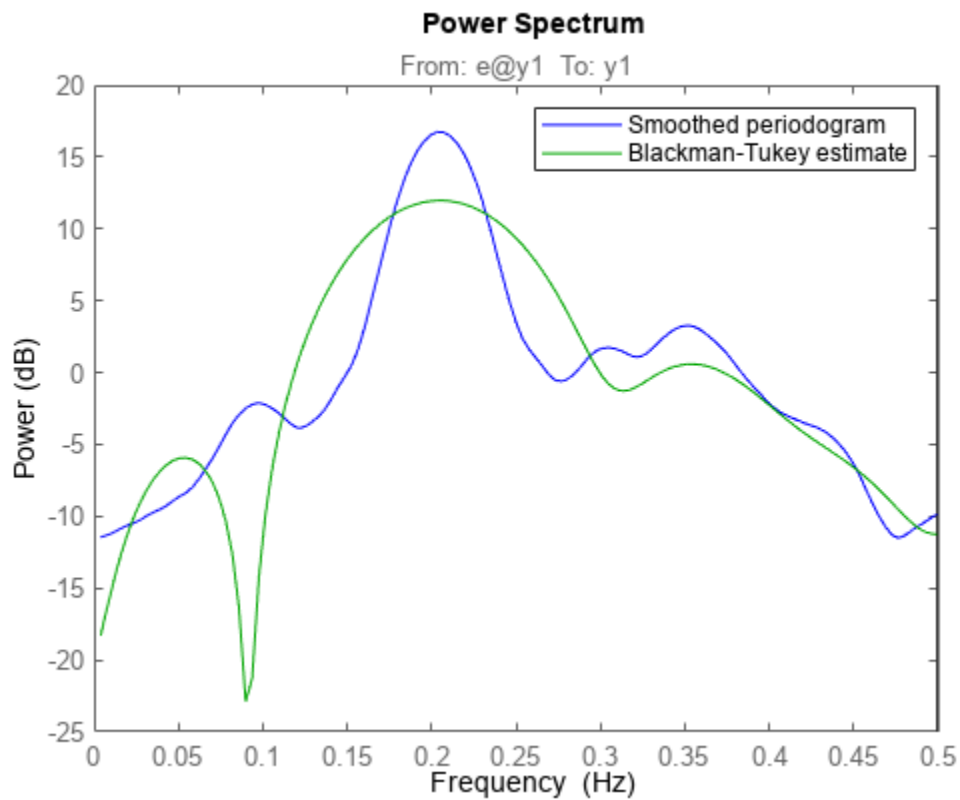
The default window length gives a very narrow lag window for this small amount of data. We can choose a larger lag window by:

```
ss20 = spa(marple,20);
spectrumplot(sp,'b',ss20,'g',w,opt);
legend({'Smoothed periodogram','Blackman-Tukey estimate'});
```
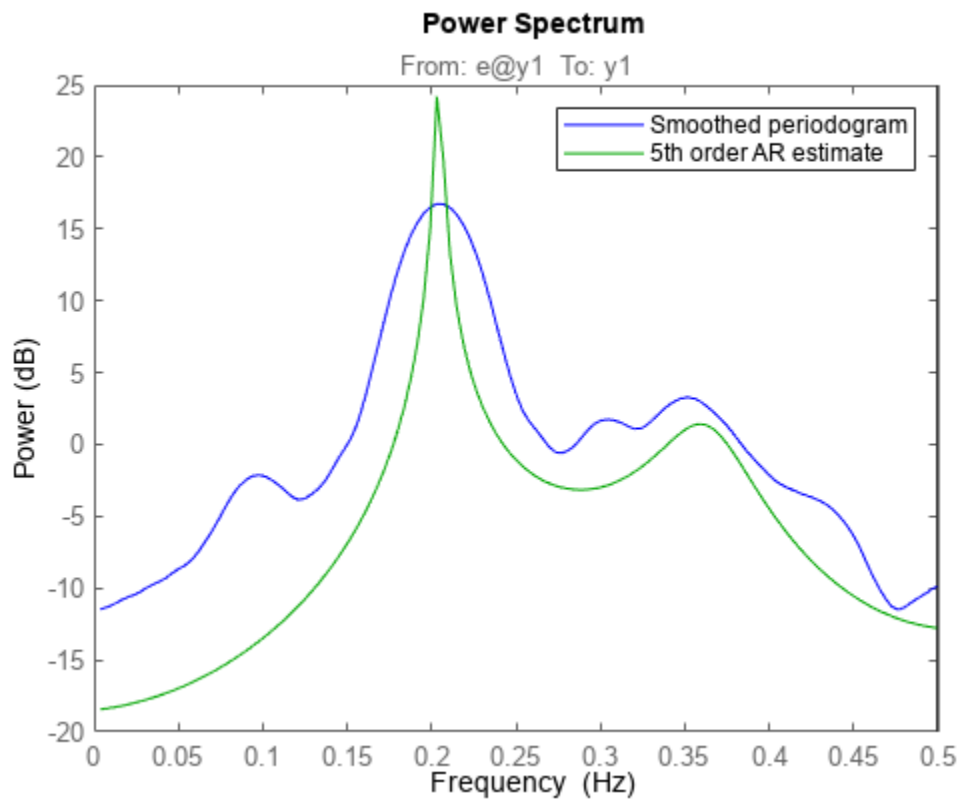
Power Spectrum
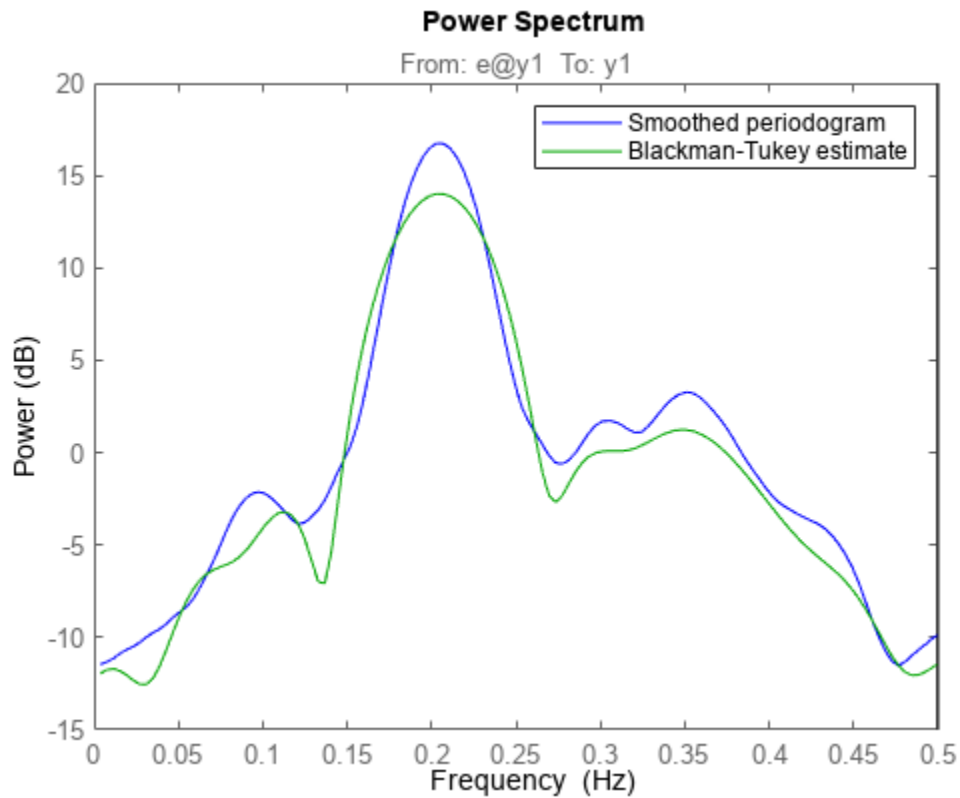
Estimating an Autoregressive (AR) Model

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'});
```

Power Spectrum

From: e@y1 To: y1

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'})
```
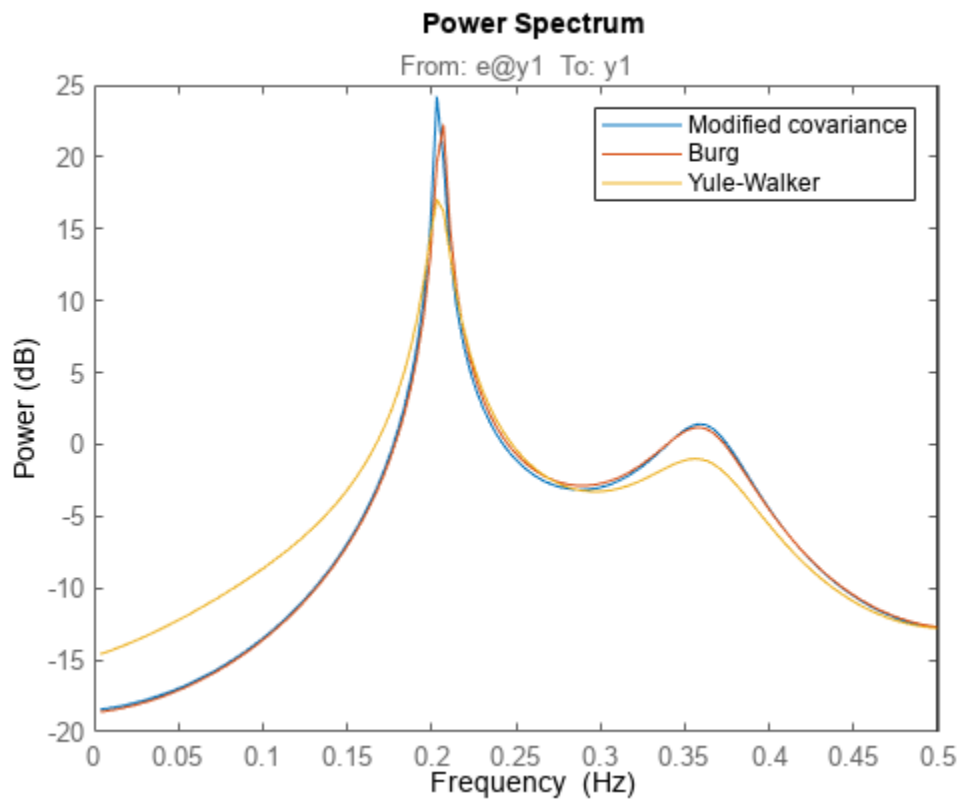
Estimating AR Model Using Instrumental Variable Approach

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'})
```

Power Spectrum

From: e@y1 To: y1
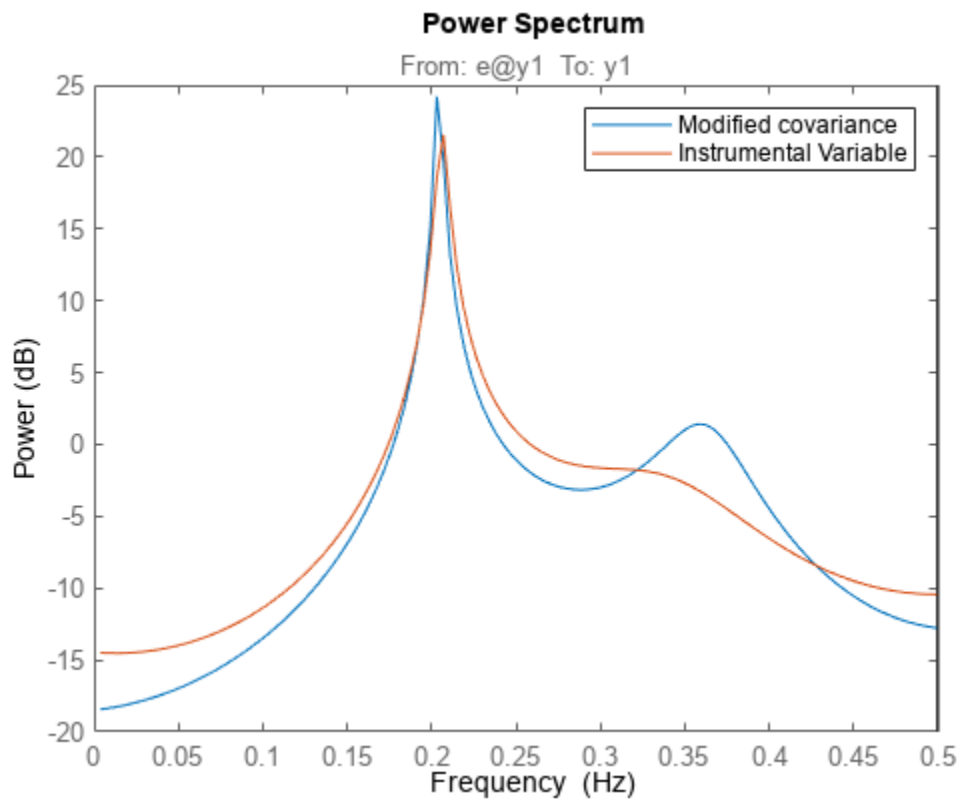
Autoregressive-Moving Average (ARMA) Model of the Spectra

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'})
```
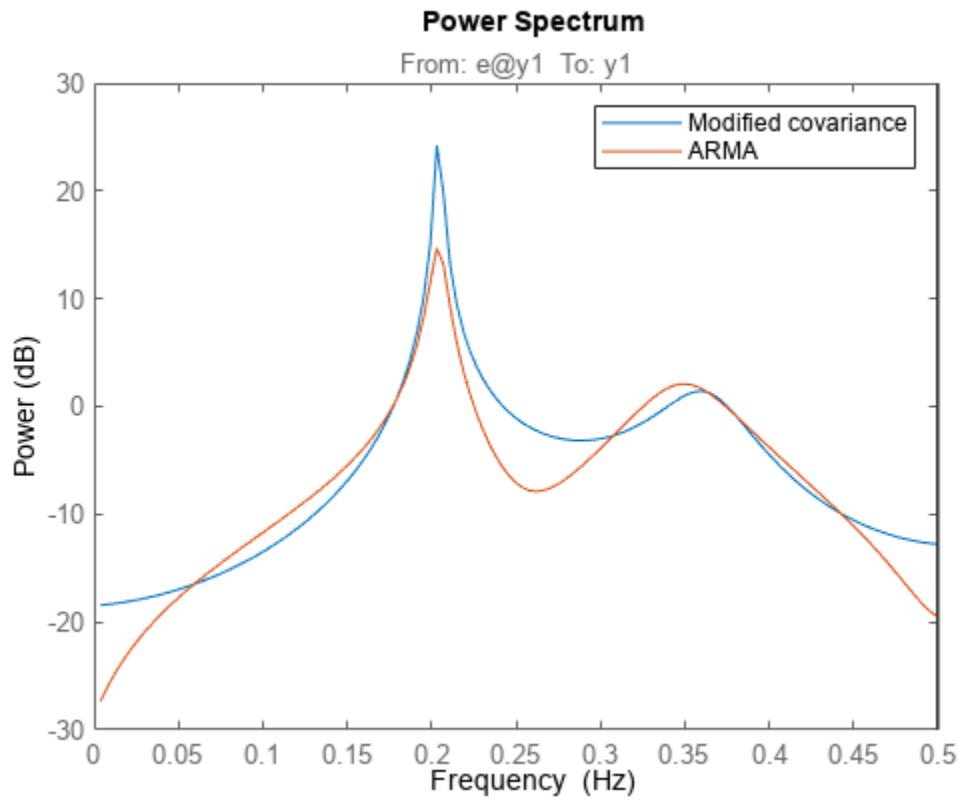
Analyze Time-Series Models

This example shows how to analyze time-series models.

A time-series model has no inputs. However, you can use many response computation commands on such models. The software treats (implicitly) the noise source `e(t)` as a measured input. Thus, `step(sys)` plots the step response assuming that the step input was applied to the noise channel `e(t)`.

To avoid ambiguity in how the software treats a time-series model, you can transform it explicitly into an input-output model using `noise2meas`. This command causes the noise input `e(t)` to be treated as a measured input and transforms the linear time series model with `Ny` outputs into an input-output model with `Ny` outputs and `Ny` inputs. You can use the resulting model with commands, such as, `bode`, `nyquist`, and `iopzmap` to study the characteristics of the `H` transfer function.
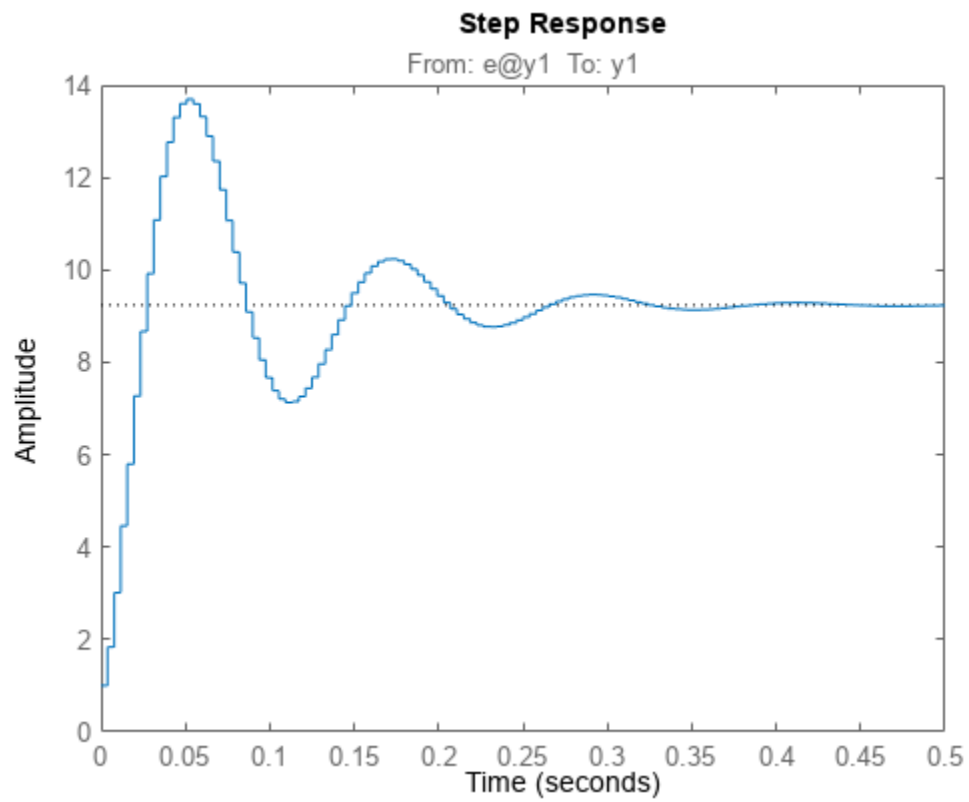
Estimate a time-series model.

```
load iddata9
sys = ar(z9,4);
```

Convert the time-series model to an input-output model.

```
iosys = noise2meas(sys);
```

Plot the step response of `H`.
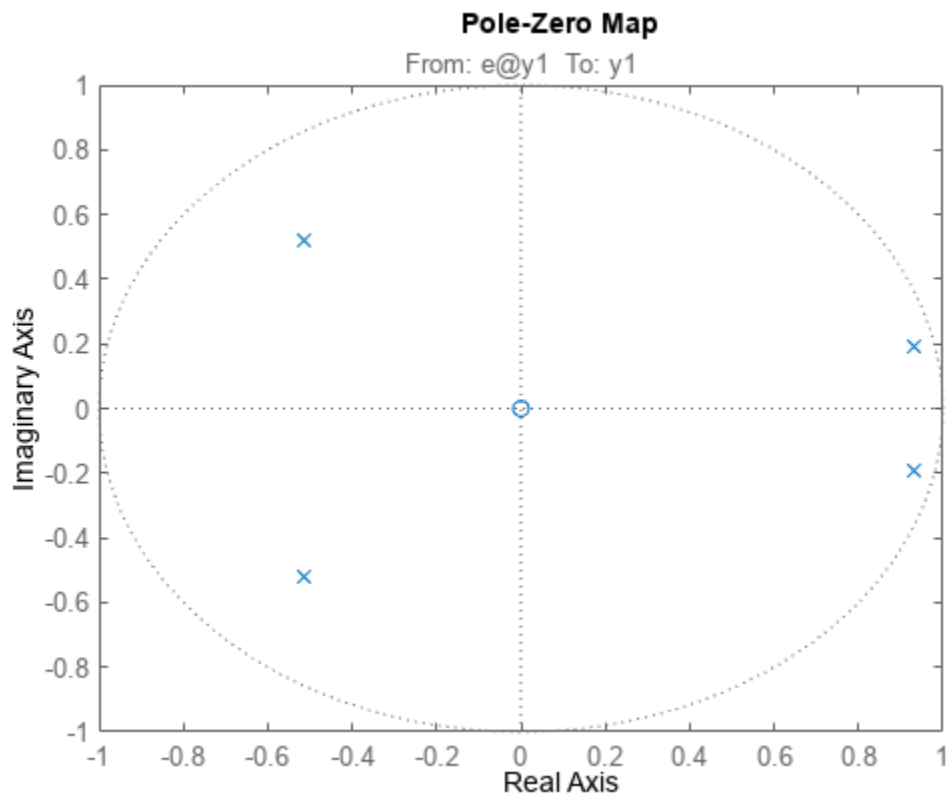
```
step(iosys);
```
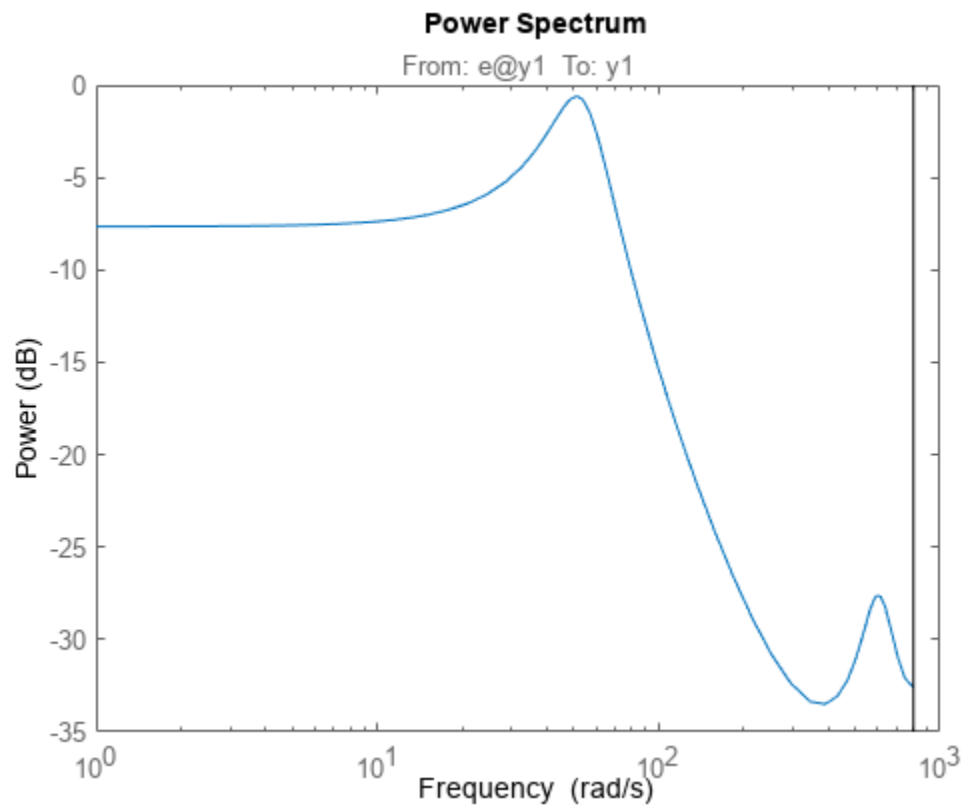
Plot the poles and zeros of H.

```
iopzmap(iosys);
```

Calculate and plot the time-series spectrum directly without converting to an input-output model.

```
spectrum(sys);
```

The command plots the time-series spectrum amplitude $\Phi(\omega) = \|H(\omega)\|^2$.

See Also
`noise2meas` | `step`

More About
- "What Are Time Series Models?" on page 14-2

Introduction to Forecasting of Dynamic System Response

Forecasting the response of a dynamic system is the prediction of future outputs of the system using past output measurements. In other words, given observations $y(t) = \{y(1), ..., y(N)\}$ of the output of a system, forecasting is the prediction of the outputs $y(N+1), ..., y(N+H)$ until a future time horizon H.

When you perform forecasting in System Identification Toolbox software, you first identify a model that fits past measured data from the system. The model can be a linear time series model such as AR, ARMA, and state-space models, or a nonlinear ARX model. If exogenous inputs influence the outputs of the system, you can perform forecasting using input-output models such as ARX and ARMAX. After identifying the model, you then use the `forecast` command to compute $y(N+1), ..., y(N+H)$. The command computes the forecasted values by:

- Generating a predictor model using the identified model.
- Computing the final state of the predictor using past measured data.
- Simulating the identified model until the desired forecasting horizon, H, using the final state as initial conditions.

This topic illustrates these forecasting steps for linear and nonlinear models. Forecasting the response of systems without external inputs (time series data) is illustrated, followed by forecasting for systems with an exogenous input. For information about how to perform forecasting in the toolbox, see "Forecast Output of Dynamic System" on page 14-44.

Forecasting Time Series Using Linear Models

The toolbox lets you forecast time series (output only) data using linear models such as AR, ARMA, and state-space models. Here is an illustration of forecasting the response of an autoregressive model, followed by the forecasting steps for more complex models such as moving-average and state-space models.

Autoregressive Models

Suppose that you have collected time series data $y(t) = \{y(1), ..., y(N)\}$ of a stationary random process. Assuming the data is a second-order autoregressive (AR) process, you can describe the dynamics by the following AR model:

$$y(t) + a_1 y(t-1) + a_2 y(t-2) = e(t)$$

Where a_1 and a_2 are the fit coefficients and $e(t)$ is the noise term.

You can identify the model using the `ar` command. The software computes the fit coefficients and variance of $e(t)$ by minimizing the 1-step prediction errors between the observations $\{y(1), ..., y(N)\}$ and model response.

Assuming that the innovations $e(t)$ are a zero mean white sequence, you can compute the predicted output $\widehat{y}(t)$ using the formula:

$$\widehat{y}(t) = -a_1 y(t-1) - a_2 y(t-2)$$

Where $y(t-1)$ and $y(t-2)$ are either measured data, if available, or previously predicted values. For example, the forecasted outputs five steps in the future are:

$$\widehat{y}(N + 1) = -a_1 y(N) - a_2 y(N - 1)$$

$$\widehat{y}(N + 2) = -a_1 \widehat{y}(N + 1) - a_2 y(N)$$

$$\widehat{y}(N + 3) = -a_1 \widehat{y}(N + 2) - a_2 \widehat{y}(N + 1)$$

$$\widehat{y}(N + 4) = -a_1 \widehat{y}(N + 3) - a_2 \widehat{y}(N + 2)$$

$$\widehat{y}(N + 5) = -a_1 \widehat{y}(N + 4) - a_2 \widehat{y}(N + 3)$$

Note that the computation of $\widehat{y}(N + 2)$ uses the previously predicted value $\widehat{y}(N + 1)$ because measured data is not available beyond time step N. Thus, the direct contribution of measured data diminishes as you forecast further into the future.

The forecasting formula is more complex for time series processes that contain moving-average terms.

Moving-Average Models

In moving-average (MA) models, the output depends on current and past innovations ($e(t)$,$e(t$-1), $e(t$-2), $e(t$-3)....). Thus, forecasting the response of MA models requires knowledge of the initial conditions of the measured data.

Suppose that time series data $y(t)$ from your system can be fit to a second-order moving-average model:

$$y(t) = e(t) + c_1 e(t - 1) + c_2 e(t - 2)$$

Suppose that y(1) and y(2) are the only available observations, and their values equal 5 and 10, respectively. You can estimate the model coefficients c_1 and c_2 using the armax command. Assume that the estimated c_1 and c_2 values are 0.1 and 0.2, respectively. Then assuming as before that $e(t)$ is a random variable with zero mean, you can predict the output value at time t using the following formula:

$$\widehat{y}(t) = c_1 e(t - 1) + c_2 e(t - 2)$$

Where e(t-1) and e(t-2) are the differences between the measured and the predicted response at times t-1 and t-2, respectively. If measured data does not exist for these times, a zero value is used because the innovations process $e(t)$ is assumed to be zero-mean white Gaussian noise.

Therefore, forecasted output at time $t = 3$ is:

$$\widehat{y}(3) = 0.1e(2) + 0.2e(1)$$

Where, the innovations e(1) and e(2) are the difference between the observed and forecasted values of output at time t equal to 1 and 2, respectively:

$$e(2) = y(2) - \widehat{y}(2) = y(2) - [0.1e(1) + 0.2\ e(0)]$$

$$e(1) = y(1) - \widehat{y}(1) = y(1) - [0.1e(0) + 0.2\ e(-1)]$$

Because the data was measured from time t equal to 1, the values of e(0) and e(-1) are unknown. Thus, to compute the forecasted outputs, the value of these initial conditions e(0) and e(-1) is required. You can either assume zero initial conditions, or estimate them.

- **Zero initial conditions**: If you specify that e(0) and e(-1) are equal to 0, the error values and forecasted outputs are:

$$e(1) = 5 - (0.1*0 + 0.2*0) = 5$$
$$e(2) = 10 - (0.1*5 + 0.2*0) = 9.5$$
$$\hat{y}(3) = 0.1*9.5 + 0.2*5 = 1.95$$

The forecasted values at times $t = 4$ and 5 are:

$$\hat{y}(4) = 0.1e(3) + 0.2e(2)$$
$$\hat{y}(5) = 0.1e(4) + 0.2e(3)$$

Here e(3) and e(4) are assumed to be zero as there are no measurements beyond time $t = 2$. This assumption yields, $\hat{y}(4) = 0.2*e(2) = 0.2*9.5 = 1.9$, and $\hat{y}(5) = 0$.

Thus, for this second-order MA model, the forecasted outputs that are more than two time steps beyond the last measured data point ($t = 2$) are all zero. In general, when zero initial conditions are assumed, the forecasted values beyond the order of a pure MA model with no autoregressive terms are all zero.

- **Estimated initial conditions**: You can estimate the initial conditions by minimizing the squared sum of 1-step prediction errors of all the measured data.

  For the MA model described previously, estimation of the initial conditions e(0) and e(-1) requires minimization of the following least-squares cost function:

  $$V = e(1)^2 + e(2)^2 = (y(1) - [0.1\ e(0) + 0.2\ e(-1)])^2 + (y(2) - [0.1\ e(1) + 0.2\ e(0)])^2$$

  Substituting a = e(0) and b = e(-1), the cost function is:

  $$V(a, b) = (5 - [0.1a + 0.2b])^2 + (10 - [0.1(5 - [0.1a + 0.2b]) + 0.2a])^2$$

  Minimizing V yields e(0) = 50 and e(-1) = 0, which gives:

  $$e(1) = 5 - (0.1*50 + 0.2*0) = 0$$
  $$e(2) = 10 - (0.1*0 + 0.2*50) = 0$$
  $$\hat{y}(3) = 0$$
  $$(4) = 0$$

  Thus, for this system, if the prediction errors are minimized over the available two samples, all future predictions are equal to zero, which is the mean value of the process. If there were more than two observations available, you would estimate e(-1) and e(0) using a least-squares approach to minimize the 1-step prediction errors over all the available data.

This example shows how to reproduce these forecasted results using the `forecast` command.

Load the measured data.

```
PastData = [5;10];
```

Create an MA model with A and C polynomial coefficients equal to 1 and [1 0.1 0.2], respectively.

```
model = idpoly(1,[],[1 0.1 0.2]);
```

Specify zero initial conditions, and forecast the output five steps into the future.

```
opt = forecastOptions('InitialCondition','z');
yf_zeroIC = forecast(model,PastData,5,opt)

yf_zeroIC = 5×1

    1.9500
    1.9000
         0
         0
         0
```

Specify that the software estimate initial conditions, and forecast the output.

```
opt = forecastOptions('InitialCondition','e');
yf_estimatedIC = forecast(model,PastData,5,opt)

yf_estimatedIC = 5×1
10^{-15} ×

   -0.3553
   -0.3553
         0
         0
         0
```

For arbitrary structure models, such as models with autoregressive *and* moving-average terms, the forecasting procedure can be involved and is therefore best described in the state-space form.

State-Space Models

The discrete-time state-space model of time series data has the form:

$$x(t + 1) = Ax(t) + Ke(t)$$
$$y(t) = Cx(t) + e(t)$$

Where, $x(t)$ is the state vector, $y(t)$ are the outputs, $e(t)$ is the noise-term. A, C, and K are fixed-coefficient state-space matrices.

You can represent any arbitrary structure linear model in state-space form. For example, it can be shown that the ARMA model described previously is expressed in state-space form using `A = [0 0;1 0]`, `K = [0.5;0]` and `C = [0.2 0.4]`. You can estimate a state-space model from observed data using commands such as `ssest` and `n4sid`. You can also convert an existing polynomial model such as AR, MA, ARMA, ARX, and ARMAX into the state-space form using the `idss` command.

The advantage of state-space form is that any autoregressive or moving-average model with multiple time lag terms (`t-1,t-2,t-3,...`) only has a single time lag (`t-1`) in state variables when the model is converted to state-space form. As a result, the required initial conditions for forecasting translate into a single value for the initial state vector `X(0)`. The `forecast` command converts all linear model to state-space form and then performs forecasting.

To forecast the response of a state-space model:

1   Generate a 1-step ahead predictor model for the identified model. The predictor model has the form:

$$\widehat{x}(t+1) = (A - K*C)\,\widehat{x}(t) + Ky(t)$$
$$\widehat{y}(t) = C*\widehat{x}(t)$$

Where $y(t)$ is the measured output and $\widehat{y}(t)$ is the predicted value. The measured output is available until time step N and is used as an input in the predictor model. The initial state vector is $\widehat{x}(0) = x_0$.

2 Assign a value to the initial state vector x_0.

The initial states are either specified as zero, or estimated by minimizing the prediction error over the measured data time span.

Specify a zero initial condition if the system was in a state of rest before the observations were collected. You can also specify zero initial conditions if the predictor model is sufficiently stable because stability implies the effect of initial conditions diminishes rapidly as the observations are gathered. The predictor model is stable if the eigenvalues of `A-K*C` are inside the unit circle.

3 Compute $\widehat{x}(N + 1)$, the value of the states at the time instant `t = N+1`, the time instant following the last available data sample.

To do so, simulate the predictor model using the measured observations as inputs:

$$\widehat{x}(1) = (A - K*C)\,x_0 + Ky(0)$$
$$\widehat{x}(2) = (A - K*C)\widehat{x}(1) + Ky(1)$$
$$\vdots$$
$$\widehat{x}(N+1) = (A - K*C)\widehat{x}(N) + Ky(N)$$

4 Simulate the response of the identified model for H steps using $\widehat{x}(N + 1)$ as initial conditions, where H is the prediction horizon. This response is the forecasted response of the model.

Reproduce the Output of forecast Command

This example shows how to manually reproduce forecasting results that are obtained using the `forecast` command. You first use the `forecast` command to forecast time series data into the future. You then compare the forecasted results to a manual implementation of the forecasting algorithm.

Load time series data.

```
load iddata9 z9
```

`z9` is an `iddata` object that stores time series data (no inputs).

Specify data to use for model estimation.

```
observed_data = z9(1:128);
Ts = observed_data.Ts;
t = observed_data.SamplingInstants;
y = observed_data.y;
```

`Ts` is the sample time of the measured data, `t` is the time vector, and `y` is the vector of measured data.

Estimate a discrete-time state space model of 4th order.

```
sys = ssest(observed_data,4,'Ts',Ts);
```

Forecast the output of the state-space model 100 steps into the future using the `forecast` command.

```
H = 100;
yh1 = forecast(sys,observed_data,H);
```

`yh1` is the forecasted output obtained using the `forecast` command. Now reproduce the output by manually implementing the algorithm used by the `forecast` command.

Retrieve the estimated state-space matrices to create the predictor model.

```
A = sys.A;
K = sys.K;
C = sys.C;
```

Generate a 1-step ahead predictor where the `A` matrix of the `Predictor` model is `A-K*C` and the `B` matrix is `K`.

```
Predictor = idss((A-K*C),K,C,0,'Ts',Ts);
```

Estimate initial states that minimize the difference between the observed output `y` and the 1-step predicted response of the identified model `sys`.

```
x0 = findstates(sys,observed_data,1);
```

Propagate the state vector to the end of observed data. To do so, simulate the predictor using `y` as input and `x0` as initial states.

```
Input = iddata([],y,Ts);
opt = simOptions('InitialCondition',x0);
[~,~,x] = sim(Predictor,Input,opt);
xfinal = x(end,:)';
```

`xfinal` is the state vector value at time `t(end)`, the last time instant when observed data is available. Forecasting 100 time steps into the future starts at the next time step, `t1 = t(end)+Ts`.

To implement the forecasting algorithm, the state vector value at time `t1` is required. Compute the state vector by applying the state update equation of the `Predictor` model to `xfinal`.

```
x0_for_forecasting = Predictor.A*xfinal + Predictor.B*y(end);
```

Simulate the identified model for `H` steps using `x0_for_forecasting` as initial conditions.

```
opt = simOptions('InitialCondition',x0_for_forecasting);
```

Because `sys` is a time series model, specify inputs for simulation as an H-by-0 signal, where `H` is the wanted number of simulation output samples.

```
Input = iddata([],zeros(H,0),Ts,'Tstart',t(end)+Ts);
yh2 = sim(sys,Input,opt);
```

Compare the results of the `forecast` command `yh1` with the manually computed results `yh2`.

```
plot(yh1,yh2,'r.')
```

Input-Output Data
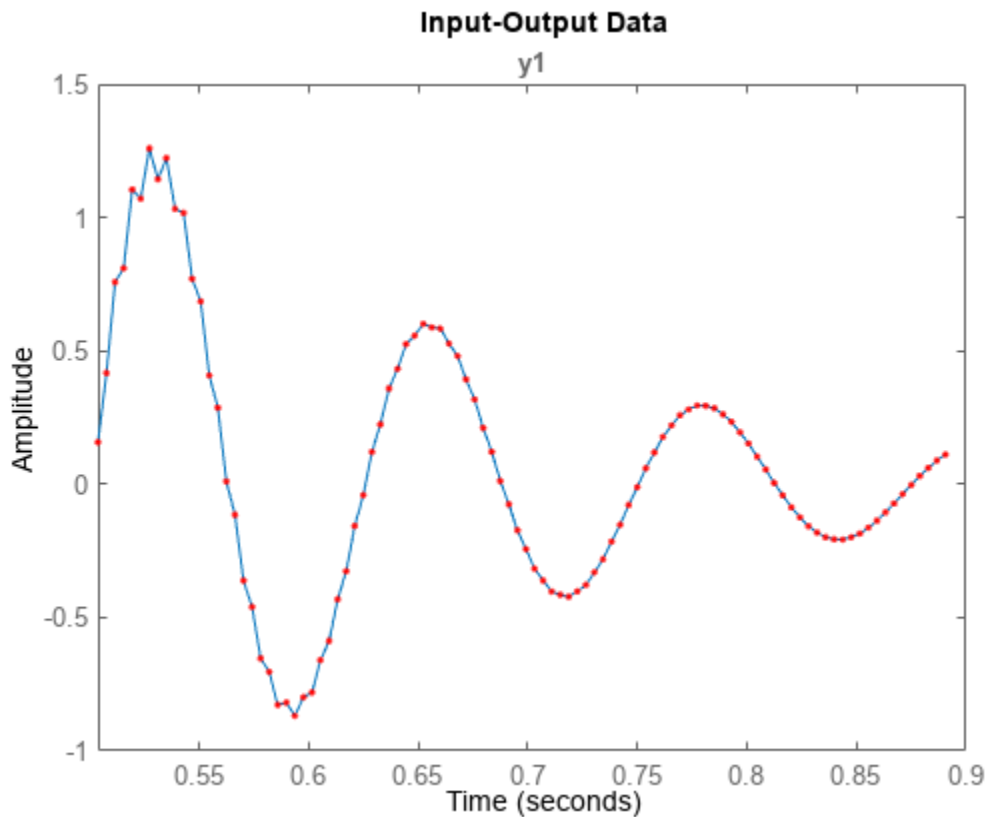
The plot shows that the results match.

Forecasting Response of Linear Models with Exogenous Inputs

When there are exogenous stimuli affecting the system, the system cannot be considered stationary. However, if these stimuli are measurable then you can treat them as inputs to the system and account for their effects when forecasting the output of the system. The workflow for forecasting data with exogenous inputs is similar to that for forecasting time series data. You first identify a model to fit the measured input-output data. You then specify the anticipated input values for the forecasting time span, and forecast the output of the identified model using the `forecast` command. If you do not specify the anticipated input values, they are assumed to be zero.

This example shows how to forecast an ARMAX model with exogenous inputs in the toolbox:

Load input-output data.

```
load iddata1 z1
```

`z1` is an `iddata` object with input-output data at 300 time points.

Use the first half of the data as past data for model identification.

```
past_data = z1(1:150);
```

Identify an ARMAX model Ay(t) = Bu(t-1) + Ce(t), of order [2 2 2 1].

```
na = 2; % A polynomial order
nb = 2; % B polynomial order
nc = 2; % C polynomial order
nk = 1; % input delay
sys = armax(past_data,[na nb nc nk]);
```
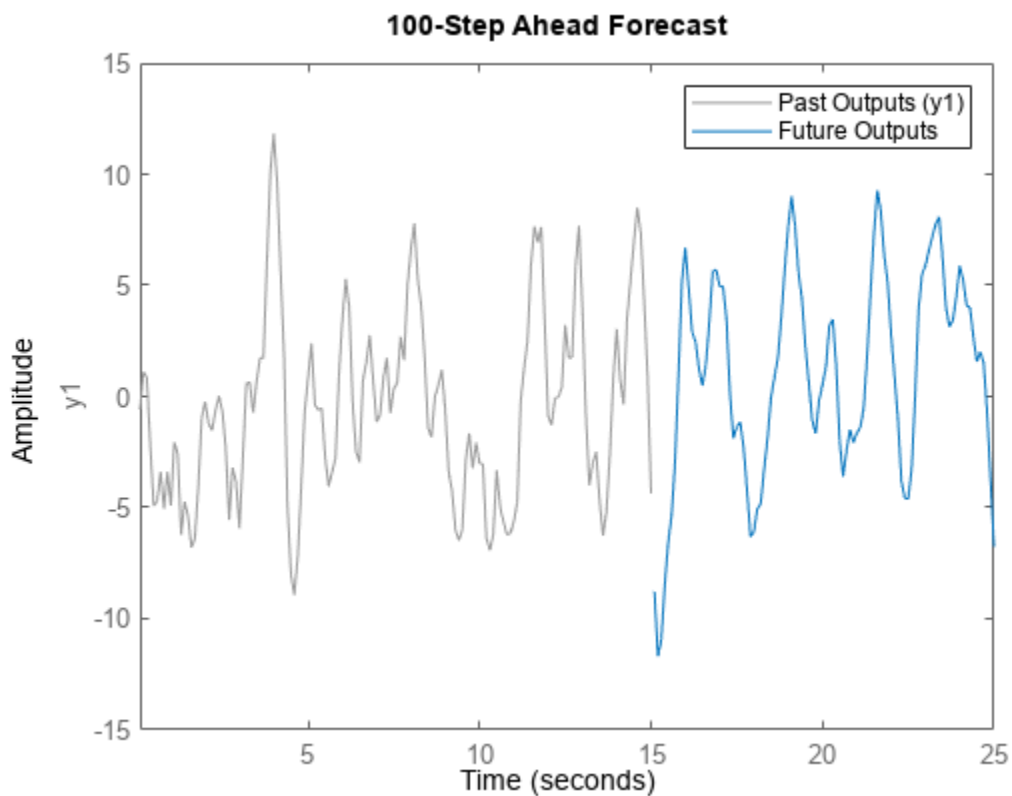
Forecast the response 100 time steps into the future, beyond the last sample of observed data past_data. Specify the anticipated inputs at the 100 future time points.

```
H = 100;
FutureInputs = z1.u(151:250);
forecast(sys,past_data,H,FutureInputs)
legend('Past Outputs','Future Outputs')
```


Forecasting Response of Nonlinear Models

The toolbox also lets you forecast data using nonlinear ARX, Hammerstein-Wiener, and nonlinear grey-box models.

Hammerstein-Wiener, and nonlinear grey-box models have a trivial noise-component, that is disturbance in the model is described by white noise. As a result, forecasting using the forecast command is the same as performing a pure simulation.

Forecasting Response of Nonlinear ARX Models

A time series nonlinear ARX model has the following structure:

$$y(t) = f(y(t-1), \ y(t-2), \ \dots, \ y(t-N)) + e(t)$$

Where f is a nonlinear function with inputs $R(t)$, the model regressors. The regressors can be the time-lagged variables $y(t-1)$, $y(t-2)$,..., $y(t-N)$ and their nonlinear expressions, such as $y(t-1)^2$, $y(t-1)y(t-2)$, $abs(y(t-1))$. When you estimate a nonlinear ARX model from the measured data, you specify the model regressors. You can also specify the structure of f using different structures such as wavelet networks and tree partitions. For more information, see the reference page for the `nlarx` estimation command.

Suppose that time series data from your system can be fit to a second-order linear-in-regressor model with the following polynomial regressors:

$$R(t) = [y(t-1), y(t-2), y(t-1)^2, y(t-2)^2, y(t-1)y(t-2)]^T$$

Then `f(R)=W'*R+c`, where `W=[`w_1 w_2 w_3 w_4 w_5`]` is a weighting vector, and `c` is the output offset.

The nonlinear ARX model has the form:

$$y(t) = w_1 y(t-1) + w_2 y(t-2) + w_3 y(t-1)^2 + w_4 y(t-2)^2 + w_5 y(t-1)y(t-2) + c + e(t)$$

When you estimate the model using the `nlarx` command, the software estimates the model parameters `W` and `c`.

When you use the `forecast` command, the software computes the forecasted model outputs by simulating the model H time steps into the future, using the last N measured output samples as initial conditions. Where N is the largest lag in the regressors, and H is the forecast horizon you specify.

For the linear-in-regressor model, suppose that you have measured 100 samples of the output y, and you want to forecast four steps into the future ($H = 4$). The largest lag in the regressors of the model is $N = 2$. Therefore, the software takes the last two samples of the data `y(99)` and `y(100)` as initial conditions, and forecasts the outputs as:

$$\widehat{y}(101) = w_1 y(100) + w_2 y(99) + w_3 y(^{100} + w_4 y(^{99} + w_5 y(100)y(99)$$

$$\widehat{y}(102) = w_1 \widehat{y}(101) + w_2 y(100) + w_3 \widehat{y}(^{101} + w_4 y(^{100} + w_5 \widehat{y}(101)y(100)$$

$$\widehat{y}(103) = w_1 \widehat{y}(102) + w_2 \widehat{y}(101) + w_3 \widehat{y}(^{102} + w_4 \widehat{y}(^{101} + w_5 \widehat{y}(102)\widehat{y}(101)$$

$$\widehat{y}(104) = w_1 \widehat{y}(103) + w_2 \widehat{y}(102) + w_3 \widehat{y}(^{103} + w_4 \widehat{y}(^{102} + w_5 \widehat{y}(103)\widehat{y}(102)$$

If your system has exogenous inputs, the nonlinear ARX model also includes regressors that depend on the input variables. The forecasting process is similar to that for time series data. You first identify the model, `sys`, using input-output data, `past_data`. When you forecast the data, the software simulates the identified model H time steps into the future, using the last N measured output samples as initial conditions. You also specify the anticipated input values for the forecasting time span, `FutureInputs`. The syntax for forecasting the response of nonlinear models with exogenous inputs is the same as that for linear models, `forecast(sys,past_data,H,FutureInputs)`.

See Also
`forecast` | `predict` | `sim`

Related Examples

- "Forecast Output of Dynamic System" on page 14-44
- "Forecast Multivariate Time Series" on page 17-25
- "Time Series Prediction and Forecasting for Prognosis" on page 23-2

More About

- "Simulate and Predict Identified Model Output" on page 17-6

Forecast Output of Dynamic System

To forecast the output of a dynamic system, you first identify a model that fits past measured data from the system, and then forecast the future outputs of the identified model. Suppose that you have a set Y of N measurements of the output of a system ($Y = \{y_1, y_2, ...y_N\}$). To forecast the outputs into the future:

1   Identify a model of the system using time series estimation commands such as `ar`, `arx`, `armax`, and `ssest`.

The software estimates the models by minimizing the squared sum of one-step ahead prediction errors. You can identify linear models such as AR, ARMA, and state-space models. You can also estimate nonlinear ARX and nonlinear grey-box models.

2   Validate the identified model using `predict` command.

The `predict` command predicts the output of an identified model over the time span of measured data ($Y_p = y_{p1}, y_{p2}, ...y_{pN}$). Use `predict` to determine if the predicted results Y_p match the observed outputs Y for a desired prediction horizon. If the predictions are good over the time span of available data, use the model for forecasting.

3   Specify forecasting options such as how the software computes initial conditions of the measured data. To specify the options, use the `forecastOptions` option set.

4   Compute the output of the identified model until a future time horizon H, ($y_{N+1}, y_{N+2},..., y_{N+H}$) using the `forecast` command. Unlike the `predict` command, the `forecast` command performs prediction into the future, in a time range beyond the last instant of measured data.

The software computes the forecasted values by:

- Generating a predictor model using the identified model.
- Computing the final state of the predictor using measured (available) data.
- Simulating the identified model using the final state as initial conditions.

For more information, see "Introduction to Forecasting of Dynamic System Response" on page 14-34.

You can also forecast outputs for systems where measurable exogenous inputs $u(t)$ influence the output observations. In this case, you first identify an input-output model using measured $y(t)$ and $u(t)$, and then use the `forecast` command.

Forecast Time Series Data Using an ARMA Model

This example shows how to forecast time series data from a system using an ARMA model. Load the time series data that is to be forecasted.

```
load iddata9 z9
past_data = z9.OutputData(1:50);
```

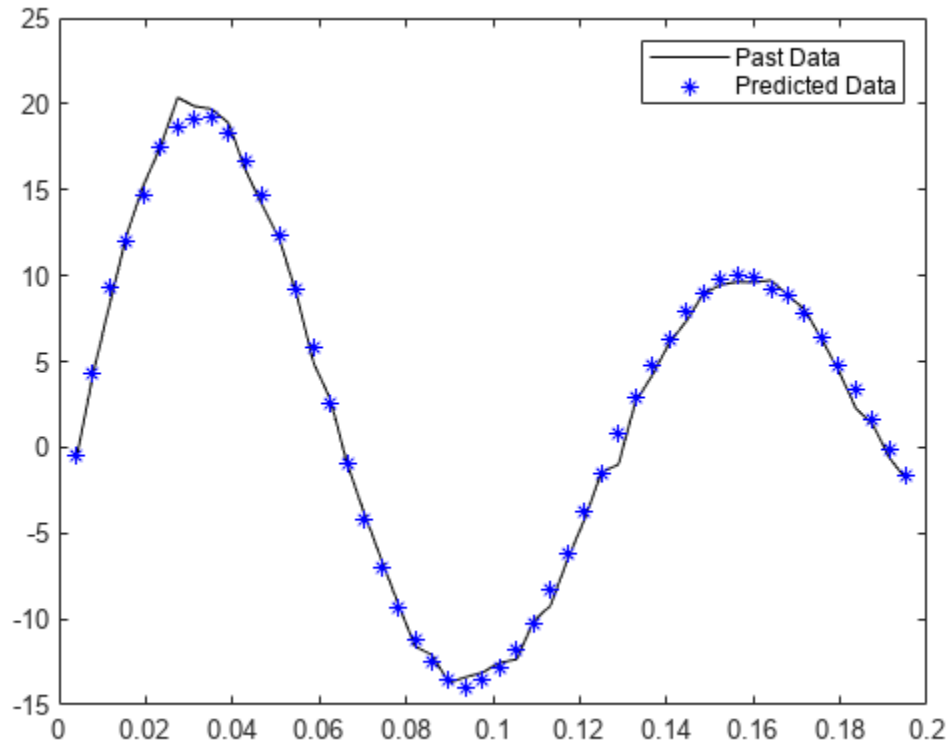Fit an ARMA model of order `[4 3]` to the measured data.

```
sys = armax(past_data,[4 3]);
```

Perform a 10-step ahead prediction to validate the model over the time-span of the measured data.

```
yp = predict(sys,past_data,10);
```

Plot the predicted response and the measured data.

```
t = z9.SamplingInstants;
t1 = t(1:50);
plot(t1,past_data,'k',t1,yp,'*b')
legend('Past Data','Predicted Data')
```


The plot shows that `sys` is a good prediction model that can be used for forecasting.
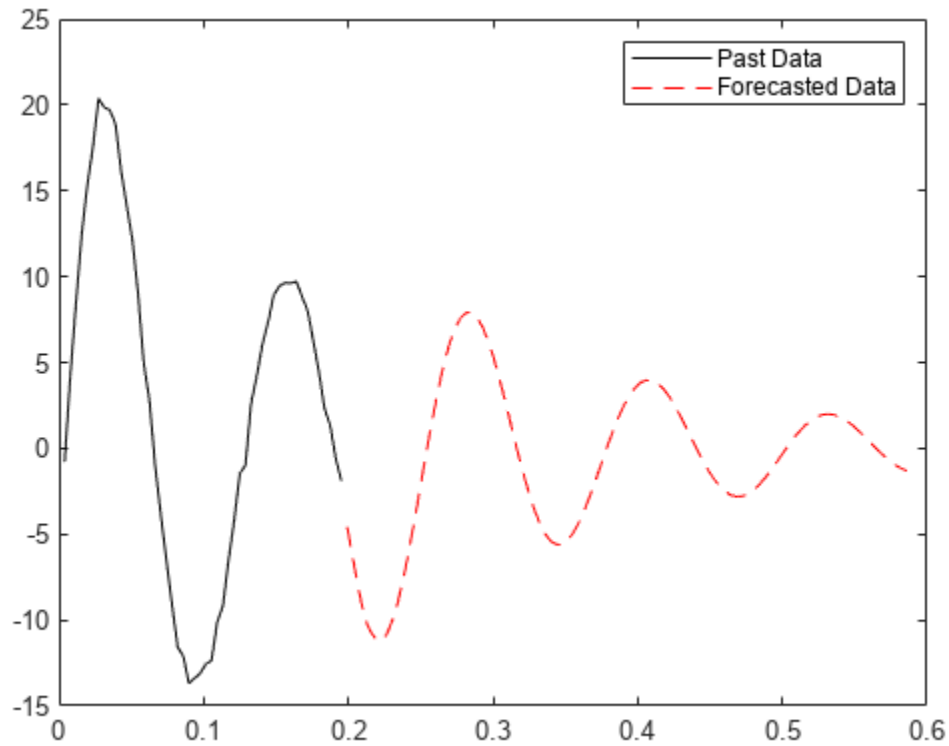
Specify zero initial conditions for the measured data.

```
opt = forecastOptions('InitialCOndition','z');
```

Forecast model output 100 steps beyond the estimation data.

```
H = 100;
yf = forecast(sys,past_data,H,opt);
```

Plot the past and forecasted data.

```
t2 = t(51:150)';
plot(t1,past_data,'k',t2,yf,'--r')
legend('Past Data','Forecasted Data')
```

See Also

More About

- "Forecast Multivariate Time Series" on page 17-25
- "Time Series Prediction and Forecasting for Prognosis" on page 23-2
- "Introduction to Forecasting of Dynamic System Response" on page 14-34

Modeling Current Signal from an Energizing Transformer

This example shows the modeling of a measured signal. We analyze the current signal from the R-phase when a 400 kV three-phase transformer is energized. The measurements were performed by Sydkraft AB in Sweden.

We describe the use of function `ar` for modeling the current signal. A non-parametric analysis of the signal is first performed. Tools for choosing a reasonable model order are then discussed, along with the use of `ar` for signal modeling. Methods for fitting a model to only a chosen range of harmonics are also discussed.

Introduction

Signals can be considered as the impulse response of an autoregressive linear model, and can thus be modeled using tools such as `ar`.

Data for signals can be encapsulated into `iddata` objects, by setting the output data of the object to the signal values, and leaving the input empty. For example, if `x(t)` represents a signal to be modeled, then the corresponding `iddata` object can be created as: `data = iddata(x,[],T);`, where `T` is the sample time of `x`.

Standard identification tools, such as `n4sid`, `ssest`, `ar` and `arx` may be used to estimate the characteristics of the "output-only" data. These models are assessed for their spectral estimation capability, as well as their ability to predict the future values of the signal from a measurement of their past values.

Analyzing Data

We begin this case study by loading the data for the current signal from the transformer:

```
load current.mat
```

Now, we package the current data (`i4r`) into an `iddata` object. The sample time is 0.001 seconds (1 ms).

```
i4r = iddata(i4r,[],0.001)  % Second argument empty for no input


i4r =

Time domain data set with 601 samples.
Sample time: 0.001 seconds

Outputs      Unit (if specified)
   y1

```
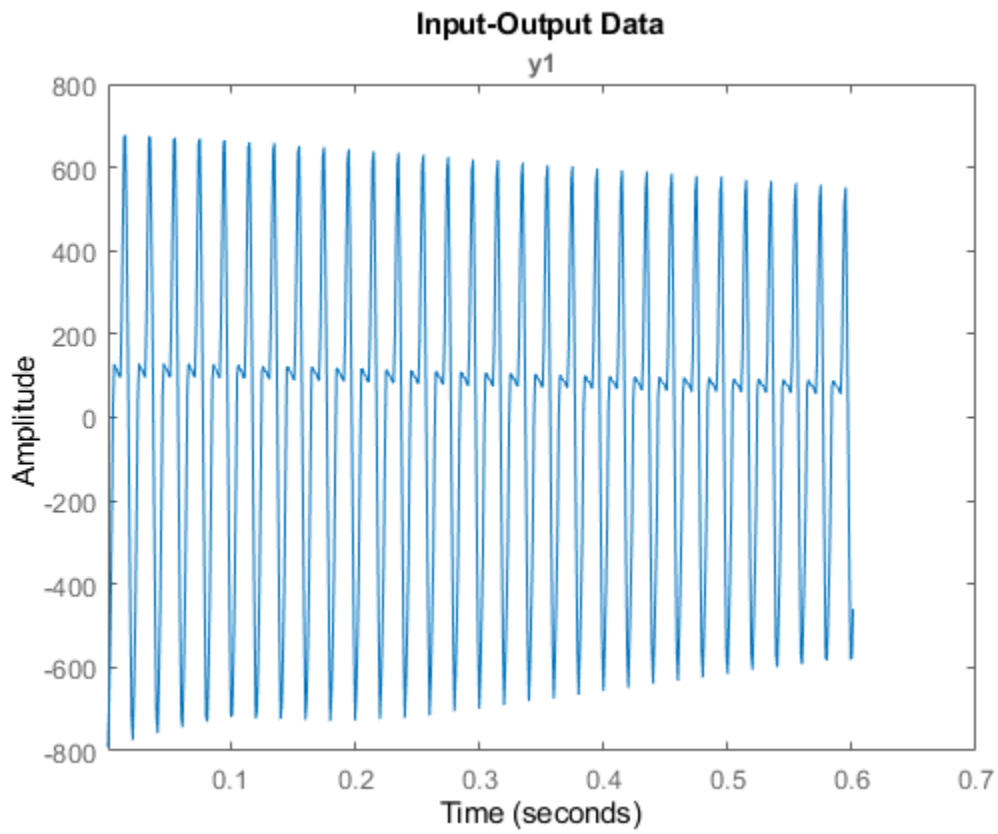
Let us now analyze this data. First, take a look at the data:

```
plot(i4r)
```
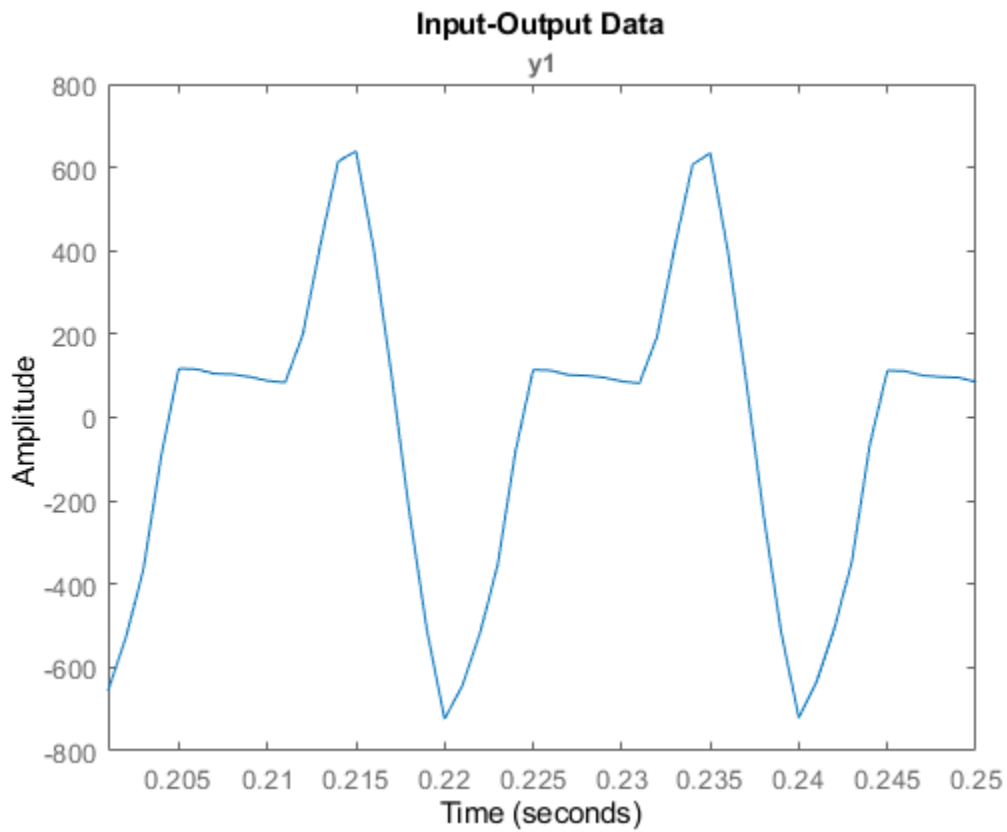
Input-Output Data

y1

A close up view of the data is shown below:

```
plot(i4r(201:250))
```

Input-Output Data

Next, we compute the raw periodogram of the signal:

```
ge = etfe(i4r)
spectrum(ge)
```
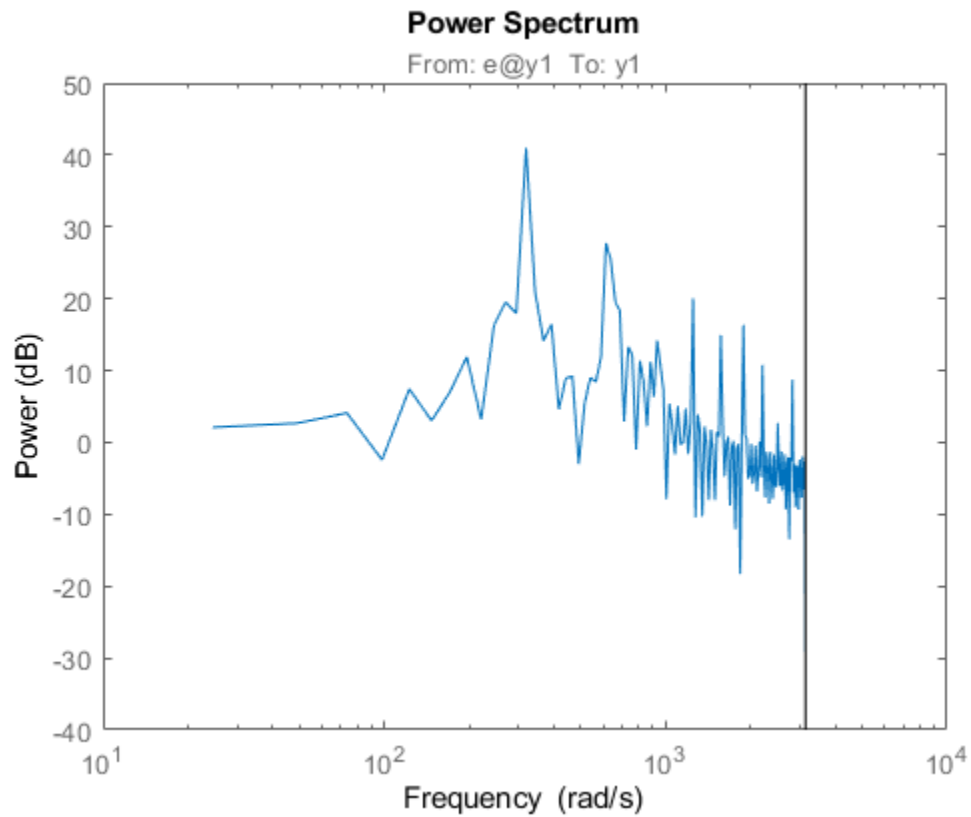
```
ge =

IDFRD model.
Contains spectrum of signal in the "SpectrumData" property.

Output channels: 'y1'
Status:
Estimated using ETFE on time domain data "i4r".
```
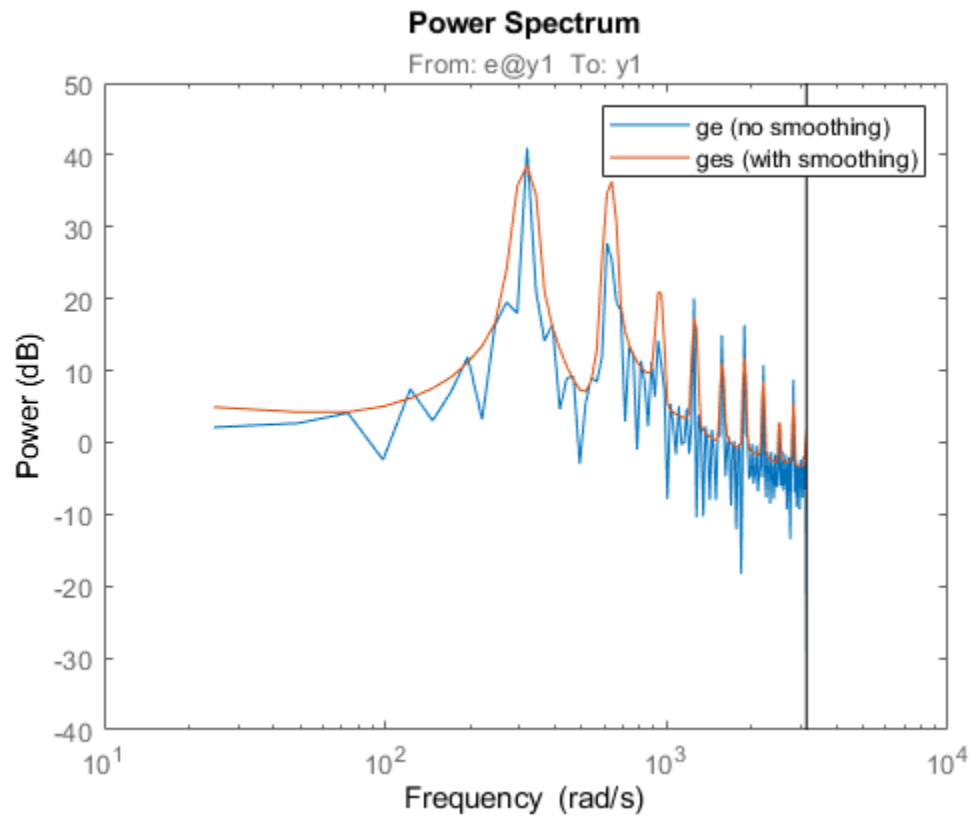
**Power Spectrum**

From: e@y1 To: y1

This periodogram reveals several harmonics, but is not very smooth. A smoothed periodogram is obtained by:

```
ges = etfe(i4r,size(i4r,1)/4);
spectrum(ge,ges);
legend({'ge (no smoothing)','ges (with smoothing)'})
```

Configure the plot to use linear frequency scale and Hz units:

```
h = spectrumplot(ges);
opt = getoptions(h);
opt.FreqScale = 'linear';
opt.FreqUnits = 'Hz';
setoptions(h,opt);
axis([0 500,-5 40])
grid on, legend('ges')
```
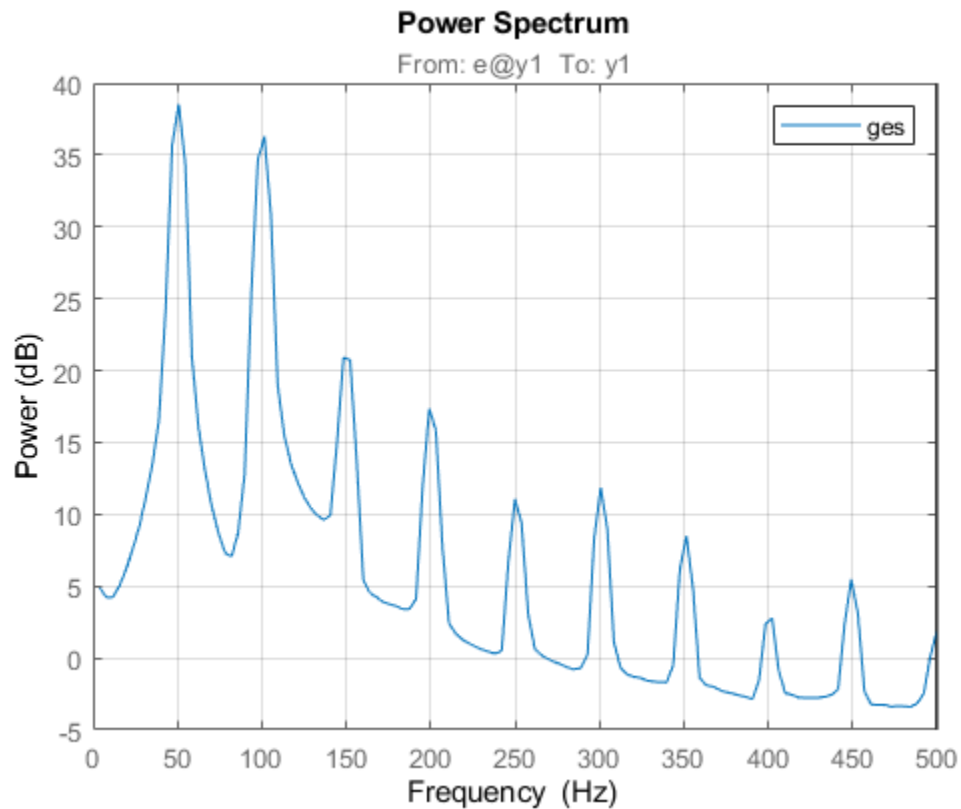
We clearly see the dominant frequency component of 50 Hz, and its harmonics.

Let us perform a spectral analysis of the data using `spa`, which uses a Hann window to compute the spectral amplitudes (as opposed to `etfe` which just computes the raw periodogram). The standard estimate (with the default window % size, which is not adjusted to resonant spectra) gives:

```
gs =  spa(i4r);
hold on
spectrumplot(gs);
legend({'ges (using etfe)','gs (using spa)'})
hold off
```
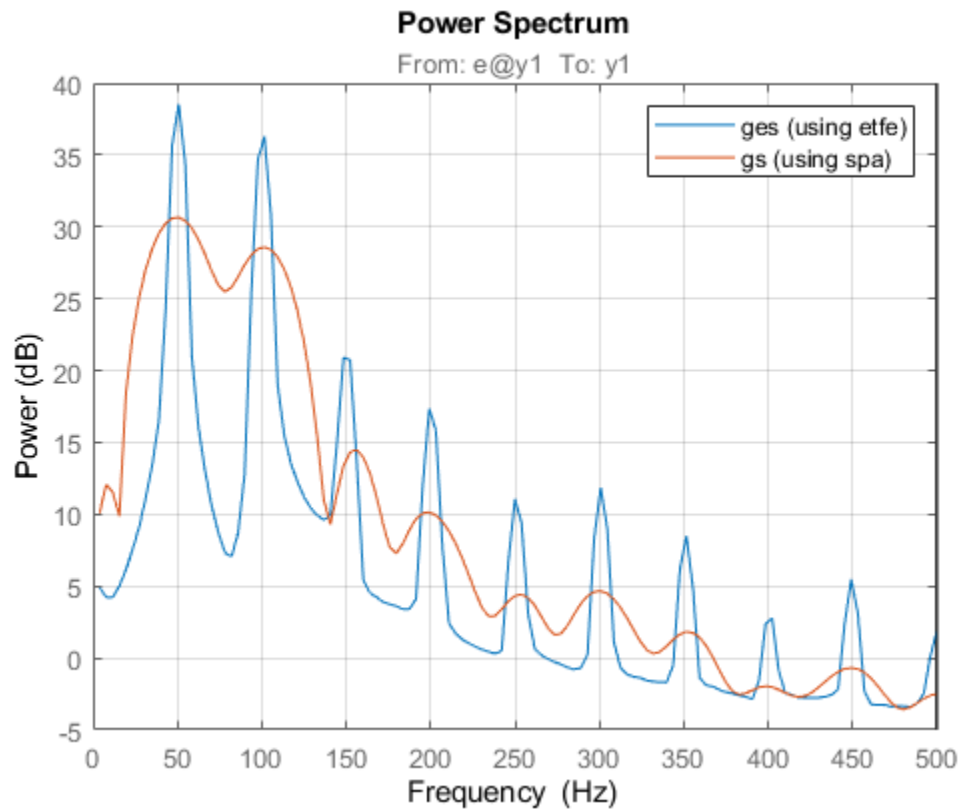
We see that a very large lag window will be required to see all the fine resonances of the signal. Standard spectral analysis does not work well. We need a more sophisticated model, such as those provided by parametric autoregressive modeling techniques.
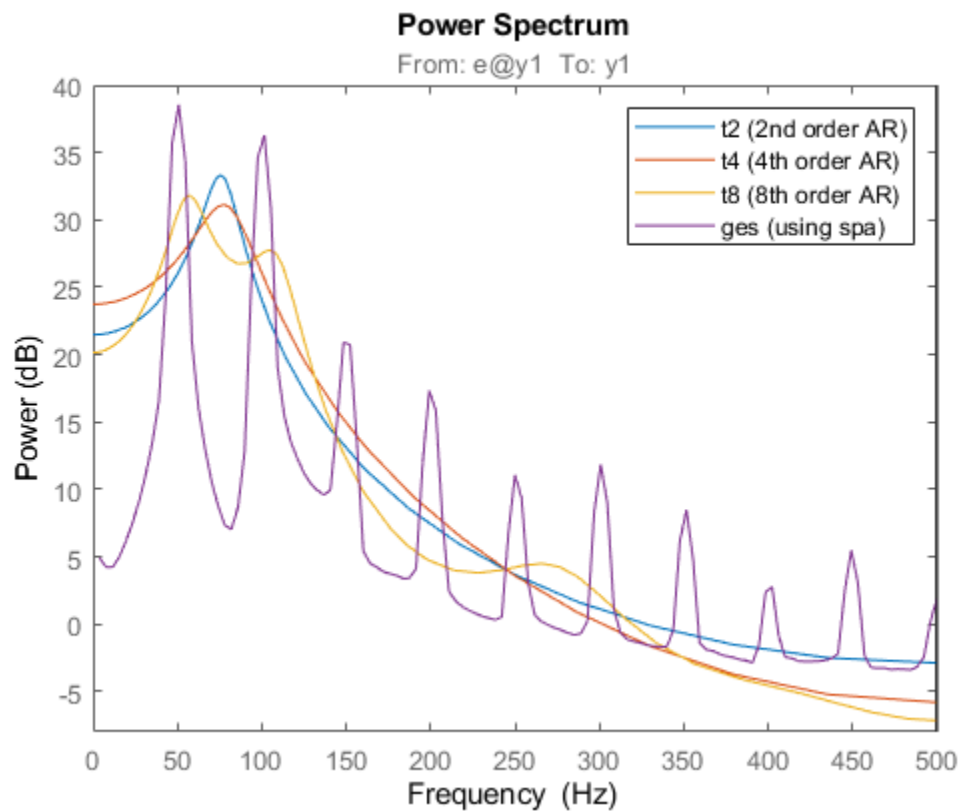
Parametric Modeling of the Current Signal

Let us now compute the spectra by parametric AR-methods. Models of 2nd 4th and 8th order are obtained by:

```
t2 = ar(i4r,2);
t4 = ar(i4r,4);
t8 = ar(i4r,8);
```

Let us take a look at their spectra:

```
spectrumplot(t2,t4,t8,ges,opt);
axis([0 500,-8 40])
legend({'t2 (2nd order AR)','t4 (4th order AR)','t8 (8th order AR)','ges (using spa)'});
```

Power Spectrum
From: e@y1 To: y1

We see that the parametric spectra are not capable of picking up the harmonics. The reason is that the AR-models attach too much attention to the higher frequencies, which are difficult to model. (See Ljung (1999) Example 8.5).

We will have to go to high order models before the harmonics are picked up.

What will a useful order be? We can use `arxstruc` to determine that.

```
V = arxstruc(i4r(1:301),i4r(302:601),(1:30)'); % Checking all order up to 30
```

Execute the following command to select the best order interactively: `nn = selstruc(V,'log');`
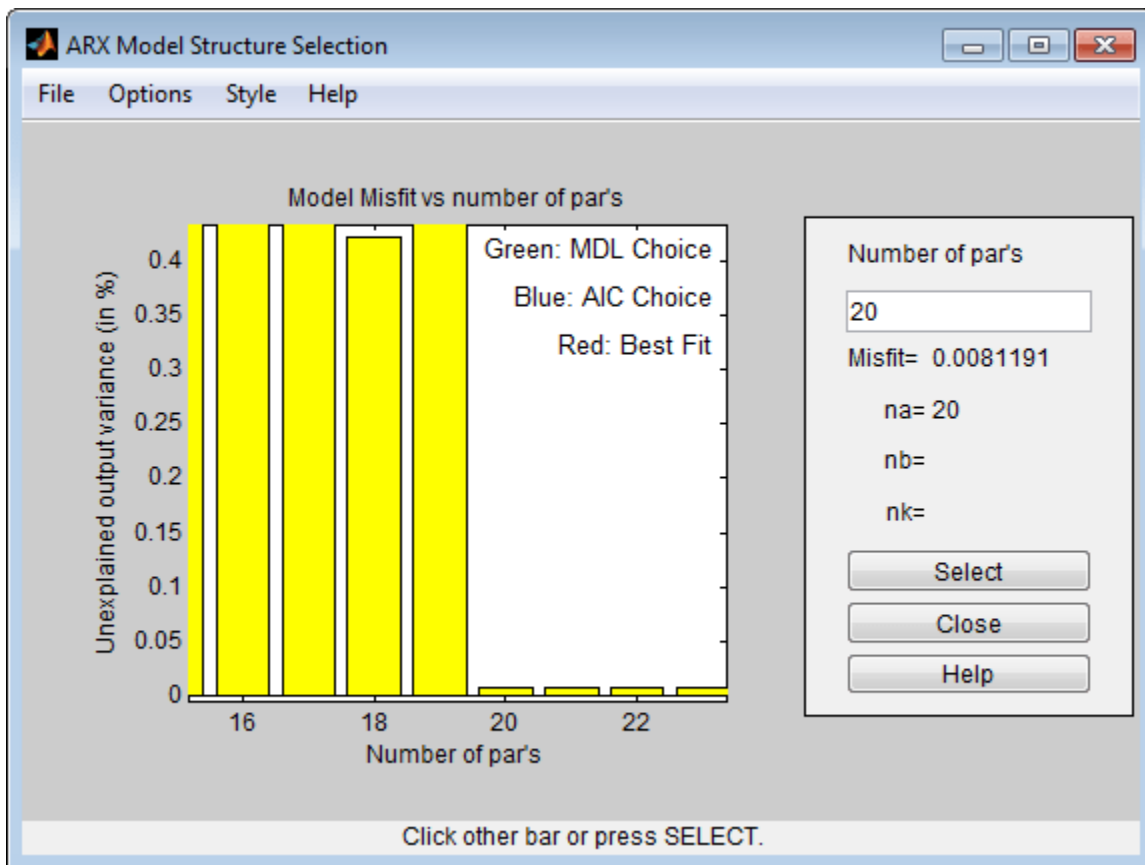
As the figure above shows, there is a dramatic drop for `n=20`. So let us pick that order for the following discussions.

```
t20 = ar(i4r,20);
spectrumplot(ges,t20,opt);
axis([0 500 -25 80])
legend({'ges (using spa)','t20 (20th order AR)'});
```

Power Spectrum
From: e@y1 To: y1
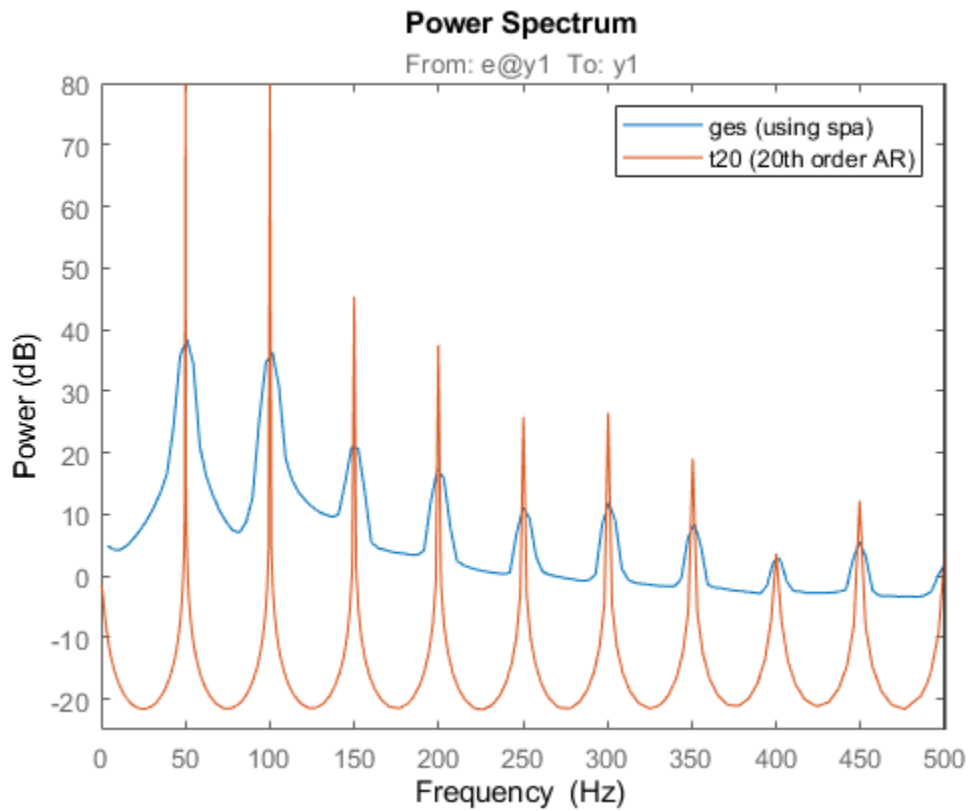
All the harmonics are now picked up, but why has the level dropped? The reason is that t20 contains very thin but high peaks. With the crude grid of frequency points in t20 we simply don't see the true levels of the peaks. We can illustrate this as follows:

```
g20c = idfrd(t20,(551:650)/600*150*2*pi); % A frequency region around 150 Hz
spectrumplot(ges,t20,g20c,opt)
axis([0 500 -25 80])
legend({'ges (using spa)','t20 (20th order AR)','g20c (resp. around 150 Hz)'});
```

Power Spectrum

From: e@y1 To: y1
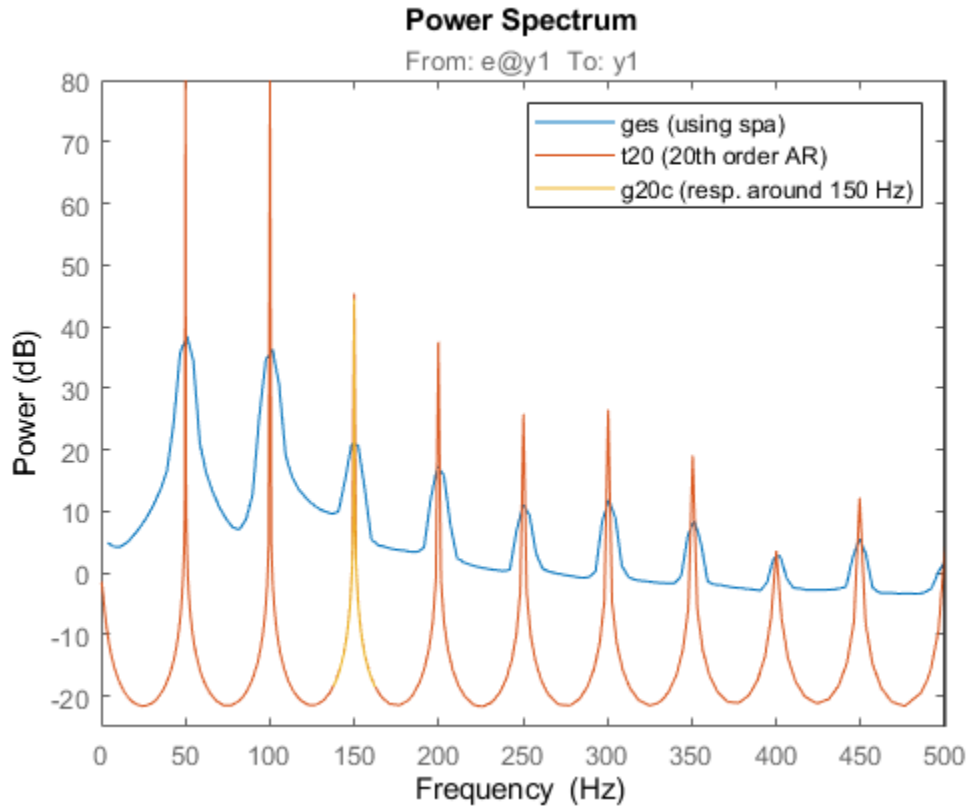
As this plot reveals, the model `t20` is fairly accurate; when plotted on a fine frequency grid, it does capture the harmonics of the signal quite accurately.

Modeling Only the Lower-Order Harmonics

If we are primarily interested in the lower harmonics, and want to use lower order models we will have to apply prefiltering of the data. We select a 5th order Butterworth filter with cut-off frequency at 155 Hz. (This should cover the 50, 100 and 150 Hz modes):

```
i4rf = idfilt(i4r,5,155/500); % 500 Hz is the Nyquist frequency
t8f = ar(i4rf,8);
```

Let us now compare the spectrum obtained from the filtered data (8th order model) with that for unfiltered data (8th order) and with the periodogram:

```
spectrumplot(t8f,t8,ges,opt)
axis([0 350 -60 80])
legend({'t8f (8th order AR, filtered data)',...
    't8 (8th order AR, unfiltered data)','ges (using spa)'});
```

Power Spectrum

From: e@y1 To: y1
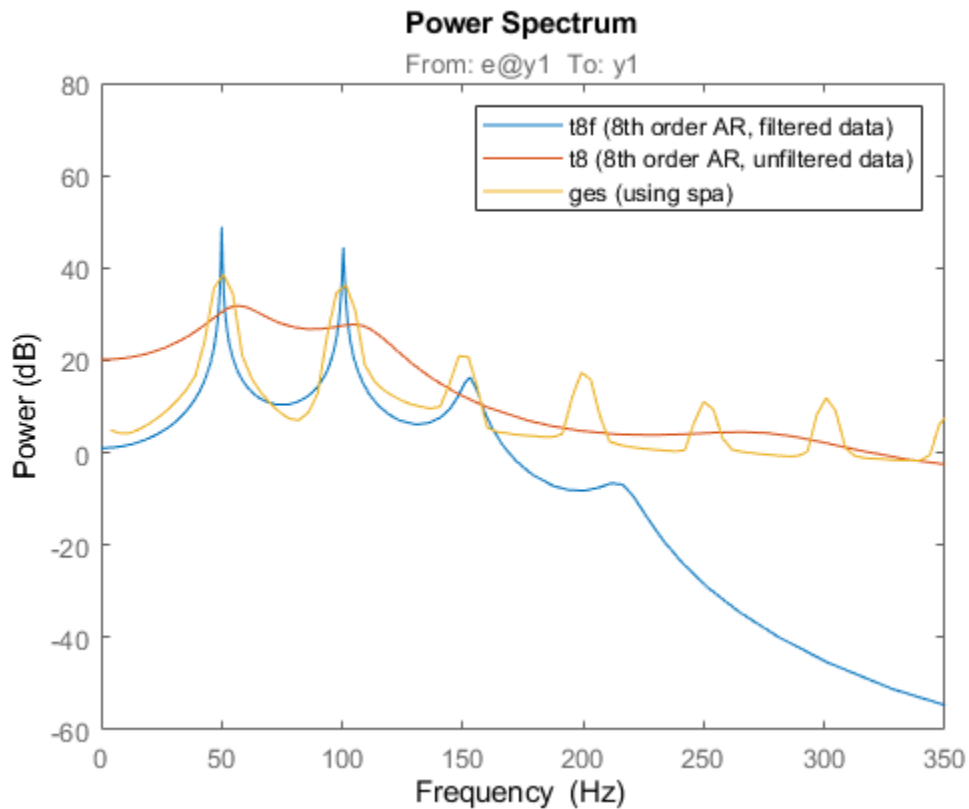
We see that with the filtered data we pick up the first three peaks in the spectrum quite well.

We can compute the numerical values of the resonances as follows: The roots of a sampled sinusoid of frequency, say `om`, are located on the unit circle at `exp(i*om*T)`, T being the sample time. We thus proceed as follows:

```
a = t8f.a % The AR-polynomial
omT = angle(roots(a))'
freqs = omT/0.001/2/pi';
% show only the positive frequencies for clarity:
freqs1 = freqs(freqs>0) % In Hz
```

```
a =

  Columns 1 through 7

    1.0000    -5.0312    12.7031    -20.6934    23.7632    -19.6987    11.5651

  Columns 8 through 9

   -4.4222     0.8619


omT =

  Columns 1 through 7
```

```
    1.3591    -1.3591    0.9620    -0.9620    0.3146    -0.3146    0.6314

  Column 8

   -0.6314


freqs1 =

  216.3063   153.1036    50.0665   100.4967
```
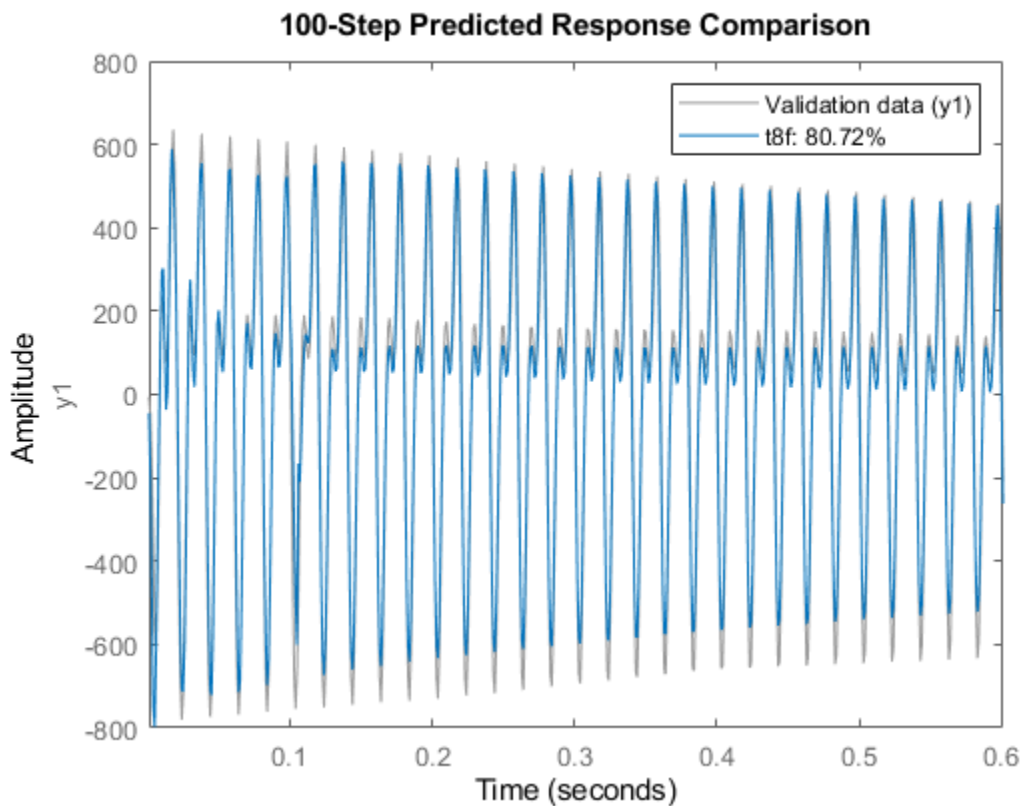
We thus find the first three harmonics (50, 100 and 150 Hz) quite well.

We could also test how well the model `t8f` is capable of predicting the signal, say 100 ms (100 steps) ahead, and evaluate the fit on samples 201 to 500:

```
compare(i4rf,t8f,100,compareOptions('Samples',201:500));
```


As observed, a model of the first 3 harmonics is pretty good at predicting the future output values, even 100 steps ahead.
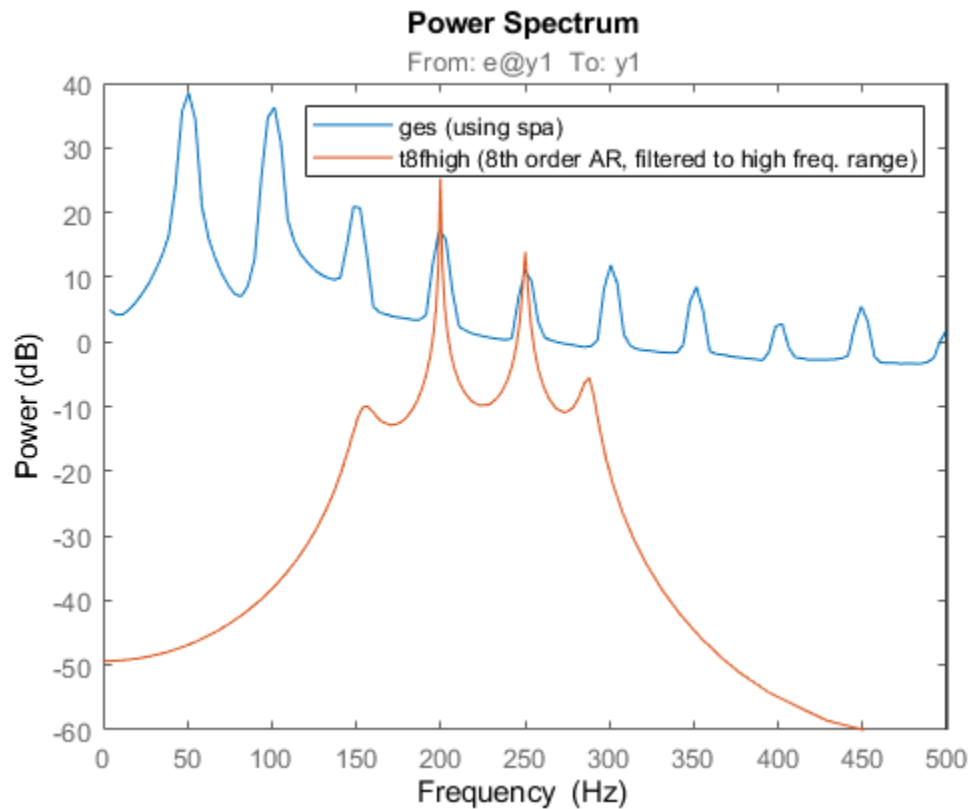
Modeling Only the Higher-Order Harmonics

If we were interested in only the fourth and fifth harmonics (around 200 and 250 Hz) we would proceed by band-filtering the data to this higher frequency range:

```
i4rff = idfilt(i4r,5,[185 275]/500);
t8fhigh = ar(i4rff,8);
spectrumplot(ges,t8fhigh,opt)
axis([0 500 -60 40])
legend({'ges (using spa)','t8fhigh (8th order AR, filtered to high freq. range)'});
```


We thus got a good model in `t8fhigh` for describing the 4th and 5th harmonics. We thus see that with proper prefiltering, low order parametric models can be built that give good descriptions of the signal over the desired frequency ranges.

Conclusions

Which model is the best? In general, a higher order model would give a higher fidelity. To analyze this, we consider what the 20th order model would give in terms of its capability in estimating harmonics:

```
a = t20.a  % The AR-polynomial
omT = angle(roots(a))'
freqs = omT/0.001/2/pi';
% show only the positive frequencies for clarity:
freqs1 = freqs(freqs>0) %In Hz


a =

  Columns 1 through 7
```

```
   1.0000    0.0034    0.0132    0.0012    0.0252    0.0059    0.0095

  Columns 8 through 14

   0.0038    0.0166    0.0026    0.0197   -0.0013    0.0143    0.0145

  Columns 15 through 21

   0.0021    0.0241   -0.0119    0.0150    0.0246   -0.0221   -0.9663


omT =

  Columns 1 through 7

        0    0.3146   -0.3146    0.6290   -0.6290    0.9425   -0.9425

  Columns 8 through 14

   1.2559   -1.2559    1.5726   -1.5726    1.8879   -1.8879    2.2027

  Columns 15 through 20

  -2.2027    2.5136   -2.5136    3.1416    2.8240   -2.8240


freqs1 =

  Columns 1 through 7

   50.0639  100.1139  149.9964  199.8891  250.2858  300.4738  350.5739

  Columns 8 through 10

  400.0586  500.0000  449.4611
```
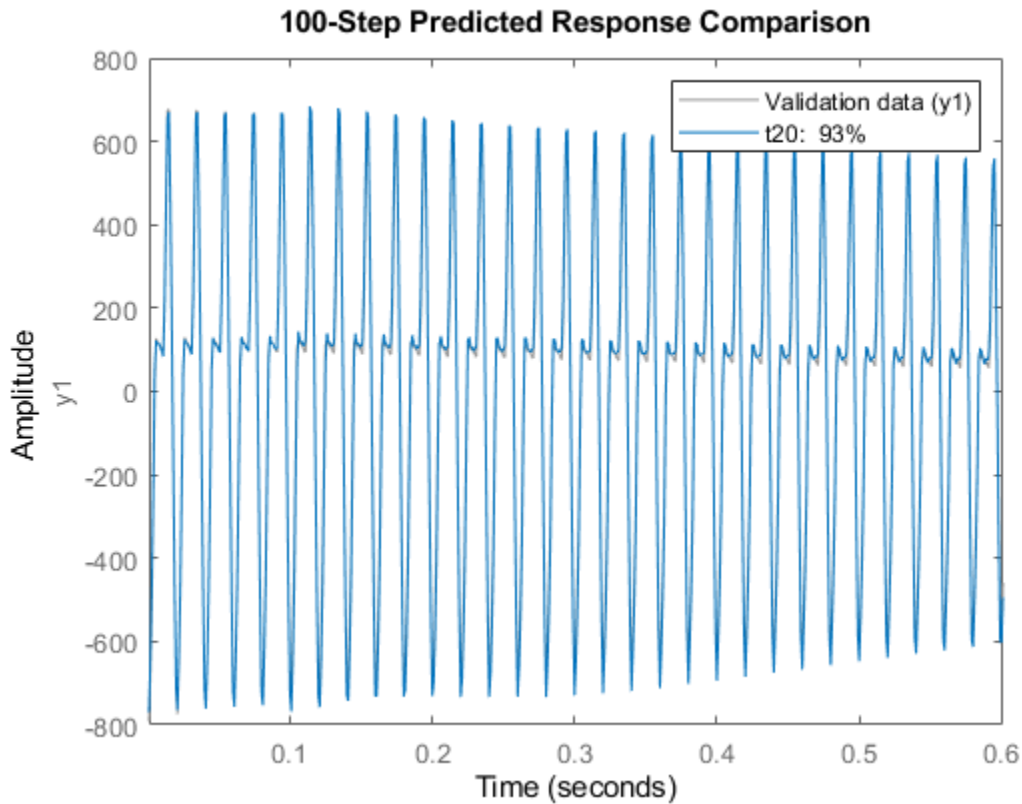
We see that this model picks up the harmonics very well. This model will predict 100 steps ahead as follows:

```
compare(i4r,t20,100,compareOptions('Samples',201:500));
```

100-Step Predicted Response Comparison

We now have a 93% fit with `t20`, as opposed to 80% for `t8f`.

We thus conclude that for a complete model of the signal, `t20` is the natural choice, both in terms of capturing the harmonics as well as in its prediction capabilities. For models in certain frequency ranges we can however do very well with lower order models, but we then have to prefilter the data accordingly.

Additional Information

For more information on identification of dynamic systems with System Identification Toolbox visit the System Identification Toolbox product information page.

Recursive Model Identification

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 phoneme.
- 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.

16

Online Estimation

What Is Online Estimation?

Online estimation algorithms estimate the parameters and states of a model when new data is available during the operation of the physical system. The System Identification Toolbox software uses linear, extended, and unscented Kalman filter, or particle filter algorithms for online state estimation. The toolbox uses recursive prediction error minimization algorithms for online parameter estimation.

Consider a heating and cooling system that does not have prior information about the environment in which it operates. Suppose that this system must heat or cool a room to achieve a certain temperature in a given amount of time. To fulfill its objective, the system must obtain knowledge of the temperature and insulation characteristics of the room. You can estimate the insulation characteristics of the room while the system is online (operational). For this estimation, use the system effort as the input and the room temperature as the output. You can use the estimated model to improve system behavior. Online estimation is ideal for estimating small deviations in the parameter values of a system at a known operating point.

Common applications of online estimation include:

- Adaptive control — Estimate a plant model to modify the controller based on changes in the plant model.
- Fault detection — Compare the online plant model with the idealized or reference plant model to detect a fault (anomaly) in the plant.
- Soft sensing — Generate a "measurement" based on the estimated plant model, and use this measurement for feedback control or fault detection.
- Verification of the experiment-data quality before starting offline estimation — Before using the measured data for offline estimation, perform online estimation for a few iterations. The online estimation provides a quick check of whether the experiment used excitation signals that captured the relevant system dynamics.

Online Parameter Estimation

Online parameter estimation is typically performed using a recursive algorithm. To estimate the parameter values at a time step, recursive algorithms use the current measurements and previous parameter estimates. Therefore, recursive algorithms are efficient in terms of memory usage. Also, recursive algorithms have smaller computational demands. This efficiency makes them suited to online and embedded applications. For more information about the algorithms, see "Recursive Algorithms for Online Parameter Estimation" on page 16-16.

In System Identification Toolbox you can perform online parameter estimation in Simulink or at the command line:

- In Simulink, use the Recursive Least Squares Estimator and Recursive Polynomial Model Estimator blocks to perform online parameter estimation. You can also estimate a state-space model online from these models by using the Recursive Polynomial Model Estimator and Model Type Converter blocks together. You can generate C/C++ code and Structured Text for these blocks using Simulink Coder and Simulink PLC Coder™ software.
- At the command line, use `recursiveAR`, `recursiveARMA`, `recursiveARX`, `recursiveARMAX`, `recursiveOE`, `recursiveBJ`, and `recursiveLS` commands to estimate model parameters for your model structure. Unlike estimation in Simulink, you can change the properties of the recursive estimation algorithm during online estimation. You can generate code and standalone applications using MATLAB Coder and MATLAB Compiler™ software.

When you perform online parameter estimation in Simulink or at the command line, the following requirements apply:

- Model must be discrete-time linear or nearly linear with parameters that vary slowly with time.
- Structure of the estimated model must be fixed during estimation.
- `iddata` object is not supported during online parameter estimation. Specify estimation output data as a real scalar and input data as a real scalar or vector.

Online State Estimation

You can perform online state estimation of systems at the command line and in Simulink:

- In Simulink, use the Kalman Filter, Extended Kalman Filter, Unscented Kalman Filter or Particle Filter blocks to perform online state estimation of discrete-time linear and nonlinear systems. You can generate C/C++ code for these blocks using Simulink Coder software. For the Kalman Filter block, you can also generate Structured Text using Simulink PLC Coder software.
- At the command line, use `extendedKalmanFilter`, `unscentedKalmanFilter` and `particleFilter` commands to estimate states of discrete-time nonlinear systems. These commands implement discrete-time extended Kalman filter (EKF), unscented Kalman filter (UKF) and particle filter algorithms. For more information about the algorithms, see "Extended and Unscented Kalman Filter Algorithms for Online State Estimation" on page 16-27. You can generate code and standalone applications using MATLAB Coder and MATLAB Compiler software.

When you perform online state estimation in Simulink or at the command line, the following requirements apply:

- System must be discrete-time. If you are using the Kalman Filter block, the system can also be continuous-time.
- `iddata` object is not supported during online state estimation. Specify estimation input-output data as real scalars or vectors.

References

[1] Ljung, L. *System Identification: Theory for the User*. Upper Saddle River, NJ: Prentice-Hall PTR, 1999, pp. 428–440.

[2] Simon, D. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. John Wiley and Sons Inc., 2006.

See Also

Functions
recursiveAR | recursiveARMA | recursiveARX | recursiveARMAX | recursiveOE | recursiveBJ | recursiveLS | extendedKalmanFilter | unscentedKalmanFilter | particleFilter

Blocks
Recursive Least Squares Estimator | Recursive Polynomial Model Estimator | Kalman Filter | Extended Kalman Filter | Unscented Kalman Filter | Particle Filter

Related Examples

How Online Parameter Estimation Differs from Offline Estimation

Online estimation algorithms estimate the parameters of a model when new data is available during the operation of the model. In offline estimation, you first collect all the input/output data and then estimate the model parameters . Parameter values estimated using online estimation can vary with time, but parameters estimated using offline estimation do not.

To perform offline estimation, use commands such as `arx`, `pem`, `ssest`, `tfest`, `nlarx`, and the System Identification app.

To perform online parameter estimation in Simulink, use the Recursive Least Squares Estimator and Recursive Polynomial Model Estimator blocks. For online estimation at the command line on page 16-21, use commands such as `recursiveARX` to create a System object™, and then use `step` command to update the model parameters.

Online estimation differs from offline estimation in the following ways:

- Model delays — You can estimate model delays in offline estimation using tools such as `delayest` (see "Determining Model Order and Delay" on page 4-39). Online estimation provides limited ability to estimate delays. For polynomial model estimation using the Recursive Polynomial Model Estimation block or the online estimation commands, you can specify a known value of the input delay (nk). If nk is unknown, choose a sufficiently large value for the number of coefficients of B (nb). The number of leading coefficients of the estimated B polynomial that are close to zero represent the input delay.

- Data preprocessing — For offline estimation data preprocessing, you can use functions such as `detrend`, `retrend`, `idfilt`, and the System Identification app.

  For online estimation using Simulink, use the tools available in the Simulink environment. For more information, see "Preprocess Online Parameter Estimation Data in Simulink" on page 16-7.

  For online parameter estimation at the command line, you cannot use preprocessing tools in System Identification Toolbox. These tools support only data specified as `iddata` objects. Implement preprocessing code as required by your application. To be able to generate C and C++ code, use commands supported by MATLAB Coder. For a list of these commands, see "Functions and Objects Supported for C/C++ Code Generation" (MATLAB Coder).

- Resetting of estimation — You cannot reset offline estimation. Online estimation lets you reset the estimation at a specific time step during estimation. For example, reset the estimation when the system changes modes or if you are not satisfied with the estimation. The reset operation sets the model states, estimated parameters, and estimated parameter covariance to their initial values.

  To reset online estimation in Simulink, on the **Algorithm and Block Options** tab of the block parameters, select the appropriate **External reset** option. At the command line, use the `reset` command.

- Enabling or disabling of estimation — You cannot selectively enable or disable offline estimation. You can use preprocessing tools to remove or filter certain portions of the data before the estimation. Online estimation lets you enable or disable estimation for chosen time spans. For example, suppose that the measured data is especially noisy or faulty (contains many outliers) for a specific time interval. Disable online estimation for this interval.

To enable or disable estimation in Simulink, on the **Algorithm and Block Options** tab of the block parameters, select the **Add enable port** check box.

At the command line, use the `EnableAdaptation` property of the System object created using online estimation commands, such as `recursiveARMAX` and `recursiveLS`. Even if you set `EnableAdaptation` to `false`, execute the `step` command. Do not skip `step` to keep parameter values constant because parameter estimation depends on current and past input/output measurements. `step` ensures that past input-output data is stored, even when it does not update the parameters.

See Also

More About
- "What Is Online Estimation?" on page 16-2

Preprocess Online Parameter Estimation Data in Simulink

Estimation data that contains deficiencies, such as drift, offset, missing samples, seasonalities, equilibrium behavior, and outliers, can adversely affect the quality of the estimation. Therefore, it is recommended that you preprocess your estimation data as needed.

Use the tools in the Simulink software to preprocess data for online estimation. Common tools to perform data preprocessing in Simulink are:

- Blocks in the Math Operations (Simulink) library. Use these blocks, for example, to subtract or add an offset or normalize a signal.
- Blocks in the "Continuous" (Simulink) and "Discrete" (Simulink) library. Use these blocks, for example, to filter a signal.
- Rate Transition block, which allows you to handle the transfer of data between blocks operating at different rates. Use this block, for example, to resample your data from a source that is operating at a different sampling rate than the online estimation block.
- MATLAB Function block, which allows you to include MATLAB code in your model. Use this block, for example, to implement a custom preprocessing algorithm.

See "Online ARMAX Polynomial Model Estimation" on page 16-69 for an example of how you can preprocess an estimation input signal by removing its mean.

See Also
Recursive Least Squares Estimator | Recursive Polynomial Model Estimator | Kalman Filter

More About
- "What Is Online Estimation?" on page 16-2

Validate Online Parameter Estimation Results in Simulink

Use the following approaches to validate an online estimation performed using the Recursive Least Squares Estimator or Recursive Polynomial Model Estimator block:

- Examine the estimation error (residuals), which is the difference between the measured and estimated outputs. The estimation error can be large at the beginning of the estimation or when there are large parameter variations. The error should get smaller as the parameter estimates converge. Small errors with respect to the size of the outputs give confidence in the estimated values.

  You can also analyze the residuals using techniques such as the whiteness test and the independence test. For such analysis, use the measured data and estimation error collected after the parameter values have settled to approximately constant values. For more information regarding these tests, see "What Is Residual Analysis?" on page 17-43

  To obtain the estimation error, in the **Algorithm and Block Options** tab, select the **Output estimation error** check box. The software adds an Error outport to the block, which you can monitor using a Scope block. This outport provides the one-step-ahead estimation error, $e(t) = y(t)–y_{est}(t)$. For the time step, t, y and y_{est} are the measured and estimated outputs, respectively.

- The parameter covariance is a measure of estimated uncertainty in the parameters, and is calculated when the forgetting factor or Kalman filter estimation algorithms are used.

  Parameter covariance is computed assuming that the residuals are white noise, and the variance of these residuals is 1. To obtain the parameter covariance, in the **Algorithm and Block Options** tab of the online estimation block parameters, select the **Output parameter covariance matrix** check box. The software adds a Covariance outport to the block, which you can monitor using a Display block. This outport provides the parameter covariance matrix, P.

  The estimated parameters can be considered as random variables with variance equal to the corresponding diagonal of the parameter covariance matrix, scaled by the variance of the residuals (`residualVariance`) at each time step. You use prior knowledge, or calculate `residualVariance` from the residuals, e. Where, e is the vector of estimation errors, $e(t)$.

  ```
  residualVariance = var(e);
  ```

  Scale the parameter covariance to calculate the variance of the estimated parameters.

  ```
  paramVariance = diag(P)*residualVariance;
  ```

  A smaller variance value gives confidence in the estimated values.

- Simulate the estimated model and compare the simulated and measured outputs. To do so, feed the measured input into a model that uses the estimated time-varying parameter values. Then, compare the model output with the measured output. The simulated output closely matching the measured output gives confidence in the estimated values.

For examples of such validation, see "Online Recursive Least Squares Estimation" on page 16-61 and "Online ARMAX Polynomial Model Estimation" on page 16-69.

If the validation indicates low confidence in the estimation, then refer to the Troubleshooting topics on the "Online Estimation" page.

See Also

Recursive Least Squares Estimator | Recursive Polynomial Model Estimator | Kalman Filter

Validate Online Parameter Estimation at the Command Line

This topic shows how to validate online parameter estimation at the command line. If the validation indicates low confidence in the estimation, then see "Troubleshoot Online Parameter Estimation" on page 16-12. After you have validated the online estimation results, you can generate C/C++ code or a standalone application using MATLAB Coder or MATLAB Compiler.

Examine the Estimation Error

The estimation error is the difference between measured output, y, and the estimated output, EstimatedOutput at each time step.

```
obj = recursiveARX;
[A,B,EstimatedOutput] = step(obj,y,u);
estimationError = y-EstimatedOutput;
```

Here, u is the input data at that time step.

The estimation errors (*residuals*) can be large at the beginning of the estimation or when there are large parameter variations. The error should get smaller as the parameter estimates converge. Small errors relative to the size of the outputs increase confidence in the estimated values.

Simulate the Estimated Model

Simulate the estimated model and compare the simulated and measured outputs. To do so, feed the measured input into a model that uses the estimated time-varying parameter values. Then compare the model output with the measured output. A close match between the simulated output and the measured output gives confidence in the estimated values.

Examine Parameter Covariance

Parameter covariance is a measure of estimated uncertainty in the parameters. The covariance is calculated when the forgetting factor or Kalman filter estimation algorithms are used.

Parameter covariance is computed assuming that the residuals are white noise, and the variance of these residuals is 1. You view the parameter covariance matrix using the ParameterCovariance property of your System object.

```
P = obj.ParameterCovariance;
```

The estimated parameters can be considered as random variables with variance equal to the corresponding diagonal of the parameter covariance matrix, scaled by the variance of the residuals (residualVariance) at each time step. You use prior knowledge, or calculate residualVariance from the residuals, e. Where, e is the vector of estimation errors, estimationError.

```
 residualVariance = var(e);
```

Scale the parameter covariance to calculate the variance of the estimated parameters.

```
paramVariance = diag(P)*residualVariance;
```

A smaller variance value gives confidence in the estimated values.

Use Validation Commands from System Identification Toolbox

You can validate a snapshot of the estimated model using validation commands for offline estimation. This validation only captures the behavior of a time-invariant model. For available offline validation techniques in System Identification Toolbox, see "Model Validation".

To use offline commands, convert your online estimation System object, `obj`, into an `idpoly` model object. Also convert your stream of input-output validation data, *Output(t)* and *Input(t)*, into an `iddata` object.

```
sys = idpoly(obj);
sys.Ts = Ts;
z = iddata(Output,Input,Ts)
```

Here, `Ts` is the sample time.

Note This conversion and any subsequent analysis are not supported by MATLAB Coder.

The validation techniques include:

*   Analysis of the residuals using techniques such as the whiteness test and the independence test using offline commands such as `resid`. For example, use `resid(z,sys)`. For information about these tests, see "What Is Residual Analysis?" on page 17-43.
*   Comparison of model output and measured output. For example, use `compare(z,sys)`.
*   Comparison of different online estimation System objects.

    You can create multiple versions of a System object with different object properties, convert each of them to `idpoly` model objects, and use `compare` to choose the best one.

    If you want to copy an existing System object and then modify properties of the copied object, use the `clone` command. Do not create additional objects using syntax `obj2 = obj`. Any changes made to the properties of the new System object created this way (`obj2`) also change the properties of the original System object (`obj`).

See Also
recursiveAR | recursiveARMA | recursiveARX | recursiveARMAX | recursiveOE | recursiveBJ | recursiveLS

Related Examples
*   "Perform Online Parameter Estimation at the Command Line" on page 16-21
*   "Generate Code for Online Parameter Estimation in MATLAB" on page 16-24
*   "Line Fitting with Online Recursive Least Squares Estimation" on page 16-43
*   "Online ARX Parameter Estimation for Tracking Time-Varying System Dynamics" on page 16-50

Troubleshoot Online Parameter Estimation

To troubleshoot online parameter estimation, check the following:

Model Structure

Check that you are using the simplest model structure that adequately captures the system dynamics.

AR and ARX model structures are good first candidates for estimating linear models. The underlying estimation algorithms for these model structures are simpler than those for ARMA, ARMAX, Output-Error, and Box-Jenkins model structures. In addition, these simpler AR and ARX algorithms are less sensitive to initial parameter guesses.

The more generic recursive least squares (RLS) estimation also has the advantage of algorithmic simplicity like AR and ARX model estimation. RLS lets you estimate parameters for a wider class of models than ARX and AR and can include nonlinearities. However, configuring an AR or ARX structure is simpler.

Consider the following when choosing a model structure:

- AR and ARX model structures — If you are estimating a time-series model (no inputs), try the AR model structure. If you are estimating an input-output model, try the ARX model structure. Also try different model orders with these model structures. These models estimate the system output based on time-shifted versions of the output and inputs signals. For example, the a and b parameters of the system $y(t) = b_1u(t)+b_2u(t\text{-}1)\text{-}a_1y(t\text{-}1)$ can be estimated using ARX models.

  For more information regarding AR and ARX models, see "What Are Polynomial Models?" on page 6-2.

- RLS estimation— If you are estimating a system that is linear in the estimated parameters, but does not fit into AR or ARX model structures, try RLS estimation. You can estimate the system output based on the time-shifted versions of input-outputs signals like the AR and ARX, and can also add nonlinear functions. For example, you can estimate the parameters p_1, p_2, and p_3 of the system $y(t) = p_1y(t\text{-}1) + p_2u(t\text{-}1) + p_3u(t\text{-}1)^2$. You can also estimate static models, such as the line-fit problem of estimating parameters a and b in $y(t) = au(t)+b$.

- ARMA, ARMAX, Output-Error, Box-Jenkins model structures — These model structures provide more flexibility compared to AR and ARX model structures to capture the dynamics of linear systems. For instance, an ARMAX model has more dynamic elements (C polynomial parameters) compared to ARX for estimating noise models. This flexibility can help when AR and ARX are not sufficient to capture the system dynamics of interest.

  Specifying initial parameter and parameter covariance values are especially recommended for these model structures. This is because the estimation method used for these model structures can get stuck at a local optima. For more information about these models, see "What Are Polynomial Models?" on page 6-2.

Model Order

Check the order of your specified model structure. You can under-fit (model order is too low) or over-fit (model order is too high) data by choosing an incorrect model order.

Ideally, you want the lowest-order model that adequately captures your system dynamics. Under-fitting prevents algorithms from finding a good fit to the model, even if all other estimation settings

are good, and there is good excitation of system dynamics. Over-fitting typically leads to high sensitivity of parameters to the measurement noise or the choice of input signals.

Estimation Data

Use inputs that excite the system dynamics adequately. Simple inputs, such as a step input, typically does not provide sufficient excitation and are good for estimating only a very limited number of parameters. One solution is to inject extra input perturbations.

Estimation data that contains deficiencies can lead to poor estimation results. Data deficiencies include drift, offset, missing samples, equilibrium behavior, seasonalities, and outliers. It is recommended that you preprocess the estimation data as needed.

For information on how to preprocess estimation data in Simulink, see "Preprocess Online Parameter Estimation Data in Simulink" on page 16-7.

For online parameter estimation at the command line, you cannot use preprocessing tools in System Identification Toolbox. These tools support only data specified as `iddata` objects. Implement preprocessing code as required by your application. To be able to generate C and C++ code, use commands supported by MATLAB Coder. For a list of these commands, see "Functions and Objects Supported for C/C++ Code Generation" (MATLAB Coder).

Initial Guess for Parameter Values

Check the initial guesses you specify for the parameter values and initial parameter covariance matrix. Specifying initial parameter guesses and initial parameter covariance matrix is recommended. These initial guesses could be based on your knowledge of the system or be obtained via offline estimation.

Initial parameter covariance represents the uncertainty in your guess for the initial values. When you are confident about your initial parameter guesses, and if the initial parameter guesses are much smaller than the default initial parameter covariance value, `10000`, specify a smaller initial parameter covariance. Typically, the default initial parameter covariance is too large relative to the initial parameter values. The result is that initial guesses are given less importance during estimation.

Initial parameter and parameter covariance guesses are especially important for ARMA, ARMAX, Output-Error, and Box-Jenkins models. Poor or no guesses can result in the algorithm finding a local minima of the objective function in the parameter space, which can lead to a poor fit.

Estimation Settings

Check that you have specified appropriate settings for the estimation algorithm. For example, for the forgetting factor algorithm, choose the forgetting factor, λ, carefully. If λ is too small, the estimation algorithm assumes that the parameter value is varying quickly with time. Conversely, if λ is too large, the estimation algorithm assumes that the parameter value does not vary much with time. For more information regarding the estimation algorithms, see "Recursive Algorithms for Online Parameter Estimation" on page 16-16.

See Also

Related Examples

- "What Is Online Estimation?" on page 16-2
- "Validate Online Parameter Estimation Results in Simulink" on page 16-8
- "Validate Online Parameter Estimation at the Command Line" on page 16-10

Generate Online Parameter Estimation Code in Simulink

You can generate C/C++ code and Structured Text for Recursive Least Squares Estimator and other online estimation blocks using products such as Simulink Coder and Simulink PLC Coder. The Model Type Converter block, which you can use with the Recursive Polynomial Model Estimator block, also supports code generation. Use the generated code to deploy online model estimation to an embedded target. For example, you can estimate the coefficients of a time-varying plant from measured input-output data and feed the coefficients to an adaptive controller. After validating the online estimation in simulation, you can generate code for your Simulink model and deploy that code to the target.

To generate code for online estimation, use the following workflow:

1 Develop a Simulink model that simulates the online model estimation. For example, create a model that simulates the input/output data, performs online estimation for this data, and uses the estimated parameter values.

2 After validating the online estimation performance in simulation, create a subsystem for the online estimation block. If you preprocess the inputs or postprocess the parameter estimates, include the relevant blocks in the subsystem.

3 Convert the subsystem to a referenced model. You generate code for this referenced model, so ensure that it uses only the blocks that support code generation. For a list of blocks that support code generation, see "Simulink Built-In Blocks That Support Code Generation" (Simulink Coder).

The original model, which now contains a model reference, is now referred to as the top model.

4 In the top model, replace the model source and sink blocks with their counterpart hardware blocks. For example, replace the simulated inputs/output blocks with the relevant hardware source block. You generate code for this model, which includes the online estimation. So, ensure that it uses only blocks that support code generation.

5 Generate code for the top model.

See Also

Recursive Least Squares Estimator | Recursive Polynomial Model Estimator | Model Type Converter | Kalman Filter

Related Examples

• "Generate Code for Model Reference Hierarchy" (Simulink Coder)

More About

• "Simulink Built-In Blocks That Support Code Generation" (Simulink Coder)

Recursive Algorithms for Online Parameter Estimation

The recursive estimation algorithms in the System Identification Toolbox can be separated into two categories:

- Infinite-history algorithms — These algorithms aim to minimize the error between the observed and predicted outputs for all time steps from the beginning of the simulation. The System Identification Toolbox supports infinite-history estimation in:

  - Recursive command-line estimators for the least-squares linear regression, AR, ARX, ARMA, ARMAX, OE, and BJ model structures
  - Simulink Recursive Least Squares Estimator and Recursive Polynomial Model Estimator blocks

- Finite-history algorithms — These algorithms aim to minimize the error between the observed and predicted outputs for a finite number of past time steps. The toolbox supports finite-history estimation for linear-in-parameters models:

  - Recursive command-line estimators for the least-squares linear regression, AR, ARX, and OE model structures
  - Simulink Recursive Least Squares Estimator block
  - Simulink Recursive Polynomial Model Estimator block, for AR, ARX, and OE structures only

  Finite-history algorithms are typically easier to tune than the infinite-history algorithms when the parameters have rapid and potentially large variations over time.

Recursive Infinite-History Estimation

General Form of Infinite-History Recursive Estimation

The general form of the infinite-history recursive estimation algorithm is as follows:

$$\widehat{\theta}(t) = \widehat{\theta}(t-1) + K(t)\big(y(t) - \widehat{y}(t)\big)$$

$\widehat{\theta}(t)$ is the parameter estimate at time t. $y(t)$ is the observed output at time t, and $\widehat{y}(t)$ 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 $y(t) - \widehat{y}(t)$ affects the update of the parameter estimate. The estimation algorithms minimize the prediction-error term $y(t) - \widehat{y}(t)$.

The gain has the following form:

$$K(t) = Q(t)\psi(t)$$

The recursive algorithms supported by the System Identification Toolbox product differ based on different approaches for choosing the form of $Q(t)$ and computing $\psi(t)$. Here, $\psi(t)$ represents the gradient of the predicted model output $\widehat{y}(t|\theta)$ 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:

$$\widehat{y}(t) = \psi^T(t)\widehat{\theta}(t-1)$$

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 $\widehat{\theta}(t-1)$. To learn how you can compute approximation for $\psi(t)$ and $\widehat{\theta}(t-1)$ for general model structures, see the section on recursive prediction-error methods in [1].

Types of Infinite-History Recursive Estimation Algorithms

The System Identification Toolbox software provides the following infinite-history recursive estimation algorithms for online estimation:

- "Forgetting Factor" on page 16-17
- "Kalman Filter" on page 16-18
- "Normalized and Unnormalized Gradient" on page 16-19

The forgetting factor and Kalman Filter formulations are more computationally intensive than gradient and unnormalized gradient methods. However, they typically have better convergence properties.

Forgetting Factor

The following set of equations summarizes the *forgetting factor* adaptation algorithm:

$$\widehat{\theta}(t) = \widehat{\theta}(t-1) + K(t)(y(t) - \widehat{y}(t))$$

$$\widehat{y}(t) = \psi^T(t)\widehat{\theta}(t-1)$$

$$K(t) = Q(t)\psi(t)$$

$$Q(t) = \frac{P(t-1)}{\lambda + \psi^T(t)P(t-1)\psi(t)}$$

$$P(t) = \frac{1}{\lambda}\left(P(t-1) - \frac{P(t-1)\psi(t)\psi(t)^T P(t-1)}{\lambda + \psi(t)^T P(t-1)\psi(t)}\right)$$

The software ensures *P(t)* is a positive-definite matrix by using a square-root algorithm to update it [2]. The software computes P assuming that the residuals (difference between estimated and measured outputs) are white noise, and the variance of these residuals is 1. R_2/2 * P is approximately equal to the covariance matrix of the estimated parameters, where R_2 is the true variance of the residuals.

Q(t) is obtained by minimizing the following function at time *t*:

$$\sum_{k=1}^{t}\lambda^{t-k}(y(k) - \widehat{y}(k))^2$$

See section 11.2 in [1] for details.

This approach discounts old measurements exponentially such that an observation that is τ samples old carries a weight that is equal to λ^τ times the weight of the most recent observation. $\tau = \frac{1}{1 - \lambda}$ represents the *memory horizon* of this algorithm. Measurements older than $\tau = \frac{1}{1 - \lambda}$ typically carry a weight that is less than about 0.3.

λ is called the forgetting factor and typically has a positive value between `0.98` and `0.995`. Set $\lambda = 1$ to estimate time-invariant (constant) parameters. Set $\lambda < 1$ to estimate time-varying parameters.

Note The forgetting factor algorithm for $\lambda = 1$ is equivalent to the Kalman filter algorithm with $R_1=0$ and $R_2=1$. For more information about the Kalman filter algorithm, see "Kalman Filter" on page 16-18.

Kalman Filter

The following set of equations summarizes the *Kalman filter* adaptation algorithm:

$$\widehat{\theta}(t) = \widehat{\theta}(t-1) + K(t)\big(y(t) - \widehat{y}(t)\big)$$

$$\widehat{y}(t) = \psi^T(t)\widehat{\theta}(t-1)$$

$$K(t) = Q(t)\psi(t)$$

$$Q(t) = \frac{P(t-1)}{R_2 + \psi^T(t)P(t-1)\psi(t)}$$

$$P(t) = P(t-1) + R_1 - \frac{P(t-1)\psi(t)\psi(t)^T P(t-1)}{R_2 + \psi(t)^T P(t-1)\psi(t)}$$

The software ensures *P(t)* is a positive-definite matrix by using a square-root algorithm to update it [2]. The software computes P assuming that the residuals (difference between estimated and measured outputs) are white noise, and the variance of these residuals is 1. R_2* P is the covariance matrix of the estimated parameters, and R_1 /R_2 is the covariance matrix of the parameter changes. Where, R_1 is the covariance matrix of parameter changes that you specify.

This formulation assumes the linear-regression form of the model:

$$y(t) = \psi^T(t)\theta_0(t) + e(t)$$

Q(t) is computed using a Kalman filter.

This formulation also assumes that the true parameters $\theta_0(t)$ are described by a random walk:

$$\theta_0(t) = \theta_0(t-1) + w(t)$$

w(t) is Gaussian white noise with the following covariance matrix, or *drift matrix R_1*:

$$Ew(t)w^T(t) = R_1$$

R_2 is the variance of the innovations *e(t)* in the following equation:

$$y(t) = \psi^T(t)\theta_0(t) + e(t)$$

The Kalman filter algorithm is entirely specified by the sequence of data $y(t)$, the gradient $\psi(t)$, R_1, R_2, and the initial conditions $\theta(t = 0)$ (initial guess of the parameters) and $P(t = 0)$ (covariance matrix that indicates parameters errors).

Note It is assumed that R_1 and $P(t = 0)$ matrices are scaled such that $R_2 = 1$. This scaling does not affect the parameter estimates.

Normalized and Unnormalized Gradient

In the linear regression case, the gradient methods are also known as the *least mean squares* (LMS) methods.

The following set of equations summarizes the *unnormalized gradient* and *normalized gradient* adaptation algorithm:

$$\widehat{\theta}(t) = \widehat{\theta}(t - 1) + K(t)(y(t) - \widehat{y}(t))$$

$$\widehat{y}(t) = \psi^T(t)\widehat{\theta}(t - 1)$$

$$K(t) = Q(t)\psi(t)$$

In the unnormalized gradient approach, $Q(t)$ is given by:

$$Q(t) = \gamma$$

In the normalized gradient approach, $Q(t)$ is given by:

$$Q(t) = \frac{\gamma}{|\psi(t)|^2 + Bias}$$

The normalized gradient algorithm scales the adaptation gain, γ, at each step by the square of the two-norm of the gradient vector. If the gradient is close to zero, this can cause jumps in the estimated parameters. To prevent these jumps, a bias term is introduced in the scaling factor.

These choices of $Q(t)$ for the gradient algorithms update the parameters in the negative gradient direction, where the gradient is computed with respect to the parameters. See pg. 372 in [1] for details.

Recursive Finite-History Estimation

The finite-history estimation methods find parameter estimates $\theta(t)$ by minimizing

$$\sum_{k = t - N + 1}^{t} (y(k) - \widehat{y}(k \mid \theta))^2,$$

where $y(k)$ is the observed output at time k, and $\widehat{y}(k \mid \theta)$ is the predicted output at time k. This approach is also known as sliding-window estimation. Finite-history estimation approaches minimize prediction errors for the last N time steps. In contrast, infinite-history estimation methods minimize prediction errors starting from the beginning of the simulation.

The System Identification Toolbox supports finite-history estimation for the linear-in-parameters models (AR and ARX) where predicted output has the form $\widehat{y}(k \mid \theta) = \Psi(k)\theta(k - 1)$. The software

constructs and maintains a buffer of regressors $\psi(k)$ and observed outputs $y(k)$ for $k = t\text{-}N\text{+}1$, $t\text{-}N\text{+}2$, ... , $t\text{-}2$, $t\text{-}1$, t. These buffers contain the necessary matrices for the underlying linear regression problem of minimizing $\|\Psi_{buffer}\theta - y_{buffer}\|_2^2$ over θ. The software solves this linear regression problem using QR factoring with column pivoting.

References

[1] Ljung, L. *System Identification: Theory for the User*. Upper Saddle River, NJ: Prentice-Hall PTR, 1999.

[2] Carlson, N.A. "Fast triangular formulation of the square root filter." *AIAA Journal*, Vol. 11, Number 9, 1973, pp. 1259-1265.

[3] Zhang, Q. "Some Implementation Aspects of Sliding Window Least Squares Algorithms." *IFAC Proceedings*. Vol. 33, Issue 15, 2000, pp. 763-768.

See Also

Recursive Least Squares Estimator | Recursive Polynomial Model Estimator | `recursiveAR` | `recursiveARMA` | `recursiveARX` | `recursiveARMAX` | `recursiveOE` | `recursiveBJ` | `recursiveLS`

More About

- "What Is Online Estimation?" on page 16-2

Perform Online Parameter Estimation at the Command Line

This topic shows how to perform online parameter estimation at the command line. The online estimation commands create a System object on page 16-21 for your model structure.

Online Estimation System Object

A System object is a specialized MATLAB object designed specifically for implementing and simulating dynamic systems with inputs that change over time. System objects use internal states to store past behavior, which is used in the next computational step.

After you create a System object, you use commands to process data or obtain information from or about the object. System objects use a minimum of two commands to process data — a constructor to create the object and the `step` command to update object parameters using real-time data. This separation of declaration from execution lets you create multiple, persistent, reusable objects, each with different settings.

You can use the following commands with the online estimation System objects in System Identification Toolbox:

Command	Description
`step`	Update model parameter estimates using recursive estimation algorithms and real-time data. `step` puts the object into a locked state. In a locked state, you cannot change any nontunable properties or input specifications, such as model order, data type, or estimation algorithm. During execution, you can only change tunable properties.
`release`	Unlock the System object. Use this command to enable setting of nontunable parameters.
`reset`	Reset the internal states of a locked System object to the initial values, and leave the object locked.
`clone`	Create another System object with the same object property values. Do not create additional objects using syntax `obj2 = obj`. Any changes made to the properties of the new object created this way (`obj2`) also change the properties of the original object (`obj`).
`isLocked`	Query locked status for input attributes and nontunable properties of the System object.

Note If all data necessary for estimation is available at once, and you are estimating a time-invariant model, use the offline estimation commands for model parameter estimation. For example, use `arx` instead of `recursiveARX`.

Workflow for Online Parameter Estimation at the Command Line

1   Choose a model structure for your application.

Ideally, you want the simplest model structure that adequately captures the system dynamics. For considerations to keep in mind, see "Model Structure" on page 16-12.

2   Create an online estimation System object for your model structure by using one of the following commands:

- `recursiveAR` — Time-series AR model
- `recursiveARMA` — Time-series ARMA model
- `recursiveARX` — SISO or MISO ARX model
- `recursiveARMAX` — SISO ARMAX model
- `recursiveOE` — SISO output-error polynomial model
- `recursiveBJ` — SISO Box-Jenkins polynomial model
- `recursiveLS` — Single-output system that is linear in estimated parameters

```
obj = recursiveARX;
```

You can specify additional object properties such as the recursive estimation algorithm and initial parameter guesses. For information about the algorithms used, see "Recursive Algorithms for Online Parameter Estimation" on page 16-16.

3   Acquire input-output data in real time.

Specify estimation output data, `y`, as a real scalar, and input data, `u`, as a real scalar or vector. Data specified as an `iddata` object is not supported for online estimation.

4   Preprocess the estimation data.

Estimation data that contains deficiencies can lead to poor estimation results. Data deficiencies include drift, offset, missing samples, equilibrium behavior, seasonalities, and outliers. Preprocess the estimation data as needed. For considerations to keep in mind, see "Estimation Data" on page 16-13.

For online parameter estimation at the command line, you cannot use preprocessing tools in System Identification Toolbox. These tools support only data specified as `iddata` objects. Implement preprocessing code as required by your application. To be able to generate C and C++ code, use commands supported by MATLAB Coder. For a list of these commands, see "Functions and Objects Supported for C/C++ Code Generation" (MATLAB Coder).

5   Update the parameters of the model using incoming input-output data.

Use the `step` command to execute the specified recursive algorithm over each measurement of input-output data.

```
[A,B,yhat] = step(obj,y,u);
```

The output of the `step` command gives the estimated parameters (A and B), and estimated model output (`yhat`), at each set of input-output data.

Calling `step` on an object puts that object into a locked state. You can check the locked status of a System object using `isLocked`. When the object is locked, you cannot change any nontunable properties or input specifications such as model order, data type, or estimation algorithm. To

change a nontunable property, use the `release` command to unlock the System object. You can use `release` on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

6 Post-process estimated parameters.

If necessary, you can post-process the estimated parameters. For instance, you can use a low-pass filter to smooth out noisy parameter estimates. To be able to generate C and C++ code, use commands supported by MATLAB Coder. For a list of these commands, see "Functions and Objects Supported for C/C++ Code Generation" (MATLAB Coder).

7 Validate the online estimation.

For details about the validation, see "Validate Online Parameter Estimation at the Command Line" on page 16-10. If you are not satisfied with the estimation, use the `reset` command to set the parameters of the System object to their initial value.

8 Use the estimated parameters for your application.

After validating the online parameter estimation, you can use MATLAB Compiler or MATLAB Coder to deploy the code in your application.

See Also
`recursiveAR` | `recursiveARMA` | `recursiveARX` | `recursiveARMAX` | `recursiveOE` | `recursiveBJ` | `recursiveLS` | `step` | `reset` | `release` | `isLocked` | `clone`

Related Examples
- "Validate Online Parameter Estimation at the Command Line" on page 16-10
- "Generate Code for Online Parameter Estimation in MATLAB" on page 16-24
- "Line Fitting with Online Recursive Least Squares Estimation" on page 16-43
- "Online ARX Parameter Estimation for Tracking Time-Varying System Dynamics" on page 16-50

More About
- "What Is Online Estimation?" on page 16-2
- "How Online Parameter Estimation Differs from Offline Estimation" on page 16-5
- "Recursive Algorithms for Online Parameter Estimation" on page 16-16

Generate Code for Online Parameter Estimation in MATLAB

This topic shows how to generate C/C++ code from online estimation MATLAB code that uses a System object. C/C++ code is generated using the `codegen` command from MATLAB Coder. Use the generated code to deploy online estimation algorithms to an embedded target.

You can also deploy online estimation code by creating a standalone application using MATLAB Compiler. MATLAB Compiler software supports System objects for use inside MATLAB functions, but does not support System objects for use in MATLAB scripts.

For Simulink based workflows, use the online estimator blocks from System Identification Toolbox, such as Recursive Least Squares Estimator and Recursive Polynomial Model Estimator. You can generate C/C++ code and Structured Text for the online estimation blocks using Simulink Coder and Simulink PLC Coder.

Supported Online Estimation Commands

Code generation support is available for these online estimation System objects:

- `recursiveAR`
- `recursiveARMA`
- `recursiveARX`
- `recursiveARMAX`
- `recursiveOE`
- `recursiveBJ`
- `recursiveLS`

Code generation support is available only for the following System object commands:

- `step`
- `reset`
- `release`
- `isLocked`

Generate Code for Online Estimation

To generate code for online estimation:

1 Create a function to declare your System object as persistent, and initialize the object. You define the System object as persistent to maintain the object states between calls.

```
function [A,B,EstimatedOutput] = arxonline(output,input)
% Declare System object as persistent
persistent obj;
if isempty(obj)
    obj = recursiveARX([1 2 2],'EstimationMethod','Gradient');
end
[A,B,EstimatedOutput] = step(obj,output,input);
end
```

The function creates a System object for online estimation of an ARX model of order [1 2 1], using the unnormalized gradient algorithm, and estimation data, `input` and `output`. Save this function on the MATLAB path. Alternatively, you can specify the full path name for this function.

The persistent System object is initialized with condition `if isempty(obj)` to ensure that the object is initialized only once, when the function is called the first time. Subsequent calls to the function just execute the `step` command to update the estimated parameters. During initialization you specify the nontunable properties of the object, such as `EstimationMethod`, `Orders`, and `DataType`.

2 Generate C/C++ code and MEX-files using the `codegen` command from MATLAB Coder.

```
codegen arxonline -args {1,1}
```

The syntax `-args {1,1}` specifies a set of example arguments to your function. The example arguments set the dimensions and data types of the function arguments `output` and `input` as double-precision scalars.

3 Use the generated code.

- Use the generated C/C++ code to deploy online model estimation to an embedded target.
- Use the generated MEX-file for testing the compiled C/C++ code in MATLAB. The generated MEX-file is also useful for accelerating simulations of parameter estimation algorithms in MATLAB.

  Load the estimation data. In this example, use a static data set for illustration.

  ```
  load iddata3
  output = z3.y;
  input = z3.u;
  ```

  Update the model parameters by calling the generated MEX-file.

  ```
  for i = 1:numel(input)
      [A,B,EstimatedOutput] = arxonline_mex(output(i),input(i));
  end
  ```

Rules and Limitations When Using System Objects in Generated MATLAB Code

The following rules and limitations apply to using online estimation System objects when writing MATLAB code suitable for code generation.

Object Construction and Initialization

- If System objects are stored in persistent variables, initialize objects once by embedding the object handles in an `if` statement with a call to `isempty( )`.
- Set arguments to System object constructors as compile-time constants when using the `codegen` command. For more information, see `coder.Constant`.
- Do not initialize System objects properties with other MATLAB class objects as default values in code generation. Initialize these properties in the constructor.

Inputs and Outputs

- Do not change the data type of the System object inputs.

- Do not pass a System object as an example input argument to a function being compiled with `codegen`.
- Do not pass a System object to functions declared as extrinsic (functions called in interpreted mode) using the `coder.extrinsic` function. System objects returned from extrinsic functions and scope System objects that automatically become extrinsic can be used as inputs to another extrinsic function, but they do not generate code.

Cell Arrays

- Cell arrays cannot contain System objects.

Tunable and Nontunable Properties of System Objects

- The value assigned to a nontunable property must be a constant, and there can be at most one assignment to that property (including the assignment in the constructor).
- You can set the tunable properties of online estimation System objects at construction time or by using dot notation after that.

See Also
recursiveAR | recursiveARMA | recursiveARX | recursiveARMAX | recursiveOE | recursiveBJ | recursiveLS | step | reset | release | isLocked | codegen

Related Examples

More About

Extended and Unscented Kalman Filter Algorithms for Online State Estimation

You can use discrete-time extended and unscented Kalman filter algorithms for online state estimation of discrete-time nonlinear systems. If you have a system with severe nonlinearities, the unscented Kalman filter algorithm may give better estimation results. You can perform the state estimation in Simulink and at the command line. To perform the state estimation, you first create the nonlinear state transition function and measurement function for your system.

At the command line, you use the functions to construct the `extendedKalmanFilter` or `unscentedKalmanFilter` object for desired algorithm, and specify whether the process and measurement noise terms in the functions are additive or non-additive. After you create the object, you use the `predict` and `correct` commands to estimate the states using real-time data. For information about the order in which to execute these commands, see the `predict` and `correct` reference pages.

In Simulink, you specify these function in the Extended Kalman Filter and Unscented Kalman Filter blocks. You also specify whether the process and measurement noise terms in the functions are additive or non-additive. In the blocks, the software decides the order in which prediction and correction of state estimates is done.

Extended Kalman Filter Algorithm

The `extendedKalmanFilter` command and Extended Kalman Filter block implement the first-order discrete-time Kalman filter algorithm. Assume that the state transition and measurement equations for a discrete-time nonlinear system have non-additive process and measurement noise terms with zero mean and covariance matrices Q and R, respectively:

$$x[k + 1] = f(x[k], w[k], u_s[k])$$

$$y[k] = h(x[k], v[k], u_m[k])$$

$$w[k] \sim (0, Q[k])$$

$$v[k] \sim (0, R[k])$$

Here f is a nonlinear state transition function that describes the evolution of states x from one time step to the next. The nonlinear measurement function h relates x to the measurements y at time step k. These functions can also have additional input arguments that are denoted by u_s and u_m. The process and measurement noise are w and v, respectively. You provide Q and R.

In the block, the software decides the order of prediction and correction of state estimates. At the command line, you decide the order. For information about the order in which to execute these commands, see the `predict` and `correct` reference pages. Assuming that you implement the `correct` command before `predict`, the software implements the algorithm as follows:

1. Initialize the filter object with initial values of the state, `x[0]`, and state estimation error covariance matrix, P.

   $$\widehat{x}[0 \mid -1] = E(x[0])$$

   $$P[0 \mid -1] = E(x[0] - \widehat{x}[0 \mid -1])(^x$$

Here $\widehat{x}$ is the state estimate and $\widehat{x}[k_a|k_b]$ denotes the state estimate at time step k_a using measurements at time steps $0,1,\dots,k_b$. So $\widehat{x}[0|-1]$ is the best guess of the state value before you make any measurements. You specify this value when you construct the filter.

2   For time steps $k = 0,1,2,3,\dots$, perform the following:

a   Compute the Jacobian of the measurement function, and update the state and state estimation error covariance using the measured data, y[k]. At the command line, the correct command performs this update.

$$C[k] = \frac{\partial h}{\partial x}\Big|_{\widehat{x}[k|k-1]}$$

$$S[k] = \frac{\partial h}{\partial v}\Big|_{\widehat{x}[k|k-1]}$$

The software calculates these Jacobian matrices numerically unless you specify the analytical Jacobian.

$$K[k] = P[k|k-1]C[^k(^C$$

$$\widehat{x}[k|k] = \widehat{x}[k|k-1] + K[k](y[k] - h(\widehat{x}[k|k-1], 0, u_m[k]))$$

$$\widehat{P}[k|k] = P[k|k-1] - K[k]\,C[k]\,P[k|k-1]$$

Here K is the Kalman gain.

b   Compute the Jacobian of the state transition function, and predict the state and state estimation error covariance at the next time step. In the software, the predict command performs this prediction.

$$A[k] = \frac{\partial f}{\partial x}\Big|_{\widehat{x}[k|k]}$$

$$G[k] = \frac{\partial f}{\partial w}\Big|_{\widehat{x}[k|k]}$$

The software calculates these Jacobian matrices numerically unless you specify the analytical Jacobian. This numerical computation may increase processing time and numerical inaccuracy of the state estimation.

$$P[k+1|k] = A[k]P[k|k]A[^k + G[k]Q[k]G[^k$$

$$\widehat{x}[k+1|k] = f(\widehat{x}[k|k], 0, u_s[k])$$

The correct function uses these values in the next time step. For better numerical performance, the software uses the square-root factorization of the covariance matrices. For more information on this factorization, see [2] .

The Extended Kalman Filter block supports multiple measurement functions. These measurements can have different sample times as long as their sample time is an integer multiple of the state transition sample time. In this case, a separate correction step is performed corresponding to measurements from each measurement function.

The algorithm steps described previously assume that you have non-additive noise terms in the state transition and measurement functions. If you have additive noise terms in the functions, the changes in the algorithm are:

- If the process noise w is additive, that is the state transition equation has the form $x[k] = f(x[k-1], u_s[k-1]) + w[k-1]$, then the Jacobian matrix G[k] is an identity matrix.

- If the measurement noise v is additive, that is the measurement equation has the form $y[k] = h(x[k], u_m[k]) + v[k]$, then the Jacobian matrix S[k] is an identity matrix.

Additive noise terms in the state and transition functions reduce the processing time.

The first-order extended Kalman filter uses linear approximations to nonlinear state transition and measurement functions. As a result, the algorithm may not be reliable if the nonlinearities in your system are severe. The unscented Kalman filter algorithm may yield better results in this case.

Unscented Kalman Filter Algorithm

The unscented Kalman filter algorithm and Unscented Kalman Filter block use the unscented transformation to capture the propagation of the statistical properties of state estimates through nonlinear functions. The algorithm first generates a set of state values called sigma points. These sigma points capture the mean and covariance of the state estimates. The algorithm uses each of the sigma points as an input to the state transition and measurement functions to get a new set of transformed state points. The mean and covariance of the transformed points is then used to obtain state estimates and state estimation error covariance.

Assume that the state transition and measurement equations for an *M*-state discrete-time nonlinear system have additive process and measurement noise terms with zero mean and covariances *Q* and *R*, respectively:

$$x[k+1] = f(x[k], u_s[k]) + w[k]$$

$$y[k] = h(x[k], u_m[k]) + v[k]$$

$$w[k] \sim (0, Q[k])$$

$$v[k] \sim (0, R[k])$$

You provide the initial values of *Q* and *R* in the `ProcessNoise` and `MeasurementNoise` properties of the unscented Kalman filter object.

In the block, the software decides the order of prediction and correction of state estimates. At the command line, you decide the order. For information about the order in which to execute these commands, see the `predict` and `correct` reference pages. Assuming that you implement the `correct` command before `predict`, the software implements the algorithm as follows:

1   Initialize the filter object with initial values of the state, x[0], and state estimation error covariance, *P*.

$$\widehat{x}[0 \,|\, -1] = E(x[0])$$

$$P[0 \,|\, -1] = E(x[0] - \widehat{x}[0 \,|\, -1])(^x$$

Here $\widehat{x}$ is the state estimate and $\widehat{x}[k_a | k_b]$ denotes the state estimate at time step k_a using measurements at time steps 0,1,...,k_b. So $\widehat{x}[0 \,|\, -1]$ is the best guess of the state value before you make any measurements. You specify this value when you construct the filter.

2   For each time step *k*, update the state and state estimation error covariance using the measured data, y[k]. In the software, the `correct` command performs this update.

a   Choose the sigma points $\hat{x}^{(i)}[k \,|\, k - 1]$ at time step k.

$$\hat{x}^{(0)}[k \,|\, k - 1] = \hat{x}[k \,|\, k - 1]$$

$$\hat{x}^{(i)}[k \,|\, k - 1] \quad = \quad \hat{x}[k \,|\, k - 1] + \Delta x^{(i)} \qquad i = 1, \dots, 2M$$

$$\Delta x^{(i)} \quad = \quad (\sqrt{cP[k \,|\, k - 1]})_i \quad i = 1, \dots, M$$

$$\Delta x^{(M + i)} = \quad - (\sqrt{cP[k \,|\, k - 1]})_i \quad i = 1, \dots, M$$

Where $c = \alpha^2(M + \kappa)$ is a scaling factor based on number of states M, and the parameters α and κ. For more information about the parameters, see "Effect of Alpha, Beta, and Kappa Parameters" on page 16-32. $\sqrt{cP}$ is the matrix square root of cP such that $\sqrt{cP} \, (\sqrt{cP})^T = cP$ and $(\sqrt{cP})_i$ is the ith column of $\sqrt{cP}$.

b   Use the nonlinear measurement function to compute the predicted measurements for each of the sigma points.

$$\hat{y}^{(i)}[k \,|\, k - 1] = h(\hat{x}^{(i)}[k \,|\, k - 1], u_m[k]) \qquad i = 0, 1, \dots, 2M$$

c   Combine the predicted measurements to obtain the predicted measurement at time k.

$$\hat{y}[k] = \sum_{i = 0}^{2M} W_M^{(i)} \hat{y}^{(i)}[k \,|\, k - 1]$$

$$W_M^{(0)} = 1 - \frac{M}{\alpha^2(M + \kappa)}$$

$$W_M^i = \frac{1}{2\alpha^2(M + \kappa)} \qquad i = 1, 2, \dots, 2M$$

d   Estimate the covariance of the predicted measurement. Add R[k] to account for the additive measurement noise.

$$P_y = \sum_{i = 0}^{2M} W_c^{(i)}(\hat{y}^{(i)}[k \,|\, k - 1] - \hat{y}[k])(\hat{y}^{(i)} + R[k]$$

$$W_c^{(0)} = (2 - \alpha^2 + \beta) - \frac{M}{\alpha^2(M + \kappa)}$$

$$W_c^i = 1/(2\alpha^2(M + \kappa)) \qquad i = 1, 2, \dots, 2M$$

For information about β parameter, see "Effect of Alpha, Beta, and Kappa Parameters" on page 16-32.

e   Estimate the cross-covariance between $\hat{x}[k \,|\, k - 1]$ and $\hat{y}[k]$.

$$P_{xy} = \frac{1}{2\alpha^2(m + \kappa)} \sum_{i = 1}^{2M} (\hat{x}^{(i)}[k \,|\, k - 1] - \hat{x}[k \,|\, k - 1])(\hat{y}^{(i)}[k \,|\, k - 1] - \hat{y}[k])^T$$

The summation starts from $i = 1$ because $\hat{x}^{(0)}[k \,|\, k - 1] - \hat{x}[k \,|\, k - 1] = 0$.

f   Obtain the estimated state and state estimation error covariance at time step k.

$$K = P_{xy}P_y^{-1}$$

$$\widehat{x}[k\,|\,k] = \widehat{x}[k\,|\,k-1] + K(y[k] - \widehat{y}[k])$$

$$P[k\,|\,k] = P[k\,|\,k-1] - KP_yK_k^T$$

Here K is the Kalman gain.

3 Predict the state and state estimation error covariance at the next time step. In the software, the `predict` command performs this prediction.

a Choose the sigma points $\widehat{x}^{(i)}[k\,|\,k]$ at time step k.

$$\widehat{x}^{(0)}[k\,|\,k] = \widehat{x}[k\,|\,k]$$

$$\widehat{x}^{(i)}[k\,|\,k] = \widehat{x}[k\,|\,k] + \Delta x^{(i)} \qquad i = 1, ..., 2M$$

$$\Delta x^{(i)} = (\sqrt{cP[k\,|\,k]})_i \quad i = 1, ..., M$$

$$\Delta x^{(M+i)} = -(\sqrt{cP[k\,|\,k]})_i \quad i = 1, ..., M$$

b Use the nonlinear state transition function to compute the predicted states for each of the sigma points.

$$\widehat{x}^{(i)}[k+1\,|\,k] = f(\widehat{x}^{(i)}[k\,|\,k], u_s[k])$$

c Combine the predicted states to obtain the predicted states at time $k+1$. These values are used by the `correct` command in the next time step.

$$\widehat{x}[k+1\,|\,k] = \sum_{i=0}^{2M} W_M^{(i)}\widehat{x}^{(i)}[k+1\,|\,k]$$

$$W_M^{(0)} = 1 - \frac{M}{\alpha^2(M+\kappa)}$$

$$W_M^i = \frac{1}{2\alpha^2(M+\kappa)} \qquad i = 1, 2, ..., 2M$$

d Compute the covariance of the predicted state. Add `Q[k]` to account for the additive process noise. These values are used by the `correct` command in the next time step.

$$P[k+1\,|\,k] = \sum_{i=0}^{2M} W_c^{(i)}(\widehat{x}^{(i)}[k+1\,|\,k] - \widehat{x}[k+1\,|\,k])(\widehat{x}^{(i)}[k+1\,|\,k] - \widehat{x}[k+1\,|\,k])^T + Q[k]$$

$$W_c^{(0)} = (2 - \alpha^2 + \beta) - \frac{M}{\alpha^2(M+\kappa)}$$

$$W_c^i = 1/(2\alpha^2(M+\kappa)) \quad i = 1, 2, ..., 2M$$

The Unscented Kalman Filter block supports multiple measurement functions. These measurements can have different sample times as long as their sample time is an integer multiple of the state transition sample time. In this case, a separate correction step is performed corresponding to measurements from each measurement function.

The previous algorithm is implemented assuming additive noise terms in the state transition and measurement equations. For better numerical performance, the software uses the square-root factorization of the covariance matrices. For more information on this factorization, see [2] .

If the noise terms are non-additive, the main changes to the algorithm are:

- The `correct` command generates `2*(M+V)+1` sigma points using `P[k|k-1]` and `R[k]`, where `V` is the number of elements in measurement noise `v[k]`. The `R[k]` term is no longer added in the algorithm step 2(d) because the extra sigma points capture the impact of measurement noise on `Py`.

- The `predict` command generates `2*(M+W)+1` sigma points using `P[k|k]` and `Q[k]`, where `W` is the number of elements in process noise `w[k]`. The `Q[k]` term is no longer added in the algorithm step 3(d) because the extra sigma points capture the impact of process noise on `P[k+1|k]`.

Effect of Alpha, Beta, and Kappa Parameters

To compute the state and its statistical properties at the next time step, the unscented Kalman filter algorithm generates a set of state values distributed around the mean state value. The algorithm uses each sigma points as an input to the state transition and measurement functions to get a new set of transformed state points. The mean and covariance of the transformed points is then used to obtain state estimates and state estimation error covariance.

The spread of the sigma points around the mean state value is controlled by two parameters α and κ. A third parameter, β, impacts the weights of the transformed points during state and measurement covariance calculations.

- α — Determines the spread of the sigma points around the mean state value. It is usually a small positive value. The spread of sigma points is proportional to α. Smaller values correspond to sigma points closer to the mean state.

- κ — A second scaling parameter that is usually set to 0. Smaller values correspond to sigma points closer to the mean state. The spread is proportional to the square-root of κ.

- β — Incorporates prior knowledge of the distribution of the state. For Gaussian distributions, $\beta = 2$ is optimal.

You specify these parameters in the `Alpha`, `Kappa`, and `Beta` properties of the unscented Kalman filter. If you know the distribution of state and state covariance, you can adjust these parameters to capture the transformation of higher-order moments of the distribution. The algorithm can track only a single peak in the probability distribution of the state. If there are multiple peaks in the state distribution of your system, you can adjust these parameters so that the sigma points stay around a single peak. For example, choose a small `Alpha` to generate sigma points close to the mean state value.

References

[1] Simon, Dan. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. Hoboken, NJ: John Wiley and Sons, 2006.

[2] Van der Merwe, Rudolph, and Eric A. Wan. "The Square-Root Unscented Kalman Filter for State and Parameter-Estimation." *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings* (Cat. No.01CH37221), 6:3461–64. Salt Lake City, UT, USA: IEEE, 2001. https://doi.org/10.1109/ICASSP.2001.940586.

See Also

Functions
extendedKalmanFilter | unscentedKalmanFilter

Blocks
Extended Kalman Filter | Unscented Kalman Filter

External Websites

- Understanding Kalman Filters: Nonlinear State Estimators — MATLAB Video Series

Validate Online State Estimation at the Command Line

After you use the `extendedKalmanFilter`, `unscentedKalmanFilter` or `particleFilter` commands for online state estimation of a nonlinear system, validate the estimation before deploying the code in your application. If the validation indicates low confidence in the estimation, then see "Troubleshoot Online State Estimation" on page 16-42 for next steps. After you have validated the online estimation results, you can generate C/C++ code or a standalone application using MATLAB Coder or MATLAB Compiler software.

To validate the performance of your filter, perform state estimation using measured or simulated output data from different scenarios.

- Obtain output data from your system at different operating conditions and input values — To ensure that estimation works well under all operating conditions of interest. For example, suppose that you want to track the position and velocity of a vehicle using noisy position measurements. Measure the data at different vehicle speeds and slow and sharp maneuvers.

- For each operating condition of interest, obtain multiple sets of experimental or simulated data with different noise realizations — To ensure different noise values do not deteriorate estimation performance.

For each of these scenarios, test the filter performance by examining the output estimation error and state estimation error. For an example about performing and validating online state estimation, see "Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter" on page 16-99.

Examine Output Estimation Error

The output estimation error is the difference between the measured output, `y`, and the estimated output, `yEstimated`. You can obtain the estimated output at each time step by using the measurement function of the system. For example, if `vdpMeasurementFcn.m` is the measurement function for your nonlinear system, and you are performing state estimation using an extended Kalman filter object, `obj`, you can compute the estimated output using the current state estimates as:

```
yEstimated = vdpMeasurementFcn(obj.State);
estimationError = y-yEstimated;
```

Here `obj.State` is the state value $\hat{x}[k|k-1]$ after you estimate the states using the `predict` command. $\hat{x}[k|k-1]$ is the predicted state estimate for time `k`, estimated using measured output until a previous time `k-1`.

If you are using `extendedKalmanFilter` or `unscentedKalmanFilter`, you can also use `residual` to get the estimation error:

```
[residual,residualCovariance] = residual(obj,y);
```

The estimation errors (residuals) must have the following characteristics:

- Small magnitude — Small errors relative to the size of the outputs increase confidence in the estimated values.

- Zero mean

- Low autocorrelation, except at zero time lag — To compute the autocorrelation, you can use the MATLAB `xcorr` command.

Examine State Estimation Error for Simulated Data

When you simulate the output data of your nonlinear system and use that data for state estimation, you know the true state values. You can compute the errors between estimated and true state values and analyze the errors. The estimated state value at any time step is the value stored in `obj.State` after you estimate the states using the `predict` or `correct` command. The state estimation errors must satisfy the following characteristics:

- Small magnitude
- Zero mean
- Low autocorrelation, except at zero time lag

You can also compute the covariance of the state estimation error and compare it to the state estimation error covariance stored in the `StateCovariance` property of the filter. Similar values increase confidence in the performance of the filter.

See Also
extendedKalmanFilter | unscentedKalmanFilter | particleFilter | residual

More About
- "Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter" on page 16-99
- "Extended and Unscented Kalman Filter Algorithms for Online State Estimation" on page 16-27
- "Troubleshoot Online State Estimation" on page 16-42
- "Generate Code for Online State Estimation in MATLAB" on page 16-39

Validate Online State Estimation in Simulink

After you use the Extended Kalman Filter, Unscented Kalman Filter or Particle Filter blocks for online state estimation of a nonlinear system, validate the estimation before deploying the code in your application. If the validation indicates low confidence in the estimation, then see "Troubleshoot Online State Estimation" on page 16-42 for next steps. After you have validated the online estimation results, you can generate C/C++ code for the blocks using Simulink Coder software.

To validate the performance of your filter, perform state estimation using measured or simulated output data from these scenarios.

- Obtain output data from your system at different operating conditions and input values — To ensure that estimation works well under all operating conditions of interest. For example, suppose that you want to track the position and velocity of a vehicle using noisy position measurements. Measure the data at different vehicle speeds and slow and sharp maneuvers.

- For each operating condition of interest, obtain multiple sets of experimental or simulated data with different noise realizations — To ensure that different noise values do not deteriorate estimation performance.

For each of these scenarios, test the filter performance by examining the residuals and state estimation error.

Examine Residuals

The residual, or output estimation error, is the difference between the measured system output yMeasured[k], and the estimated system output yPredicted[k|k-1] at time step k. Here, yPredicted[k|k-1] is the estimated output at time step k, which is predicted using output measurements until time step k-1.

The blocks do not explicitly output yPredicted[k|k-1], however you can compute the output using the estimated state values and your state transition and measurement functions. For an example, see "Compute Residuals and State Estimation Errors" on page 16-37.

The residuals must have the following characteristics:

- Small magnitude — Small errors relative to the size of the outputs increase confidence in the estimated values.
- Zero mean
- Low autocorrelation, except at zero time lag — To compute the autocorrelation, you can use the Autocorrelation block from DSP System Toolbox™ software.

Examine State Estimation Error for Simulated Data

When you simulate the output data of your nonlinear system and use that data for state estimation, you know the true state values. You can compute the errors between estimated and true state values and analyze the errors. The estimated state value at any time step is output at the **xhat** port of the blocks. The state estimation errors must satisfy the following characteristics:

- Small magnitude
- Zero mean

- Low autocorrelation, except at zero time lag

You can also compute the covariance of the state estimation error, and compare it to the state estimation error covariance that is output by the blocks in the **P** port of the blocks. Similar values increase confidence in the performance of the filter.

Compute Residuals and State Estimation Errors

This example shows how to estimate the states of a discrete-time Van der Pol oscillator and compute state estimation errors and residuals for validating the estimation. The residuals are the output estimation errors, that is, they are the difference between the measured and estimated outputs.

In the Simulink™ model `vdpStateEstimModel`, the Van der Pol Oscillator block implements the oscillator with nonlinearity parameter, mu, equal to 1. The oscillator has two states. A noisy measurement of the first state `x1` is available.

The model uses the Unscented Kalman Filter block to estimate the states of the oscillator. Since the block requires discrete-time inputs, the Rate Transition block samples `x1` to give the discretized output measurement `yMeasured[k]` at time step `k`. The Unscented Kalman Filter block outputs the estimated state values `xhat[k|k]` at time step `k`, using `yMeasured` until time k. The filter block uses the previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`. For information about these functions, see "Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter" on page 16-99.
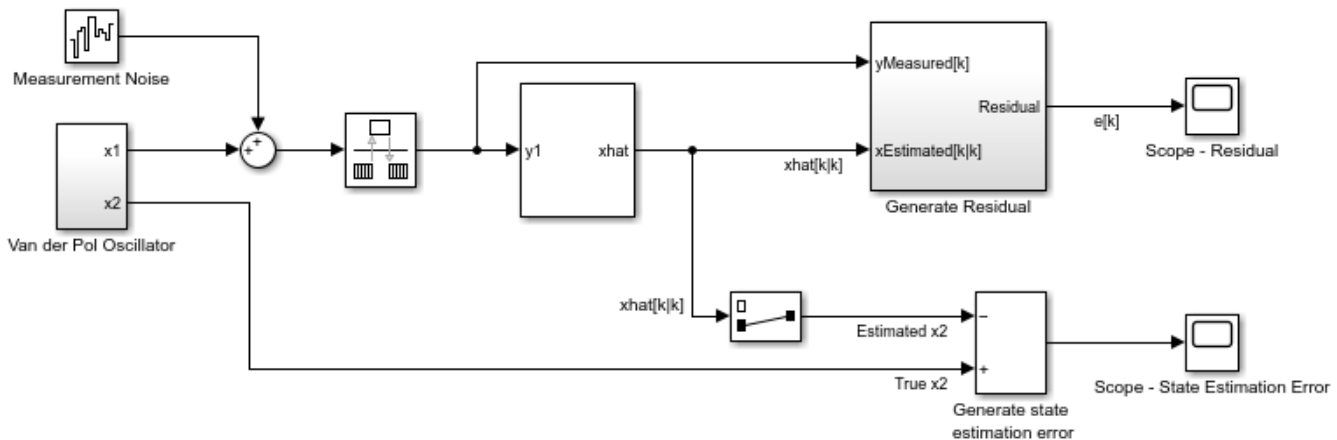
Copyright 2016-2017 The MathWorks, Inc.

To validate the state estimation, the model computes the residuals in the Generate Residual block. In addition, since the true state values are known, the model also computes the state estimation errors.

To compute the residuals, the Generate Residual block first computes the estimated output `yPredicted[k|k-1]` using the estimated states and state transition and measurement functions. Here, `yPredicted[k|k-1]` is the estimated output at time step `k`, predicted using output measurements until time step `k-1`. The block then computes the residual at time step `k` as `yMeasured[k]` - `yPredicted[k|k-1]`.

MATLAB Function - Generate Predicted Output
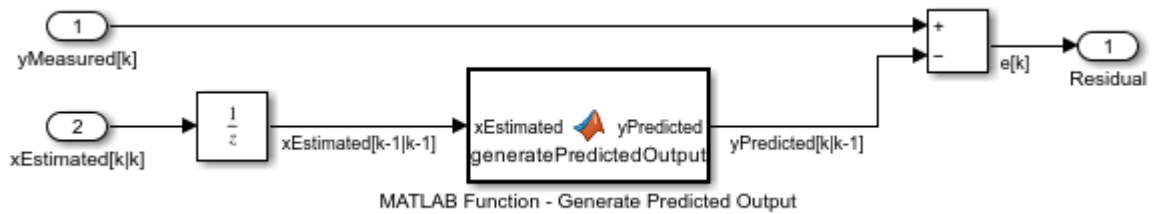
Examine the residuals and state estimation errors, and ensure that they have a small magnitude, zero mean, and low autocorrelation.

In this example, the Unscented Kalman Filter block outputs `xhat[k|k]` because the **Use the current measurements to improve state estimates** parameter of the block is selected. If you clear this parameter, the block instead outputs `xhat[k|k-1]`, the predicted state value at time step `k`, using `yMeasured` until time `k-1`. In this case, compute `yPredicted[k|k-1] = MeasurementFcn(xhat[k|k-1])`, where `MeasurementFcn` is the measurement function for your system.

See Also
Kalman Filter | Extended Kalman Filter | Unscented Kalman Filter | Particle Filter

More About
- "Extended and Unscented Kalman Filter Algorithms for Online State Estimation" on page 16-27
- "Troubleshoot Online State Estimation" on page 16-42

Generate Code for Online State Estimation in MATLAB

You can generate C/C++ code from MATLAB code that uses `extendedKalmanFilter`, `unscentedKalmanFilter` and `particleFilter` objects for online state estimation. C/C++ code is generated using the `codegen` command from MATLAB Coder software. Use the generated code to deploy online estimation algorithms to an embedded target. You can also deploy online estimation code by creating a standalone application using MATLAB Compiler software.

To generate C/C++ code for online state estimation:

1   Create a function to declare your filter object as persistent, and initialize the object. You define the object as persistent to maintain the object states between calls.

```
function [CorrectedX] = ukfcodegen(output)
% Declare object as persistent.
persistent obj;
if isempty(obj)
% Initialize the object.
obj = unscentedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,[2;0]);
obj.MeasurementNoise = 0.01;
end
% Estimate the states.
CorrectedX = correct(obj,output);
predict(obj);
end
```

The function creates an unscented Kalman filter object for online state estimation of a van der Pol oscillator with two states and one output. You use the previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`, and specify the initial state values for the two states as `[2;0]`. Here `output` is the measured output data. Save the `ukfcodegen.m` function on the MATLAB path. Alternatively, you can specify the full path name for this function.

In the `ukfcodegen.m` function, the persistent object is initialized with condition `if isempty(obj)` to ensure that the object is initialized only once, when the function is called the first time. Subsequent calls to the function only execute the `predict` and `correct` commands to update the state estimates. During initialization, you specify the nontunable properties of the object, such as `StateTransitionFcn` (specified in `ukfcodegen.m` as `vdpStateFcn.m`) and `MeasurementFcn` (specified in `ukfcodegen.m` as `vdpMeasurementFcn.m`). After that, you can specify only the tunable properties. For more information, see "Tunable and Nontunable Object Properties" on page 16-40.

In the state transition and measurement functions you must use only commands that are supported for code generation. For a list of these commands, see "Functions and Objects Supported for C/C++ Code Generation" (MATLAB Coder). Include the compilation directive `%#codegen` in these functions to indicate that you intend to generate code for the function. Adding this directive instructs the MATLAB Code Analyzer to help you diagnose and fix violations that would result in errors during code generation. For an example, type `vdpStateFcn.m` at the command line.

2   Generate C/C++ code and MEX-files using the `codegen` command from MATLAB Coder software.

```
codegen ukfcodegen -args {1}
```

The syntax `-args {1}` specifies an example of an argument to your function. The argument sets the dimensions and data types of the function argument `output` as a double-precision scalar.

Note If you want a filter with single-precision floating-point variables, you must specify the initial value of the states as single-precision during object construction.

```
obj = unscentedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,single([2;0]))
```

Then to generate code, use the following syntax.

```
codegen ukfcodegen -args {{single(1)}}
```

---

3 Use the generated code.

- Use the generated C/C++ code to deploy online state estimation to an embedded target.

- Use the generated MEX-file for testing the compiled C/C++ code in MATLAB. The generated MEX-file is also useful for accelerating simulations of state estimation algorithms in MATLAB.

  Load the estimation data. Suppose that your output data is stored in the `measured_data.mat` file.

  ```
  load measured_data.mat output
  ```

  Estimate the states by calling the generated MEX-file.

  ```
  for i = 1:numel(output)
      XCorrected = ukfcodegen_mex(output(i));
  end
  ```

This example generates C/C++ code for compiling a MEX-file. To generate code for other targets, see `codegen` in the MATLAB Coder documentation.

Tunable and Nontunable Object Properties

Property Type	Extended Kalman Filter Object	Unscented Kalman Filter Object	Particle Filter Object
Tunable properties that you can specify multiple times either during object construction, or afterward using dot notation	`State`, `StateCovariance`, `ProcessNoise`, and `MeasurementNoise`	`State`, `StateCovariance`, `ProcessNoise`, `MeasurementNoise`, `Alpha`, `Beta`, and `Kappa`	`Particles` and `Weights`
Nontunable properties that you can specify only once, either during object construction, or afterward using dot notation, but before using the `predict` or `correct` commands	`StateTransitionFcn`, `MeasurementFcn`, `StateTransitionJacobianFcn`, and `MeasurementJacobianFcn`	`StateTransitionFcn` and `MeasurementFcn`	`StateTransitionFcn`, `MeasurementLikelihoodFcn`, `StateEstimationMethod`, `StateOrientation`, `ResamplingPolicy` and `ResamplingMethod`

Property Type	Extended Kalman Filter Object	Unscented Kalman Filter Object	Particle Filter Object
Nontunable properties that you must specify during object construction	HasAdditiveProcessNoise and HasAdditiveMeasurementNoise	HasAdditiveProcessNoise and HasAdditiveMeasurementNoise	

See Also
extendedKalmanFilter | particleFilter | unscentedKalmanFilter

More About

Troubleshoot Online State Estimation

After you perform state estimation of a nonlinear system using linear, extended, or unscented Kalman filter or particle filter algorithms, you validate the estimation before deploying the code in your application. If the validation indicates low confidence in the estimation, check the following filter properties that you specified:

- Initial state and state covariance values — If you find that the measured and estimated outputs of your system are diverging at the beginning of state estimation, check the initial values that you specified.

- State transition and measurement functions — Verify that the functions you specify are a good representation of the nonlinear system. If the true system is continuous-time, to implement the algorithms, you discretize the state transition and measurement equations and use the discretized versions. If the state estimation results are not satisfactory, consider decreasing the sample time used for discretization. Alternatively, try a different discretization method. For an example of how to discretize a continuous-time state transition function, type `edit vdpStateFcn.m` at the command line. Also see, "Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter" on page 16-99.

- Process and measurement noise covariance values — If the difference between estimated and measured outputs of your system is large, try specifying different values for the process and measurement noise covariance values.

- Choice of algorithm — If you are using the extended Kalman filter algorithm, you can try the unscented Kalman filter, or the particle filter algorithm instead. The unscented Kalman filter and particle filter may capture the nonlinearities in the system better.

To troubleshoot state estimation, you can create multiple versions of the filter with different properties, perform state estimation, and choose the filter that gives the best validation results.

At the command line, if you want to copy an existing filter object and then modify properties of the copied object, use the `clone` command. Do not create additional objects using syntax `obj2 = obj`. Any changes made to the properties of the new object created in this way (`obj2`) also change the properties of the original object (`obj`).

See Also

Functions
extendedKalmanFilter | unscentedKalmanFilter | particleFilter

Blocks
Extended Kalman Filter | Unscented Kalman Filter | Particle Filter

More About

- "Validate Online State Estimation at the Command Line" on page 16-34
- "Validate Online State Estimation in Simulink" on page 16-36
- "Generate Code for Online State Estimation in MATLAB" on page 16-39

Line Fitting with Online Recursive Least Squares Estimation

This example shows how to perform online parameter estimation for line-fitting using recursive estimation algorithms at the MATLAB® command line. You capture the time-varying input-output behavior of the hydraulic valve of a continuously variable transmission.

Physical System

The system is a continuously variable transmission (CVT) driven by a hydraulic valve, inspired by reference [1]. The valve pressure is connected to the CVT which allows it to change its speed ratio and to transmit torque from the engine to the wheels. The input-output behavior of the valve can be approximated by:

$$y(t) = k(t)u(t) + b(t) \ \text{ for } \ \frac{-b(t)}{k(t)} < u \leq 1$$

Here, t is the current time, y(t) is the valve pressure in bar, u(t) is the unitless input in the range of [0, 1]. The condition $\frac{-b}{k} < u$ is the dead-band of the valve.

The slope, k(t), and offset, b(t), depend on the system temperature. They vary as the system warms up from cold start to typical operating temperature. You want to estimate k(t) and b(t) based on noisy measurements of u(t) and y(t).

Data

The true slope and offset parameters are k(0)=70 and b(0)=-15 at time t=0s. At t=50s the engine starts. The parameters vary over time until they reach k(950)=50 and b(950)=-13 at t=950s. The sample time is Ts=0.1s.

The content of the input signal u is critical for parameter estimation. Consider a case where u, and hence y, is constant. Then there are infinitely many k and b values that satisfy y = k u + b. u(t) must be persistently exciting the system for successful estimation of k(t) and b(t). In this example, the input u:

- is zero from t=0s until t=50s.
- has step changes to 0.40, 0.45, 0.50, 0.55, 0.60, 0.55, 0.50, 0.45, 0.40 every 100s, from t=50s until t=950s.
- a Gaussian random variable with zero mean, 0.02 standard deviation was added at each time step from t=50s until t=950s to provide extra excitation of system dynamics for identification purposes.

The output is generated with the aforementioned true values of k(t), b(t) along with the input signal u(t), using y(t) = k(t) u(t) + b(t) + e(t). e(t), measurement noise, is a Gaussian random variable with zero mean and standard deviation 0.05.

```
load LineFittingRLSExample u y k b t;
figure();
subplot(2,1,1);
plot(t,u);
ylabel('Input signal, u, [unitless]');
subplot(2,1,2);
plot(t,y);
```

```
ylabel('Valve pressure, y, [bar]');
xlabel('Time [s]');
```
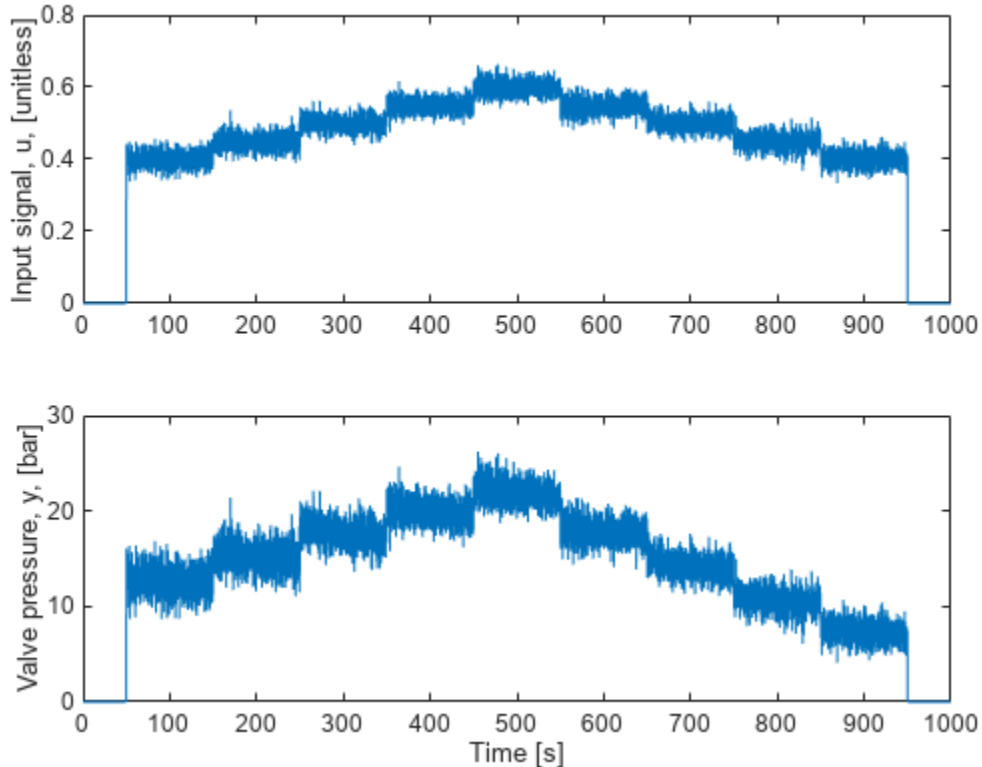

Online Parameter Estimation Using Recursive Least Squares

Write the valve input-output model using vector notation:

$$y(t) = k(t)u(t) + b(t) + e(t)$$
$$= [u(t)\ 1][k(t)\ b(t)]^T + e(t)$$
$$= H(t)x(t) + e(t)$$

where $H(t) = [u(t)\ 1]$ is the regressors and $x = [k(t)\ b(t)]^T$ is the parameters to be estimated. $e(t)$ is the unknown noise. You use the recursiveLS estimation command to create a System object for online parameter estimation. You then use the step command to update the parameter estimates, x(t), at each time-step based on H(t) and y(t).

You specify the following recursiveLS System Object properties:

- **Number of parameters:** 2.
- **EstimationMethod:** 'ForgettingFactor' (default). This method has only one scalar parameter, ForgettingFactor, which requires limited prior information regarding parameter values.
- **ForgettingFactor:** 0.95. The parameters are expected to vary over time, hence less than 1.
  $\dfrac{1}{1-\lambda} = 20$ is the number of past data samples that influence the estimates most.

- **InitialParameters:** [70; -15], an initial guess for the parameter values. Optional, but recommended for reducing initial transients.

- **InitialParameterCovariance:** Your estimate of uncertainty in the initial parameter guess. Set it to a small value, 1% of the absolute value of the initial parameters, if you have confidence in the initial parameter guesses. Optional but recommended, especially when you specify InitialParameters. This is only utilized with the ForgettingFactor and KalmanFilter estimation methods.

```
X = recursiveLS(2,... % 2=number of estimated parameters
    'EstimationMethod','ForgettingFactor',...
    'ForgettingFactor',0.95,...
    'InitialParameters',[70; -15],...
    'InitialParameterCovariance',[0.7 0.15]);
```

This example simulates the online operation of the estimator by providing one (y(t),H(t)) pair to the estimator at a time. Call the step command to update parameters with each new pair. The parameter adaptation is enabled only when the input u is outside the dead band (u>0.3).

```
theta = zeros(numel(u),2);
yHat = zeros(numel(u),1);
PHat = zeros(numel(u),2,2);
for kk=1:numel(u)
    % enable parameter estimation only when u is outside the dead-band
    if u(kk)>=0.3
        X.EnableAdaptation = true();
    else
        X.EnableAdaptation = false();
    end
    [theta(kk,:),yHat(kk)] = step(X,y(kk),[u(kk) 1]); % get estimated parameters and output
    PHat(kk,:,:) = X.ParameterCovariance; % get estimated uncertainty in parameters
    % perform any desired tasks with the parameters
end
```

Estimated Parameters

The true and estimated parameter values are:

```
figure();
subplot(2,1,1);
plot(t,theta(:,1),t,k); % Estimated and real slope, respectively
ylabel('Slope');
xlabel('Time');
ylim([49 71]);
legend('Estimated','Real','Location','Best');
subplot(2,1,2);
plot(t,theta(:,2),t,b); % Estimated and real offset, respectively
ylabel('Offset');
xlabel('Time');
ylim([-15.25 -12.75]);
```
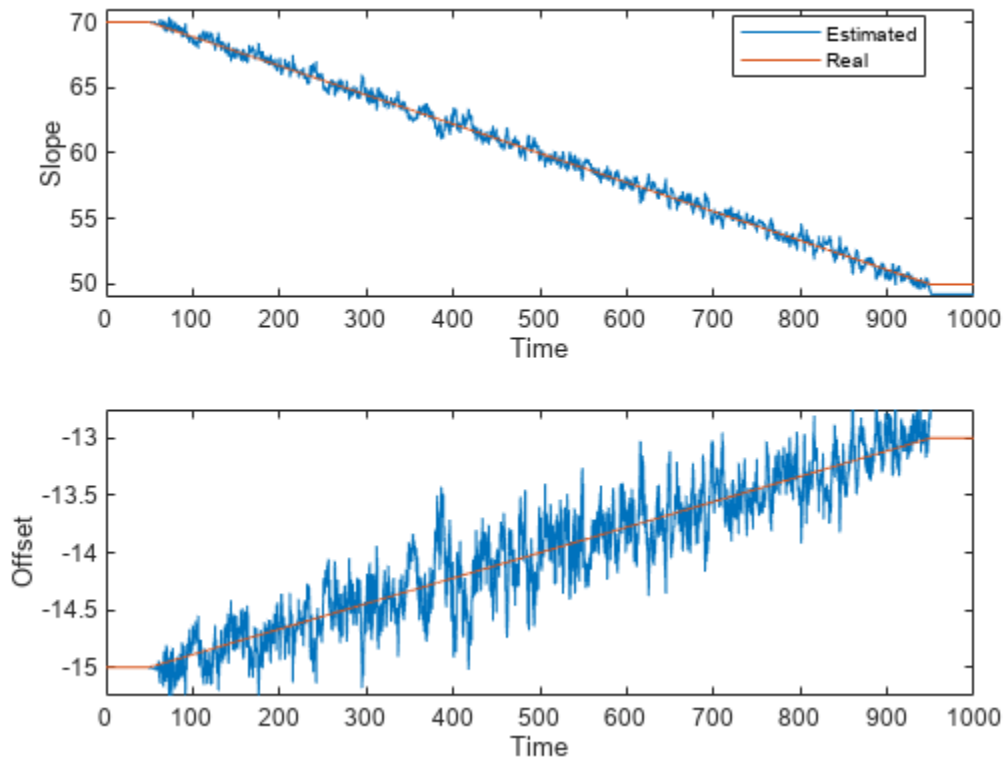
Validating the Estimated Model

The estimator provides the following two tools to judge the quality of the parameter estimates:

1 **Output estimate $\hat{y}(t)$:** The second output argument of the step method is $\hat{y}(t) = H(t)\hat{x}(t)$. The relative and absolute error between y and $\hat{y}$ are measures of the goodness of the fit.

2 **Parameter covariance estimate $\widehat{P}(t)$:** This is available with the ForgettingFactor and KalmanFilter algorithms. It is stored in the ParameterCovarianceMatrix property of the estimator. The diagonals of $\widehat{P}$ are the estimated variances of the parameters. The lower the better.

The output measurement and its estimate, as well as the associated absolute and relative errors when the engine is on are:

```
engineOn = t>50 & t<950;
figure();
subplot(2,1,1);
absoluteError = y-yHat;
plot(t(engineOn),absoluteError(engineOn));
ylim([-0.15 0.15]);
ylabel('Abs. Error [bar]');
subplot(2,1,2);
relativeError = (y-yHat)./y;
plot(t(engineOn),relativeError(engineOn));
ylim([-0.025 0.025]);
```

```
ylabel('Rel. Error [unitless]');
xlabel('Time [s]');
```
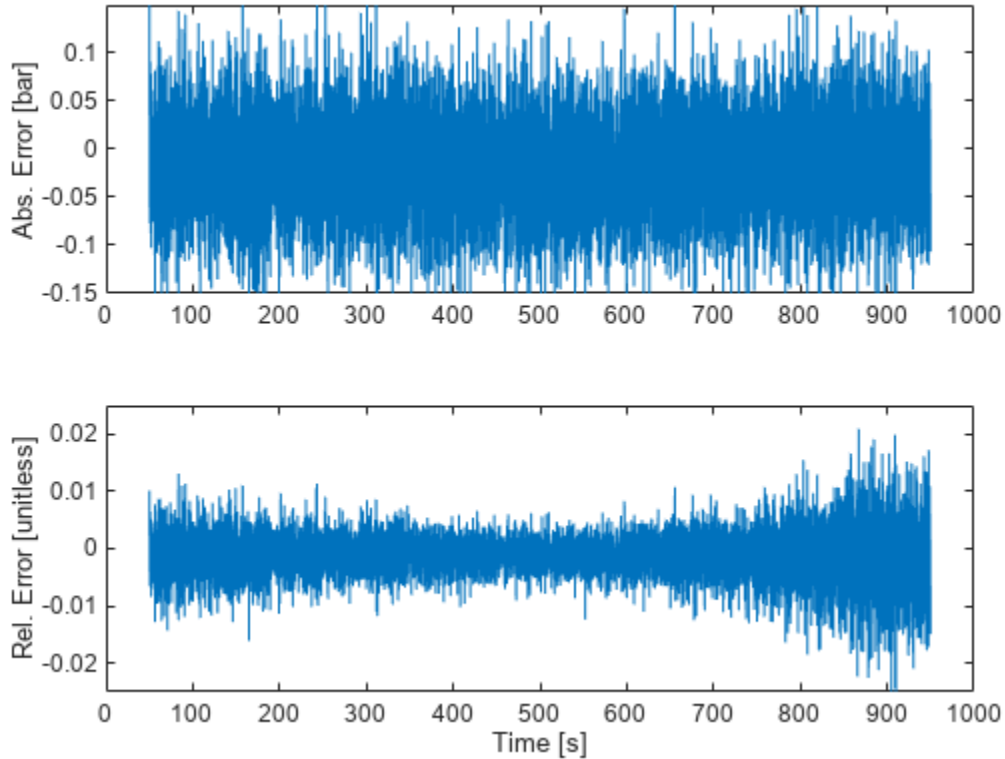

The absolute errors are about 0.1bar. The relative errors are below 2%. Both quantities are small.

The diagonals of the parameter covariance matrix, scaled by the variance of the residuals $y(t) - \hat{y}(t)$, capture the variances of parameter estimates. The square-root of the variances are the standard deviations of the parameter estimates.

```
noiseVariance = var(y(engineOn)-yHat(engineOn));
figure();
subplot(2,1,1);
hold on;
plot(t,sqrt(PHat(:,1,1)*noiseVariance));
ylim([0 1]);
ylabel('Std. dev. of slope k');
subplot(2,1,2);
plot(t,sqrt(PHat(:,2,2)*noiseVariance));
ylim([0 1]);
ylabel('Std. dev. of offset b');
xlabel('Time [s]');
hold on;
```
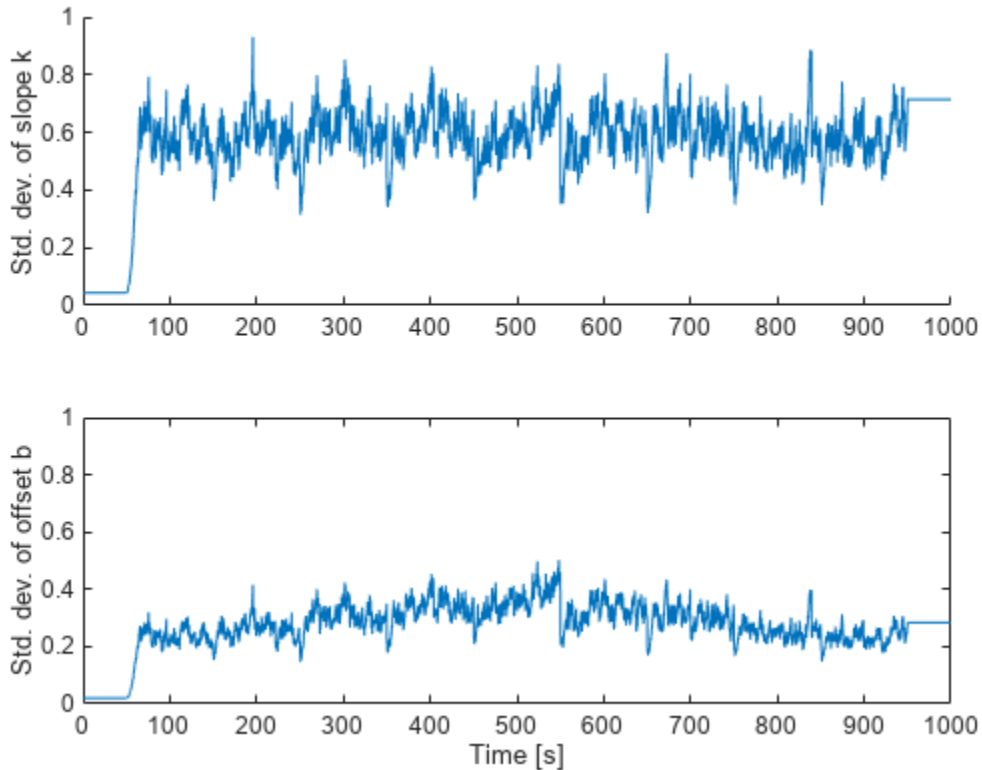
The standard deviation of the slope k fluctuates around 0.7. This is small relative to the range of values of k [50, 70]. This gives confidence in the parameter estimates. The situation is similar with the offset b, which is in the range [-15 -13].

Note that the parameter standard deviations are also estimates. They are based on the assumption that the residuals $y(t) - \hat{y}(t)$ are white. This depends on the estimation method, its associated parameters, structure of the estimated model, and the input signal u. Differences between the assumed and the actual model structure, lack of persistent input excitation, or unrealistic estimation method settings can lead to overly optimistic or pessimistic uncertainty estimates.

Summary

You performed a line fit using recursive least squares to capture the time-varying input-output behavior of a hydraulic valve. You evaluated the quality of fit by looking at two signals: the error between estimated and measured system output, and the parameter covariance estimates.

References

[1] Gauthier, Jean-Philippe, and Philippe Micheau. "Regularized RLS and DHOBE: An Adaptive Feedforward for a Solenoid Valve." Control Systems Technology, IEEE Transactions on 20.5 (2012): 1311-1318

See Also

recursiveAR | recursiveARMA | recursiveARX | recursiveARMAX | recursiveOE | recursiveBJ | recursiveLS | step | reset | release | isLocked | clone

Related Examples

- "Perform Online Parameter Estimation at the Command Line" on page 16-21
- "Validate Online Parameter Estimation at the Command Line" on page 16-10
- "Generate Code for Online Parameter Estimation in MATLAB" on page 16-24
- "Online ARX Parameter Estimation for Tracking Time-Varying System Dynamics" on page 16-50

Online ARX Parameter Estimation for Tracking Time-Varying System Dynamics

This example shows how to perform online parameter estimation for a time-varying ARX model at the MATLAB® command line. The model parameters are updated at each time step with incoming new data. This model captures the time-varying dynamics of a linear plant.

Plant

The plant can be represented as:

$$y(s) = G(s)u(s) + e(s)$$

Here, G is the transfer function and e is the white-noise. The plant has two operating modes. In the first operating mode, the transfer function is:

$$G1(s) = \frac{4500}{(s + 5)(s^2 + 1.2s + 900)}$$

The lightly damped poles in G1(s) have the damping 0.02 and natural frequency 30rad/s. In the second operating mode, the natural frequency of these poles is 60rad/s. In this mode, the transfer function is:

$$G2(s) = \frac{18000}{(s + 5)(s^2 + 2.4s + 3600)}$$

The plant operates in the first mode until t=10s, and then switches to the second mode.

The Bode plots of G1 and G2 are:

```
wn = 30; % natural frequency of the lightly damped poles
ksi = 0.02; % damping of the poles
G1 = tf(1,conv([1/5 1],[1/wn^2 2*ksi/wn 1])); % plant in mode 1
wn = 60; % natural frequency in the second operating mode
G2 = tf(1,conv([1/5 1],[1/wn^2 2*ksi/wn 1])); % plant in mode 2
bode(G1,G2,{1,125});
legend('G1','G2','Location','Best');
```
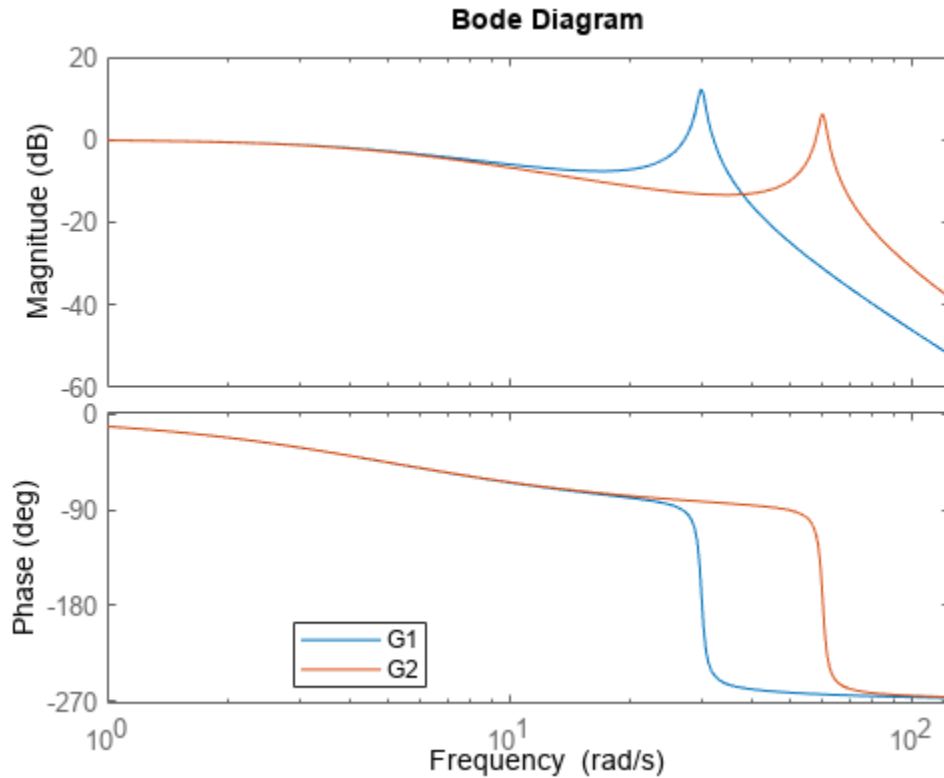
Online ARX Parameter Estimation

The aim is to estimate the dynamics of the plant during its operation. ARX is a common model structure used for this purpose. ARX models have the form:

$$y(t) = \frac{B(q)}{A(q)}u(t) + \frac{1}{A(q)}e(t)$$

Here, q^{-1} is the time-shift operator. The ratio of the polynomials $B(q)/A(q)$ captures the input-output model ($u(t)$ to $y(t)$), and $1/A(q)$ captures the noise model ($e(t)$ to $y(t)$). You are estimating the coefficients of the $A(q)$ and $B(q)$ polynomials. $e(t)$ is white noise.

ARX model structure is a good first candidate for estimating linear models. The related estimation methods have low computational burden, are numerically robust, and have the convexity property. The convexity property ensures there is no risk of parameter estimation getting stuck at a local optima. However, ARX model structure does not provide flexibility for noise models.

The lack of flexibility in noise modeling can pose difficulties if the structure of the plant does not match with the ARX model structure, or if the noise is not white. Two approaches to remedy this issue are:

1   Data Filtering: If the noise model is not important for your application, you can use data filtering techniques. For more details, see the 'Filter the Data' section.

2   Different model structures: Use ARMAX, Output-Error, and Box-Jenkins polynomial models for more flexibility in model structures.

Select Sample Time

The sample time choice is important for good approximation of the continuous-time plant by a discrete-time model. A rule of thumb is to choose the sampling frequency as 20 times the dominant dynamics of the system. The plant has the fastest dominant dynamics at 60rad/s, or about 10Hz. The sampling frequency is therefore 200Hz.

```
Ts = 0.005; % [s], Sample time, Ts=1/200
```

System Excitation

For successful estimation the plant inputs must persistently excite its dynamics. Simple inputs such as a single step input are typically not sufficient. In this example the plant is driven by a pulse with amplitude 10 and a period of 1 second. The pulse width is 50% of its period.

Generate plant input and output signals:

```
t = 0:Ts:20; % Time vector
u = double(rem(t/1,1)-0.5 < 0); % pulse
y = zeros(size(u));
% Store random number generator's states for reproducible results.
sRNG = rng;
rng('default');
% Simulate the mode-switching plant with a zero-order hold.
G1d = c2d(G1,Ts,'zoh');
B1 = G1d.num{1}.';
A1 = G1d.den{1}.'; % B1 and A1 corresponds to G1.
G2d = c2d(G2,Ts,'zoh');
B2 = G2d.num{1}.';
A2 = G2d.den{1}.'; % B2 and A2 corresponds to G2.
idx = numel(B2):-1:1;
for ct=(1+numel(B2)):numel(t)
    idx = idx + 1;
    if t(ct)<10 % switch mode after t=10s
        y(ct) = u(idx)*B1-y(idx(2:end))*A1(2:end);
    else
        y(ct) = u(idx)*B2-y(idx(2:end))*A2(2:end);
    end
end
% Add measurement noise
y = y + 0.02*randn(size(y));
% Restore the random number generator's states.
rng(sRNG);
```

Plot the input-output data:

```
figure();
subplot(2,1,1);
plot(t,u);
ylabel('Input (u)');
subplot(2,1,2);
plot(t,y);
ylim([-0.2 1.2])
ylabel('Output (y)');
xlabel('Time [s]');
```
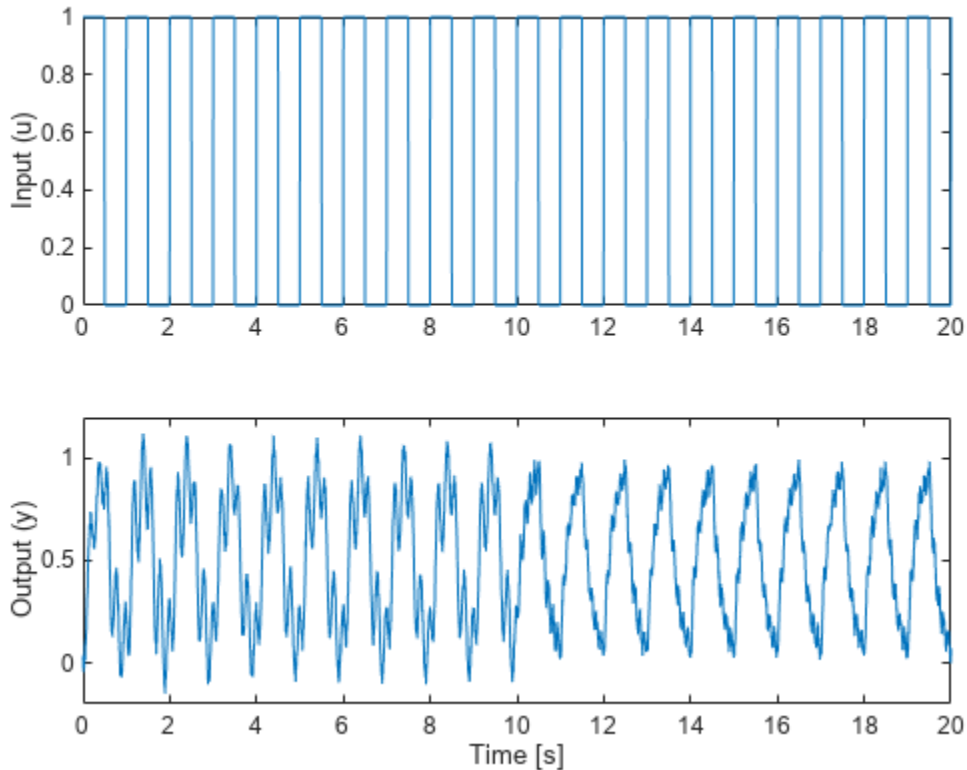
Filter the Data

The plant has the form:

$$y(t) = G(q)u(t) + e(t)$$

where e(t) is the white noise. In contrast, the ARX models have the form

$$y(t) = \frac{B(q)}{A(q)}u(t) + \frac{1}{A(q)}e(t)$$

The estimator will use B(q) and A(q) to approximate G(q). However, note the difference in the noise models. The plant has white-noise e(t) directly impacting y(t), but the ARX model assumes that a white noise term filtered by 1/A(q) impacts y(t). This mismatch will negatively affect the estimation.

When the noise model is not of interest, one method to reduce the impact of this mismatch is to use a data filter. Use a filter $F(q)$ on both u(t) and y(t) to obtain $u_f(t) = F(q)u(t)$ and $y_f(t) = F(q)y(t)$. Then use the filtered signals $u_f(t)$ and $y_f(t)$ in the estimator instead of the plant input u(t) and output y(t). The choice of data filter lets you reduce the influence of e(t) on the estimation.

The data filter F(q) is typically a low-pass or a band-pass filter based on the frequency range of importance for the application, and the characteristics of e(t). Here, a 4th order Butterworth low-pass filter with cutoff frequency 10Hz is used. This is approximately the frequency of the fastest dominant dynamics in the plant (60rad/s). A low-pass filter is sufficient here because the noise term does not have low-frequency content.

```
% Filter coefficients
Fa = [1 -3.1806 3.8612 -2.1122 0.4383]; % denominator
Fb = [4.1660e-04 1.6664e-03 2.4996e-03 1.6664e-03 4.1660e-04]; % numerator
% Filter the plant input for estimation
uf = filter(Fb,Fa,u);
% Filter the plant output
yf = filter(Fb,Fa,y);
```

Set Up the Estimation

Use the recursiveARX command for online parameter estimation. The command creates a System object™ for online parameter estimation of an ARX model structure. Specify the following properties of the object:

- **Model orders:** [3 1 0]. na = 3 because the plant has 3 poles. nk = 0 because plant does not have input delay. nb = 1 corresponds to no zeros in the system. nb was set after a few iterations, starting from nb=4 which corresponds to three zeros, and hence a proper model. A smaller number of estimated parameters are desirable and nb=1 yields sufficient results.

- **EstimationMethod:** 'ForgettingFactor' (default). This method has only one scalar parameter, ForgettingFactor, which requires limited prior information regarding parameter values.

- **ForgettingFactor:** 0.995. The forgetting factor, λ, is less than one as the parameters vary over time. $\frac{1}{1-\lambda}$ = 200 is the number of past data samples that influence the estimates most.

```
X = recursiveARX([3 1 0]); % [na nb nk]
X.ForgettingFactor = 0.995;
```

Create arrays to store estimation results. These are useful for validating the algorithms.

```
np = size(X.InitialParameterCovariance,1);
PHat = zeros(numel(u),np,np);
A = zeros(numel(u),numel(X.InitialA));
B = zeros(numel(u),numel(X.InitialB));
yHat = zeros(1,numel(u));
```

Use the step command to update the parameter values using one set of input-output data at each time step. This illustrates the online operation of the estimator.

```
for ct=1:numel(t)
    % Use the filtered output and input signals in the estimator
    [A(ct,:),B(ct,:),yHat(ct)] = step(X,yf(ct),uf(ct));
    PHat(ct,:,:) = X.ParameterCovariance;
end
```

View the Bode plot of the estimated transfer functions:

```
G1Hat = idpoly(A(1000,:),B(1000,:),1,1,1,[],Ts); % Model snapshot at t=10s
G2Hat = idpoly(X); % Snapshot of the latest model, at t=20s
G2Hat.Ts = G1d.Ts; % Set the sample time of the snapshot
figure();
bode(G1,G1Hat);
xlim([0.5 120]);
legend('G1','Identified model at t=10s','Location','Best');
```
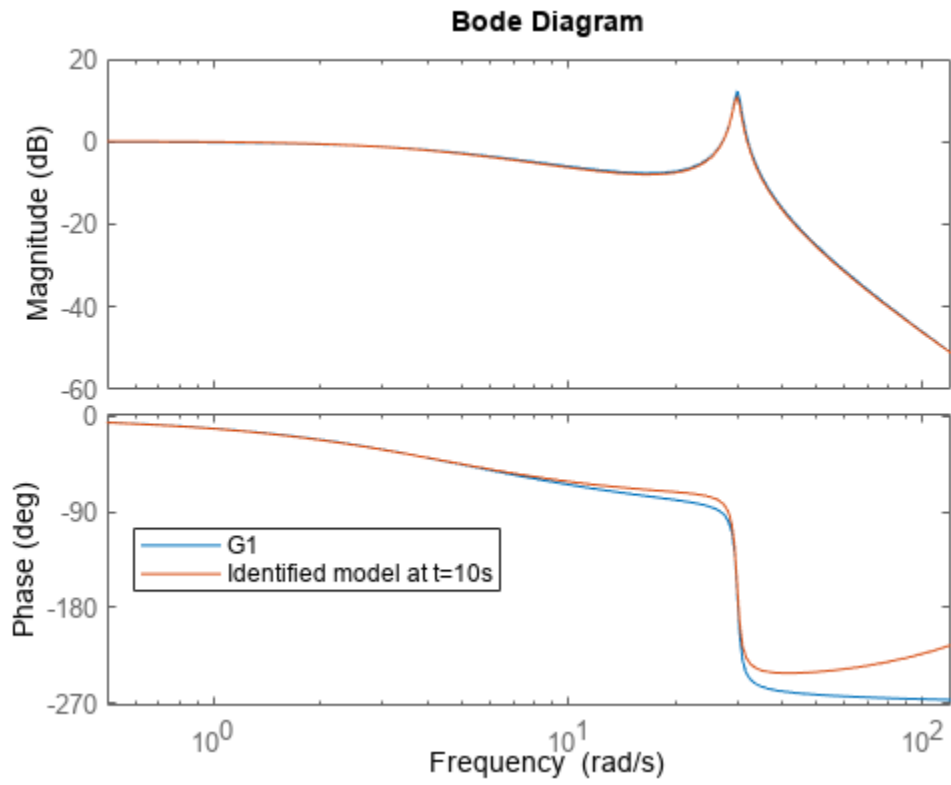
```
figure();
bode(G2,G2Hat);
xlim([0.5 120]);
legend('G2','Identified model at t=20s','Location','Best');
```

Bode Diagram
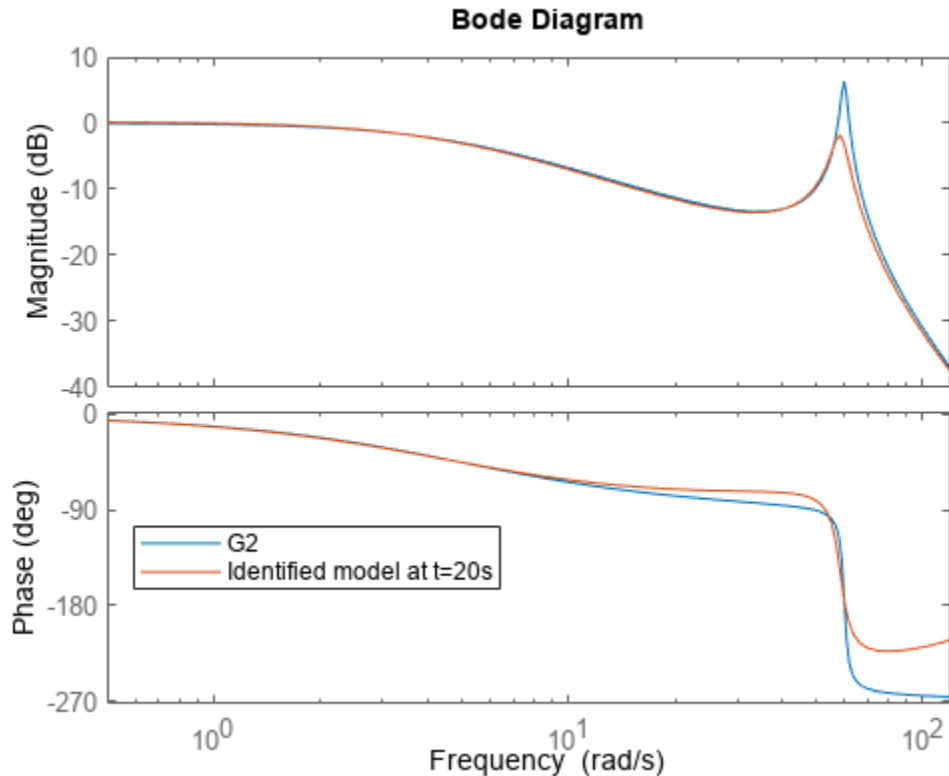
Validating the Estimated Model

Use the following techniques to validate the parameter estimation:

1   **View output estimate, yhat(t):** The third output argument of the step method is the one-step ahead prediction of the output yhat(t). This is based on current model parameters as well as current and past input-output measurements. The relative and absolute error between y(t) and yhat(t) are measures of the goodness of the fit.

2   **View the parameter covariance estimate, Phat(t):** This is available with the ForgettingFactor and KalmanFilter estimation methods. It is stored in the ParameterCovarianceMatrix property of the estimator. The diagonals of Phat(t) are the estimated variances of the parameters. It should be bounded, and the lower the better.

3   **Simulate the estimated time-varying model:** Use u(t) and the estimated parameters to simulate the model to obtain a simulated output, ysim(t). Then compare y(t) and ysim(t). This is a more strict validation than comparing y(t) and yhat(t) because ysim(t) is generated without the plant output measurements.

The absolute error yf(t)-yhat(t) and the relative error (yf(t)-yhat(t))/yf(t) are:

```
figure();
subplot(2,1,1);
plot(t,yf-yHat);
ylabel('Abs. Error');
subplot(2,1,2);
plot(t,(yf-yHat)./yf);
ylim([-0.05 0.05]);
```

```
ylabel('Rel. Error');
xlabel('Time [s]');
```
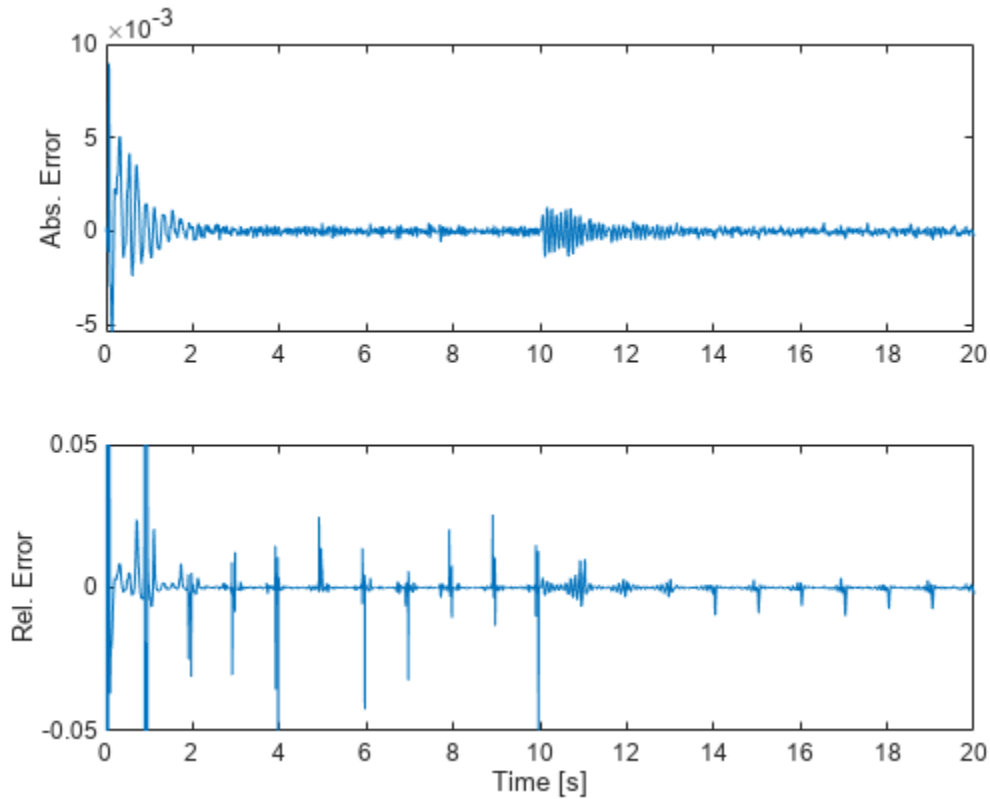

The absolute errors are on the order of 1e-3, which is small compared to the measured output signal itself. The relative error plot at the bottom confirms this, with errors being less than 5% except at the beginning of the simulation.

The diagonals of the parameter covariance matrix, scaled by the variance of the residuals y(t)-yhat(t), capture the variances of parameter estimates. The square-root of the variances are the standard deviations of the parameter estimates. The first three elements on the diagonals are the three parameters estimated in the A(q) polynomial. The last element is the single parameter in the B(q) polynomial. Let's look at the first estimated parameter in A(q)

```
noiseVariance = var(yf-yHat);
X.A(2) % The first estimated parameter. X.A(1) is fixed to 1
```

```
ans = -2.8635
```

```
sqrt(X.ParameterCovariance(1,1)*noiseVariance)
```

```
ans = 0.0175
```

The standard deviation 0.0175 is small relative to the absolute value of the parameter value 2.86. This indicates good confidence in the estimated parameter.

```
figure();
plot(t,sqrt(PHat(:,1,1)*noiseVariance));
```

```
ylabel('Standard deviation estimate for the parameter A(2)')
xlabel('Time [s]');
```
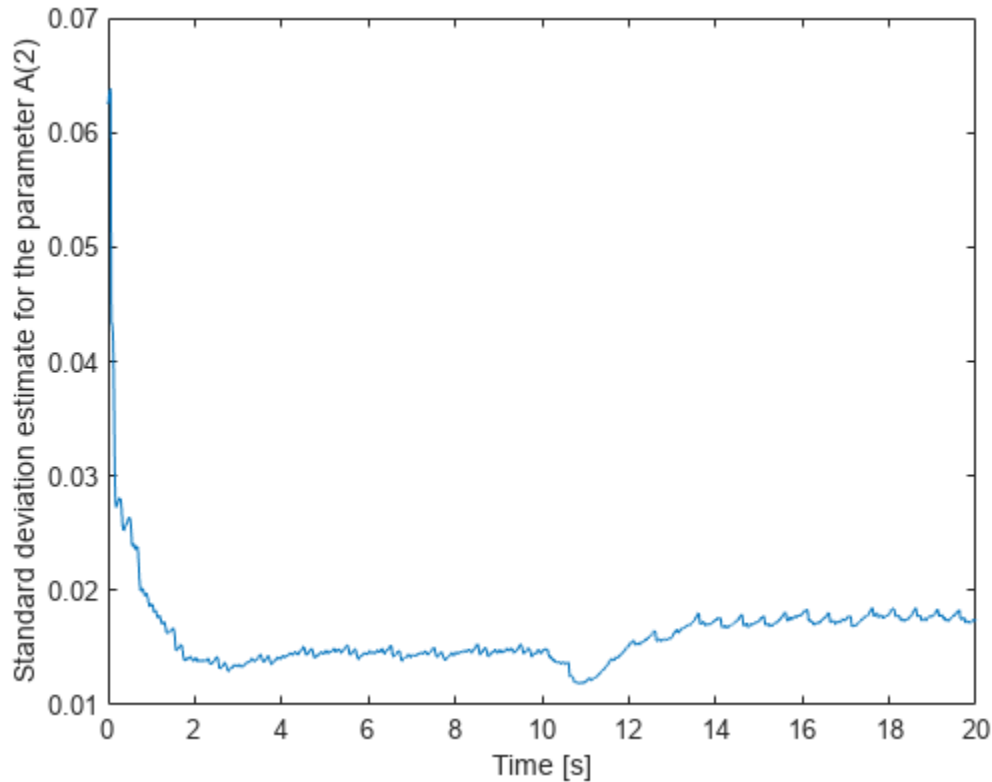

The uncertainty is small and bounded throughout the estimation. However, note that the parameter standard deviations are also estimates. They are based on the assumption that the residuals y(t)-yhat(t) are white. This depends on the estimation method, its associated parameters, the structure of the estimated model, and the input signal u. Differences between the assumed and the actual model structure, lack of persistent input excitation, or unrealistic estimation method settings can lead to overly optimistic or pessimistic uncertainty estimates.

Lastly, simulate the estimated ARX model using the stored history of estimated parameters. This simulation can also be done simultaneously with the estimation loop for validation during online operation.

```
ysim = zeros(size(y));
idx = numel(B2):-1:1;
for ct=(1+numel(B2)):numel(t)
    idx = idx + 1;
    ysim(ct) = u(idx(1))*B(idx(1),:)-ysim(idx(2:end))*A(ct,2:end)';
end
figure();
subplot(2,1,1);
plot(t,y,t,ysim);
ylabel('System Output');
legend('Measured','Estimated','Location','Best');
subplot(2,1,2);
plot(t,y-ysim);
```

```
ylim([-0.5 0.5]);
ylabel('Error, y(t)-ysim(t)');
xlabel('Time [s]');
```
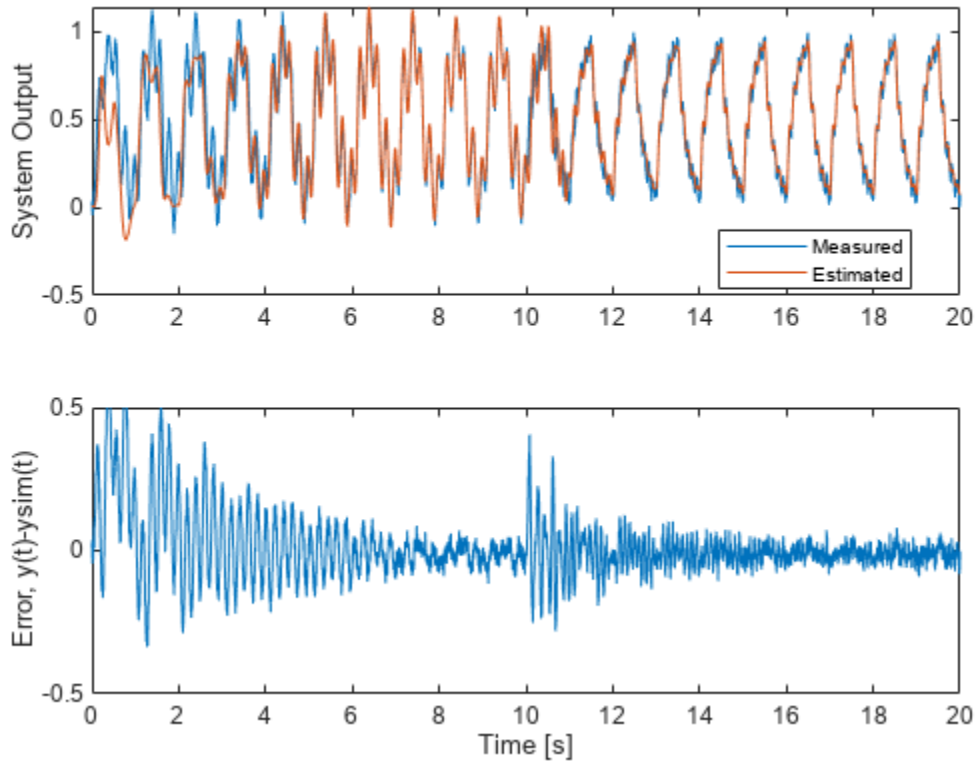

The error is large initially, but it settles to a smaller value around t=5s for the first operating mode. The large initial error can be reduced by providing the estimator an initial guess for the model parameters and initial parameter covariance. When the plant switches to the second mode, the errors grow initially but settle down as time goes on as well. This gives confidence that the estimated model parameters are good at capturing the model behavior for the given input signal.

Summary

You performed online parameter estimation for an ARX model. This model captured the dynamics of a mode-switching plant. You validated the estimated model by looking at the error between estimated, simulated, measured system outputs as well as the parameter covariance estimates.

See Also

recursiveAR | recursiveARMA | recursiveARX | recursiveARMAX | recursiveOE | recursiveBJ | recursiveLS | step | reset | release | isLocked | clone

Related Examples

- "Perform Online Parameter Estimation at the Command Line" on page 16-21
- "Validate Online Parameter Estimation at the Command Line" on page 16-10

Online Recursive Least Squares Estimation

This example shows how to implement an online recursive least squares estimator. You estimate a nonlinear model of an internal combustion engine and use recursive least squares to detect changes in engine inertia.

Engine Model

The engine model includes nonlinear elements for the throttle and manifold system, and the combustion system. The model input is the throttle angle and the model output is the engine speed in rpm.

```
open_system('iddemo_engine');
sim('iddemo_engine')
```


Copyright 2013-2015 The MathWorks, Inc.
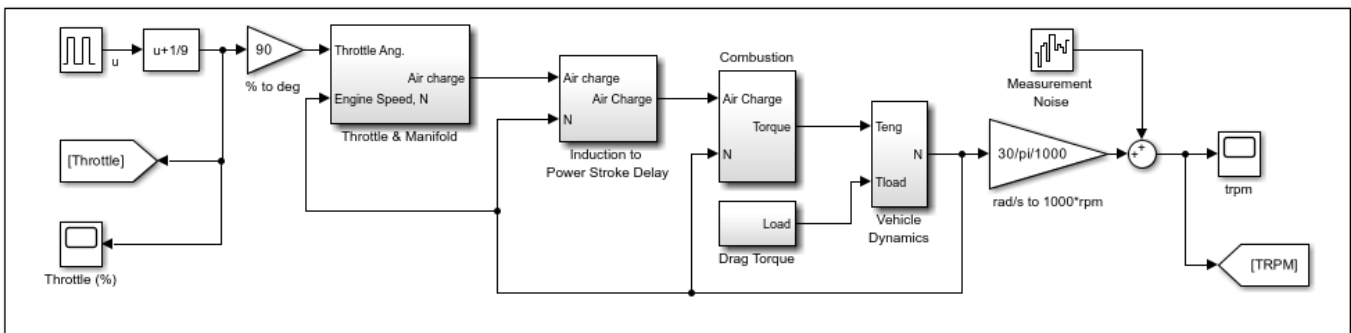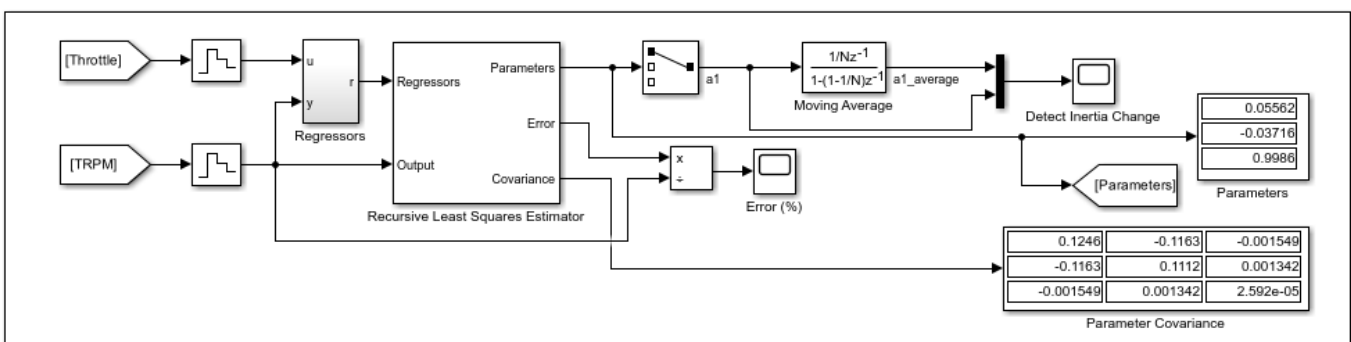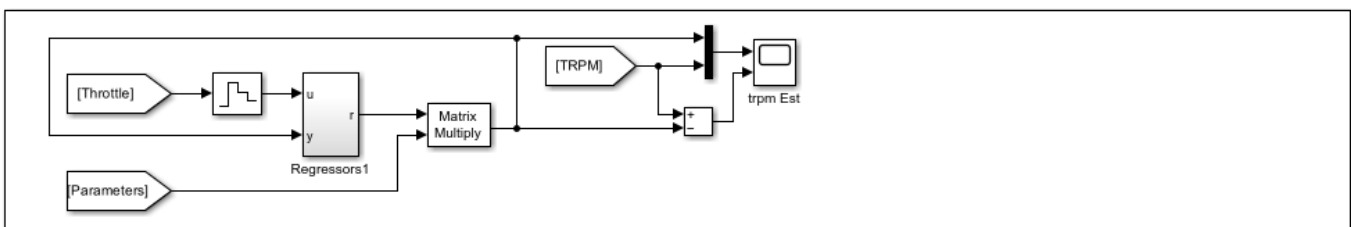
The engine model is set up with a pulse train driving the throttle angle from open to closed. The engine response is nonlinear, specifically the engine rpm response time when the throttle is open and closed are different.

At 100 seconds into the simulation an engine fault occurs causing the engine inertia to increase (the engine inertia, J, is modeled in the `iddemo_engine/Vehicle Dynamics` block). The inertia change causes engine response times at open and closed throttle positions to increase. You use online recursive least squares to detect the inertia change.

```
open_system('iddemo_engine/trpm')
```
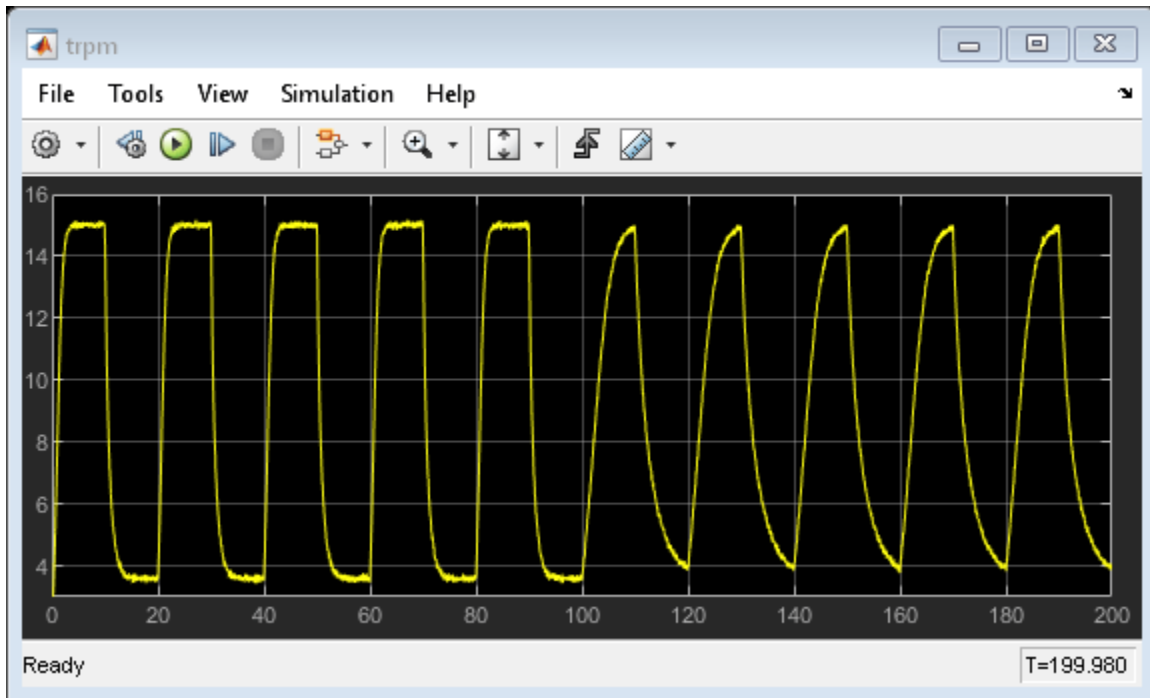

Estimation Model

The engine model is a damped second order system with input and output nonlinearities to account for different response times at different throttle positions. Use the recursive least squares block to identify the following discrete system that models the engine:

$$y_n = a_1 u_{n-1} + a_2 u_{n-1}^2 + a_3 y_{n-1}$$

Since the estimation model does not explicitly include inertia we expect the a values to change as the inertia changes. We use the changing a values to detect the inertia change.

The engine has significant bandwidth up to 16Hz. Set the estimator sampling frequency to 2*160Hz or a sample time of $T_s = 0.003$ seconds.

Recursive Least Squares Estimator Block Setup

The $u_{n-1}, u_{n-1}^2, y_{n-1}$ terms in the estimated model are the *model regressors* and inputs to the recursive least squares block that estimates the a values. You can implement the regressors as shown in the `iddemo_engine/Regressors` block.

```
open_system('iddemo_engine/Regressors');
```
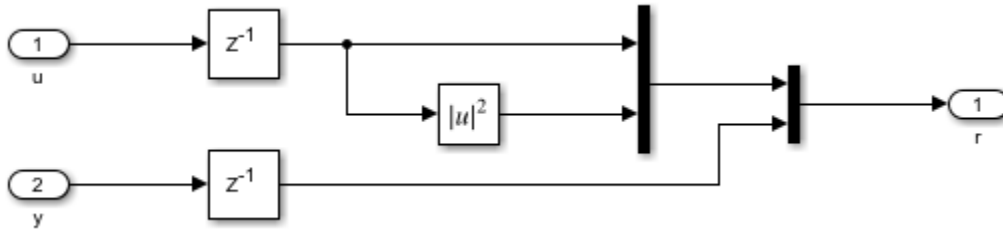

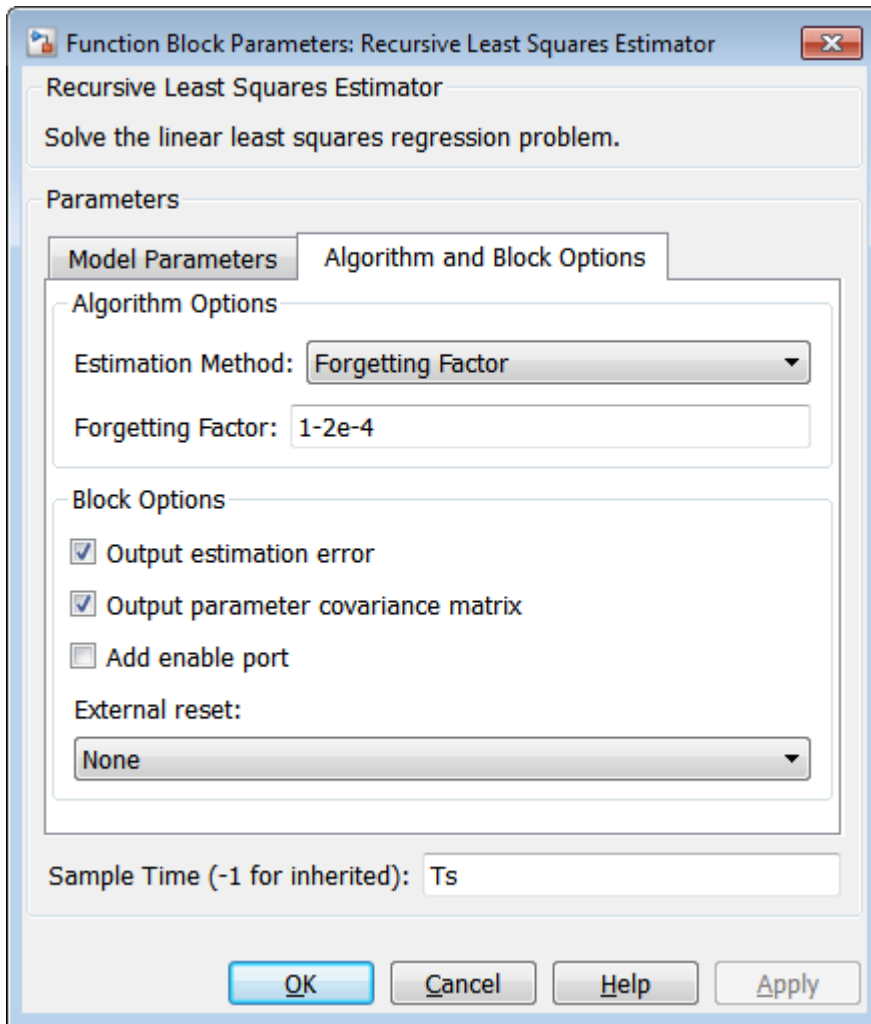Configure the Recursive Least Squares Estimator block:

- **Initial Estimate:** None. By default, the software uses a value of 1.

- **Number of parameters**: 3, one for each a regressor coefficient.

- **Parameter Covariance Matrix**: 1, the amount of uncertainty in initial guess of 1. Concretely, treat the estimated parameters as a random variable with variance 1.

- **Sample Time**: T_s.
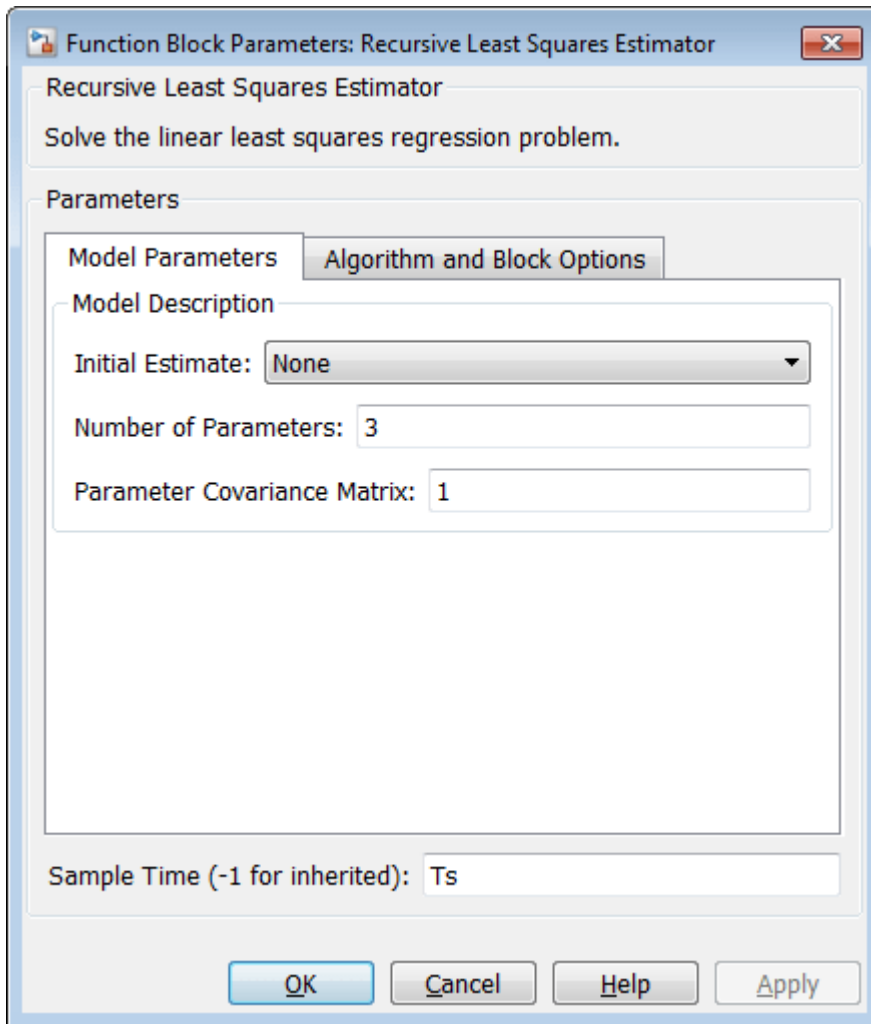
Click **Algorithm and Block Options** to set the estimation options:

- **Estimation Method**: `Forgetting Factor`

- **Forgetting Factor**: 1-2e-4. Since the estimated a values are expected to change with the inertia, set the forgetting factor to a value less than 1. Choose $\lambda = 1\text{-}2\text{e-}4$ which corresponds to a memory time constant of $T_0 = \frac{T_s}{1-\lambda}$ or 15 seconds. A 15 second memory time ensures that significant data from both the open and closed throttle position are used for estimation as the position is changed every 10 seconds.

- Select the **Output estimation error** check box. You use this block output to validate the estimation.

- Select the **Output parameter covariance matrix** check box. You use this block output to validate the estimation.

- Clear the **Add enable port** check box.
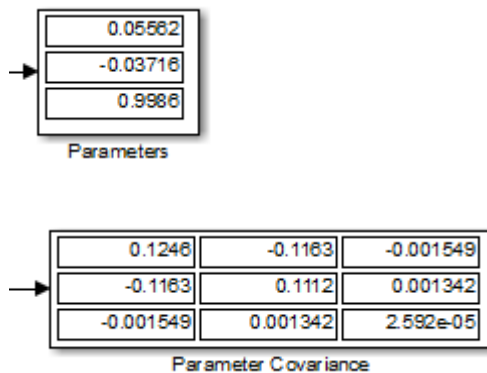
- **External reset**: None.

Validating the Estimated Model

The `Error` output of the `Recursive Least Squares Estimator` block gives the one-step-ahead error for the estimated model. This error is less than 5% indicating that for one-step-ahead prediction the estimated model is accurate.

```
open_system('iddemo_engine/Error (%)')
```
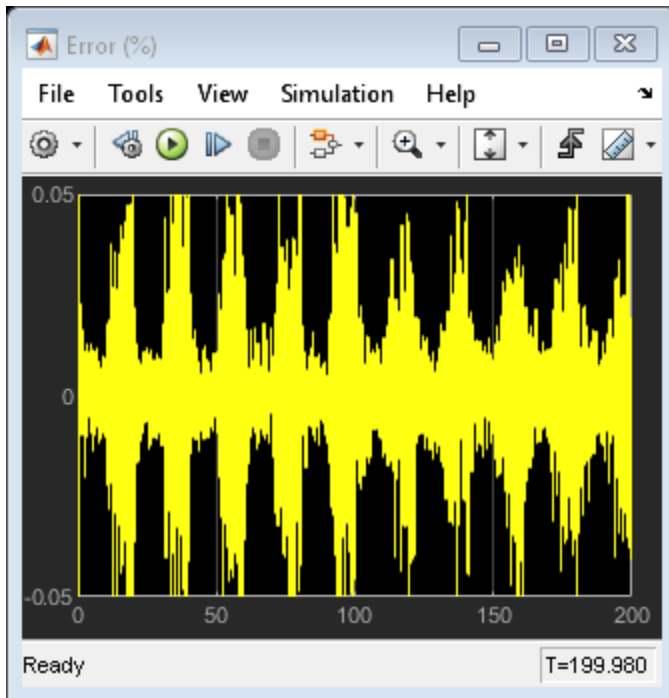
The diagonal of the parameter covariances matrix gives the variances for the a_n parameters. The a_3 variance is small relative to the parameter value indicating good confidence in the estimated value. In contrast, the a_1, a_2 variances are large relative to the parameter values indicating a low confidence in these values.

| 0.05562 |
| -0.03716 |
| 0.9986 |

Parameters

0.1246	-0.1163	-0.001549
-0.1163	0.1112	0.001342
-0.001549	0.001342	2.592e-05

Parameter Covariance

While the small estimation error and covariances give confidence that the model is being estimated correctly, it is limited in that the error is a one-step-ahead predictor. A more rigorous check is to use the estimated model in a simulation model and compare with the actual model output. The **Estimated Model** section of the simulink model implements this.

The `Regressors1` block is identical to the `Regressors` block use in the recursive estimator. The only difference is that the y signal is not measured from the plant but fed back from the output of the estimated model. The Output of the regressors block is multiplied by estimated a_n values to give $\hat{y}_n$ an estimate of the engine speed.

```
open_system('iddemo_engine/trpm Est')
```
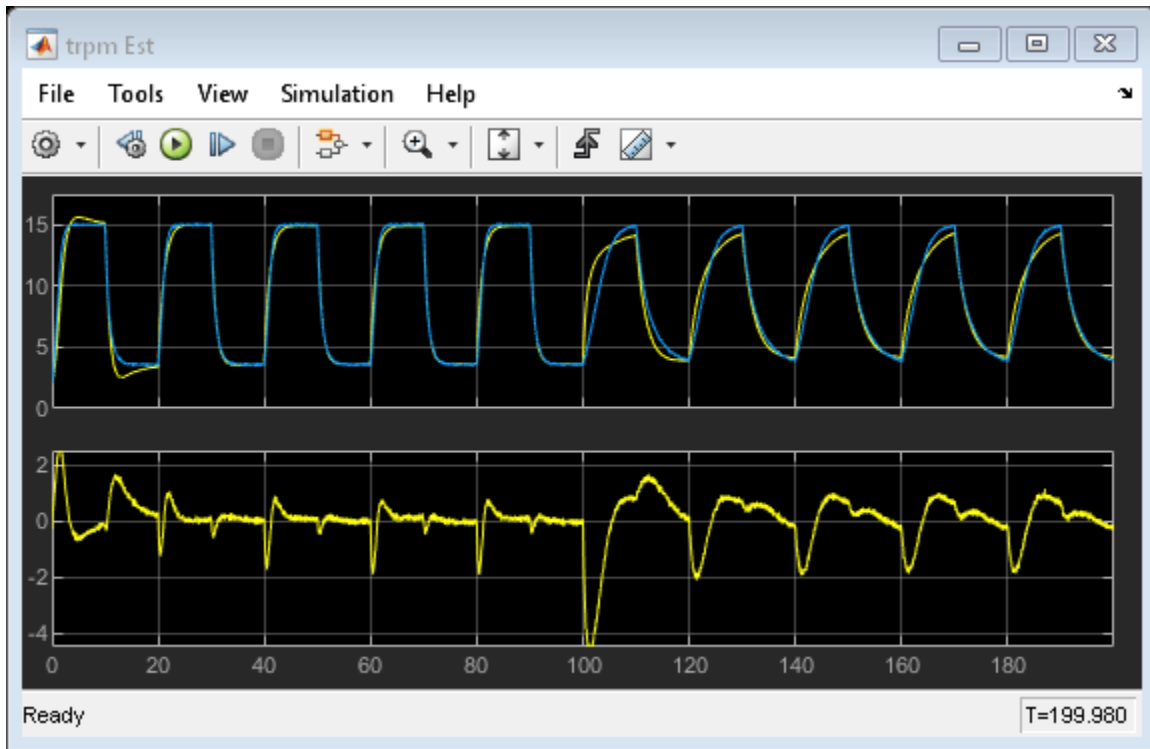
The estimated model output matches the model output fairly well. The steady-state values are close and the transient behavior is slightly different but not significantly so. Note that after 100 seconds when the engine inertia changes the estimated model output differs slightly more from the model output. This implies that the chosen regressors cannot capture the behavior of the model as well after the inertia change. This also suggests a change in system behavior.

The estimated model output combined with the low one-step-ahead error and parameter covariances gives us confidence in the recursive estimator.

Detecting Changes in Engine Inertia

The engine model is setup to introduce an inertia change 100 seconds into the simulation. The recursive estimator can be used to detect the change in inertia.

The recursive estimator takes around 50 seconds to converge to an initial set of parameter values. To detect the inertia change we examine the a_1 model coefficient that influences the $a_1 u_{n-1}$ term of the estimated model.

```
open_system('iddemo_engine/Detect Inertia Change')
```
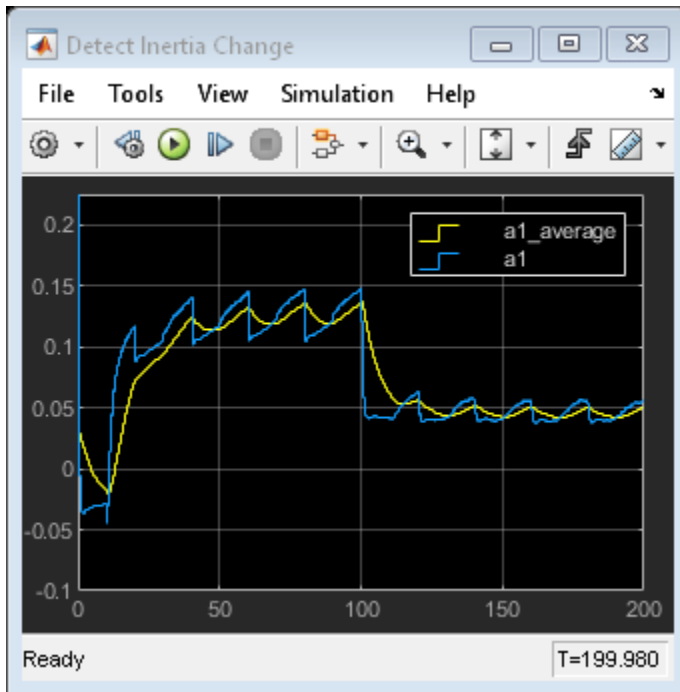
The covariance for a_1, 0.05562, is large relative to the parameter value 0.1246 indicating low confidence in the estimated value. The time plot of a_1 shows why the covariance is large. Specifically a_1 is varying as the throttle position varies indicating that the estimated model is not rich enough to fully capture different rise times at different throttle positions and needs to adjust a_1. However, we can use this to identify the inertia changes as the average value of a_1 changes as the inertia changes. You can use a threshold detector on the moving average of the a_1 parameter to detect changes in the engine inertia.

```
bdclose('iddemo_engine')
```

Online ARMAX Polynomial Model Estimation

This example shows how to implement an online polynomial model estimator. You estimate two ARMAX models for a nonlinear chemical reaction process. These models capture the behavior of the process at two operating conditions. The model behavior is identified online and used to adjust the gains of an adaptive PI controller during system operation.

Continuously Stirred Tank Reactor

A Continuously Stirred Tank Reactor (CSTR) is a common chemical system in the process industry. A schematic of the CSTR system is:
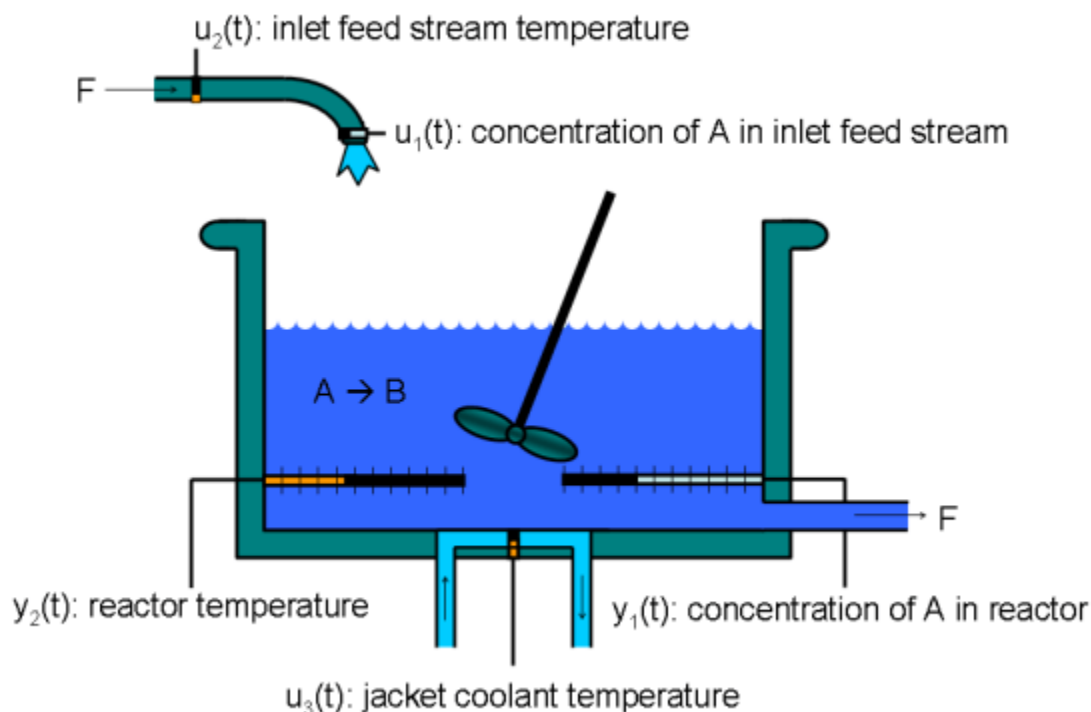
This is a jacketed diabatic (i.e., nondiabatic) tank reactor described extensively in Bequette's book "Process Dynamics: Modeling, Analysis and Simulation", published by Prentice-Hall, 1998. The vessel is assumed to be perfectly mixed, and a single first-order exothermic and irreversible reaction, A --> B, takes place. The inlet stream of reagent A is fed to the tank at a constant rate. After stirring, the end product streams out of the vessel at the same rate as reagent A is fed into the tank (the volume in the reactor tank is constant). Details of the operation of the CSTR and its 2-state nonlinear model used in this example are explained in the example "Non-Adiabatic Continuous Stirred Tank Reactor: MATLAB File Modeling with Simulations in Simulink" on page 13-218.

The inputs of the CSTR model are:

$$u_1(t) = C_{Af}(t) \quad \text{Concentration of A in inlet feed stream} [kgmol/m^3]$$
$$u_2(t) = T_f(t) \quad \text{Inlet feed stream temperature} [K]$$
$$u_3(t) = T_j(t) \quad \text{Jacket coolant temperature} [K]$$

and the outputs (y(t)), which are also the states of the model (x(t)), are:

$$y_1(t) = x_1(t) = C_A(t) \quad \text{Concentration of A in reactor tank}[kgmol/m^3]$$
$$y_2(t) = x_2(t) = T(t) \quad \text{Reactor temperature}[K]$$

The control objective is to maintain the concentration of reagent A, $C_A(t)$ at the desired level $C_{Aref}(t)$, which changes over time. The jacket temperature $T_j(t)$ is manipulated by a PI controller in order to reject disturbances arising from the inlet feed stream temperature $T_f(t)$. The input of the PI controller is the tracking error signal, $C_{Aref}(t) - C_A(t)$. The inlet feed stream concentration, $C_{Af}(t)$, is assumed to be constant. The Simulink model `iddemo_cstr` implements the CSTR plant as the block CSTR.

```
open_system('iddemo_cstr');
```


Copyright 2013-2018 The MathWorks, Inc.
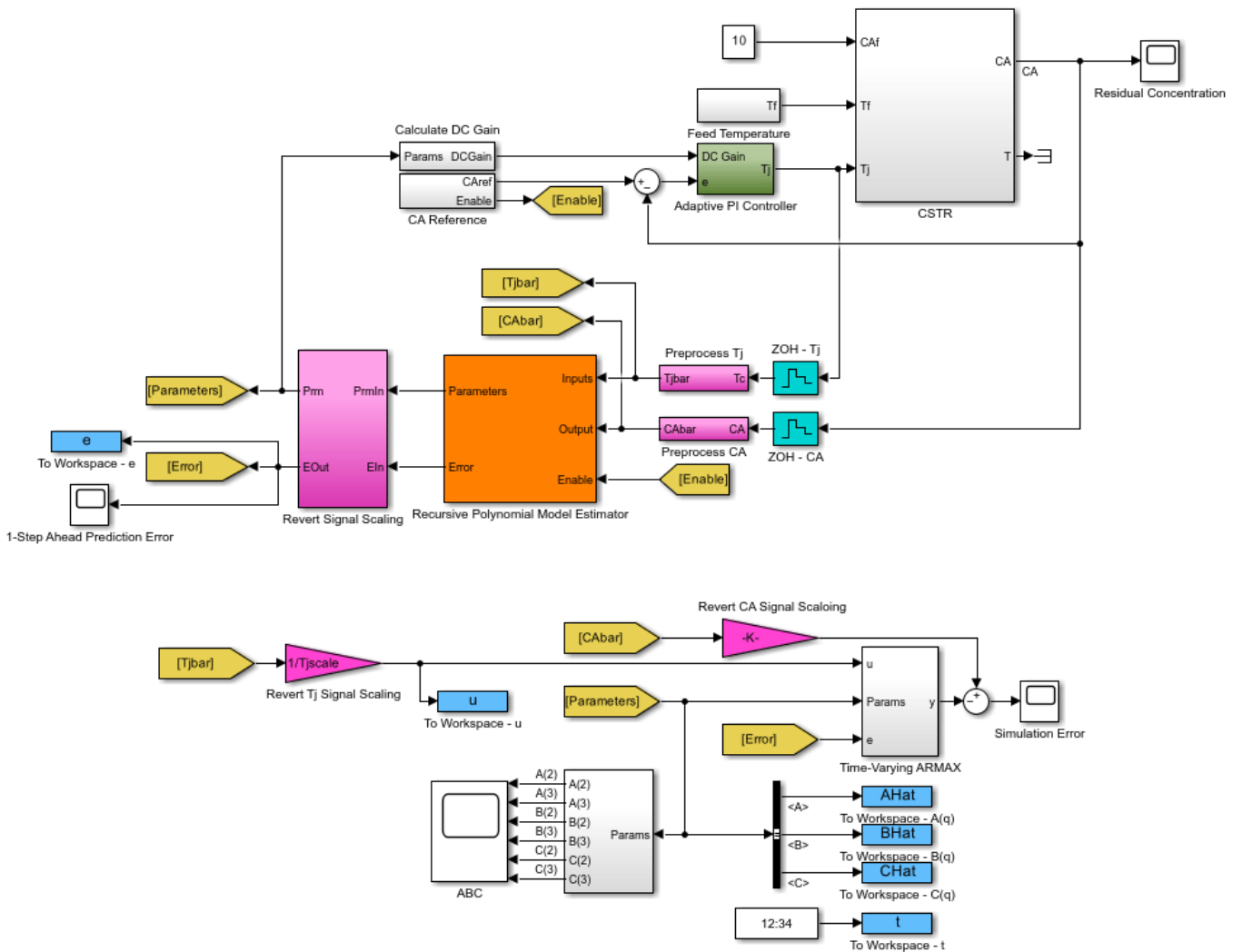
The $T_f(t)$ feed temperature input consists of a white noise disturbance on top of a constant offset. The noise power is 0.0075 [K^2]. This noise level causes up to 2% deviation from the desired $C_{Aref}(t)$.

The $C_{Aref}(t)$ signal in this example contains a step change from 1.5 [kgmol/m^3] to 2 [kgmol/m^3] at time $t = 400$. In addition to this step change, $C_{Aref}(t)$ also contains a white noise perturbation for t in the [0,200) and [400,600) ranges. The power of this white noise signal is 0.015. The noise power is adjusted empirically to approximately give a signal-to-noise ratio of 10. Not having sufficient excitation in the reference signal in closed-loop identification can lead to not having sufficient information to identify a unique model. The implementation of $C_{Aref}(t)$ is in the `iddemo_cstr/CA Reference` block.

Online Estimation for Adaptive Control

It is known from the nonlinear model that the CSTR output $C_A(t)$ is more sensitive to the control input $T_j(t)$ at higher $C_A(t)$ levels. The Recursive Polynomial Model Estimator block is used to detect this change in sensitivity. This information is used to adjust the gains of the PI controller as $C_A(t)$ varies. The aim is to avoid having a a high gain control loop which may lead to instability.

You estimate a discrete transfer-function from $T_j(t)$ to $C_A(t)$ online with the Recursive Polynomial Model Estimator block. The adaptive control algorithm uses the DC gain of this transfer function. The tracking error $C_{Aref}(t) - C_A(t)$, is divided by the normalized DC gain of the estimated transfer function. This normalization is done to have a gain of 1 on the tracking error at the initial operating point, for which the PI controller is designed. For instance, the error signal is divided by 2 if the DC gain becomes 2 times its original value. This corresponds to dividing the PI controller gains by 2. This adaptive controller is implemented in `iddemo_cstr/Adaptive PI Controller`.

Recursive Polynomial Model Estimator Block Inputs

The 'Recursive Polynomial Model Estimator' block is found under the `System Identification Toolbox/Estimators` library in Simulink. You use this block to estimate linear models with ARMAX structure. ARMAX models have the form:

$$A(q)\bar{y}(t) = B(q)\bar{u}(t) + C(q)\bar{e}(t)$$

$$A(q) = 1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_{na} z^{-na}$$
$$B(q) = (b_0 1 + b_1 z^{-1} + b_2 z^{-2} + \cdots + a_{nb-1} z^{-nb+1}) z^{-nk}$$
$$C(q) = 1 + c_1 z^{-1} + c_2 z^{-2} + \cdots + c_{nc} z^{-nc}$$

- The `Inputs` and `Output` inport of the recursive polynomial model estimator block correspond to $\bar{u}(t)$ and $\bar{y}(t)$ respectively. For the CSTR model $\bar{y}$ and $\bar{u}$ are deviations from the jacket temperature and A concentration trim operating points: $\bar{y} = C_A(t) - \bar{C}_A(t)$, $\bar{u} = T_j(t) - \bar{T}_j(t)$. It is good to scale $\bar{u}$ and $\bar{y}$ to have a peak amplitude of 1 to improve the numerical condition of the estimation problem. The trim operating points, $\bar{C}_A(t)$ and $\bar{T}_j(t)$, are not known exactly before system operation. They are estimated and extracted from the measured signals by using a first-order moving average filter. These preprocessing filters are implemented in the `iddemo_cstr/ Preprocess Tj` and `iddemo_cstr/Preprocess CA` blocks.

```
open_system('iddemo_cstr/Preprocess Tj');
```
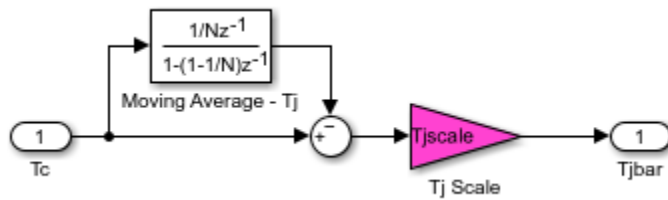
- The optional `Enable` inport of the Recursive Polynomial Model Estimator block controls the parameter estimation in the block. Parameter estimation is disabled when the Enable signal is zero. Parameter estimation is enabled for all other values of the `Enable` signal. In this example the estimation is disabled for the time intervals $t \in [200, 400)$ and $t \in [600, 800)$. During these intervals the measured input $T_j(t)$ does not contain sufficient excitation for closed-loop system identification.

Recursive Polynomial Model Estimator Block Setup

Configure the block parameters to estimate a second-order ARMAX model. In the **Model Parameters** tab, specify:

- **Model Structure:** ARMAX. Choose ARMAX since the current and past values of the disturbances acting on the system, $T_f(t)$, are expected to impact the CSTR system output $C_A(t)$.

- **Initial Estimate:** None. By default, the software uses a value of 0 for all estimated parameters.

- **Number of parameters in A(q) (na)**: 2. The nonlinear model has 2 states.

- **Number of parameters in B(q) (nb)**: 2.

- **Number of parameters in C(q) (nc)**: 2. The estimated model corresponds to a second order model since the maximum of na, nb, and nc are 2.

- **Input Delay (nk)**: 1. Like most physical systems, the CSTR system does not have direct feedthrough. Also, there are no extra time delays between its I/Os.

- **Parameter Covariance Matrix**: 1e4. Specify a high covariance value because the initial guess values are highly uncertain.

- **Sample Time**: 0.1. The CSTR model is known to have a bandwidth of about 0.25Hz. $T_s = 0.1$ chosen such that 1/0.1 is greater than 20 times this bandwidth (5Hz).
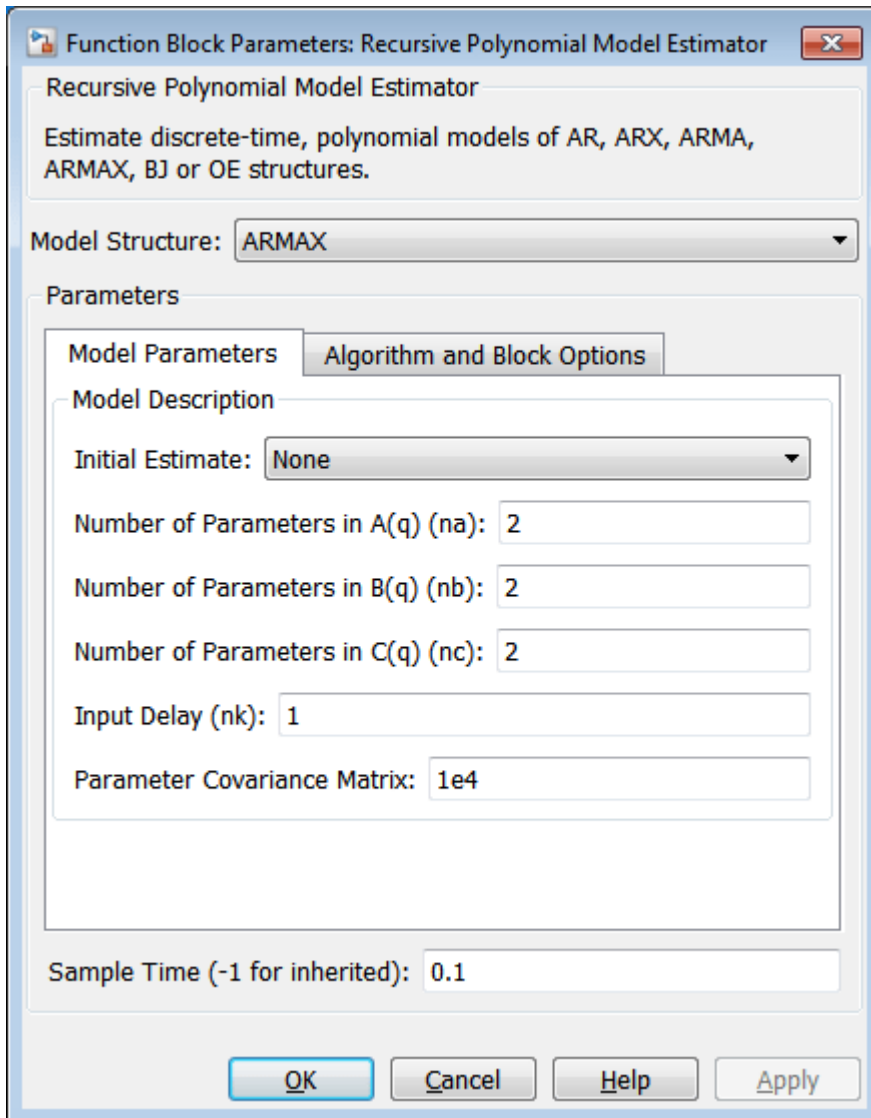
Click **Algorithm and Block Options** to set the estimation options:

- **Estimation Method**: Forgetting Factor

- **Forgetting Factor**: 1-5e-3. Since the estimated parameters are expected to change with the operating point, set the forgetting factor to a value less than 1. Choose $\lambda = 1 - 5e - 3$ which corresponds to a memory time constant of $T_0 = \frac{Ts}{1-\lambda} = 100$ seconds. A 100 second memory time ensures that a significant amount data used for identification is coming from the 200 second identification period at each operating point.

- Select the **Output estimation error** check box. You use this block output to validate the estimation.

- Select the **Add enable port** check box. You only want to adapt the estimated model parameters when extra noise is injected in the reference port. The parameter estimation n is disabled through this port when the extra noise is no longer injected.

- **External reset**: None.
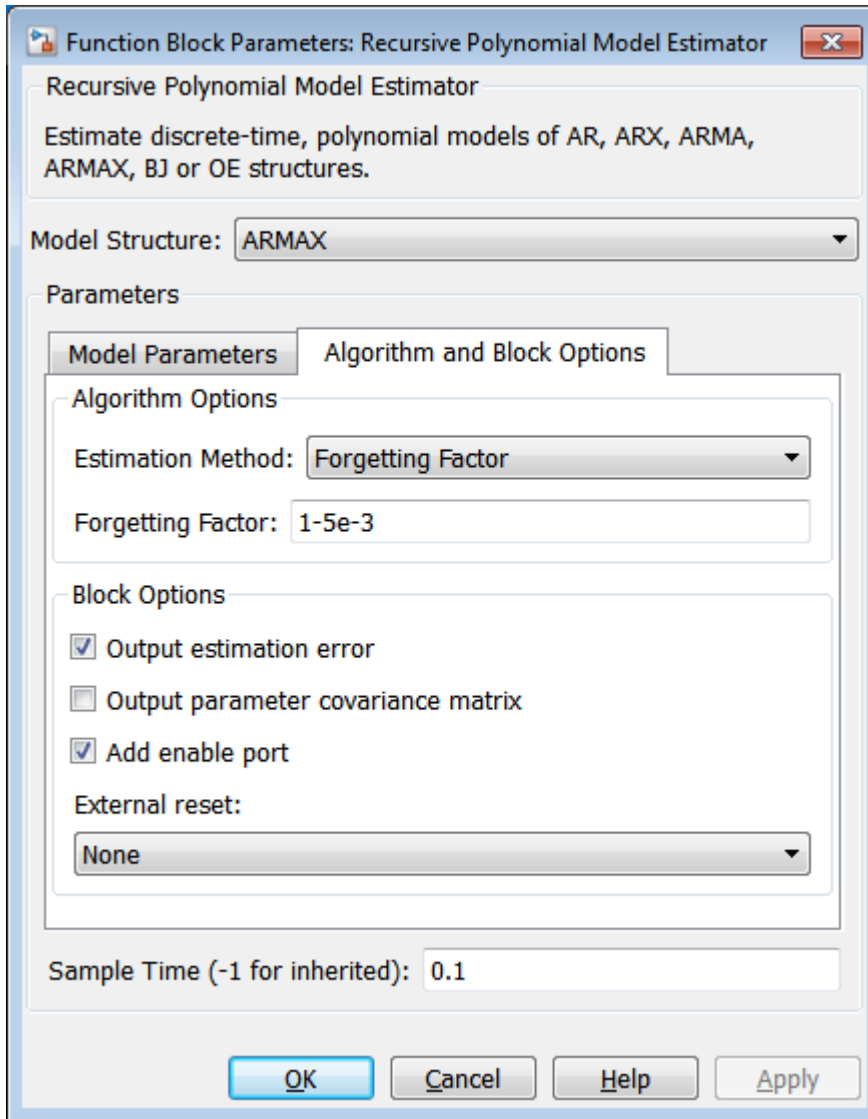
Recursive Polynomial Model Estimator Block Outputs

At every time step, the recursive polynomial model estimator provides an estimate for $A(q)$, $B(q)$, $C(q)$, and the estimation error $\bar{e}$. The `Error` outport of the polynomial model estimator block contains $\bar{e}(t)$ and is also known as the one-step-ahead prediction error. The `Parameters` outport of the block contains the A(q), B(q), and C(q) polynomial coefficients in a bus signal. Given the chosen polynomial orders ($na = 2$, $nb = 2$, $nc = 2$, $nk = 1$) the `Parameters` bus elements contain:

$$
\begin{aligned}
A(t) &= [1 \ a_1(t) \ a_2(t)] \\
B(t) &= [0 \ b_0(t) \ b_1(t)] \\
C(t) &= [1 \ c_1(t) \ c_2(t)]
\end{aligned}
$$

The estimated parameters in the A(q), B(q), and C(q) polynomials change during simulation as follows:

```
sim('iddemo_cstr');
close_system('iddemo_cstr/Preprocess Tj');
open_system('iddemo_cstr/ABC');
```
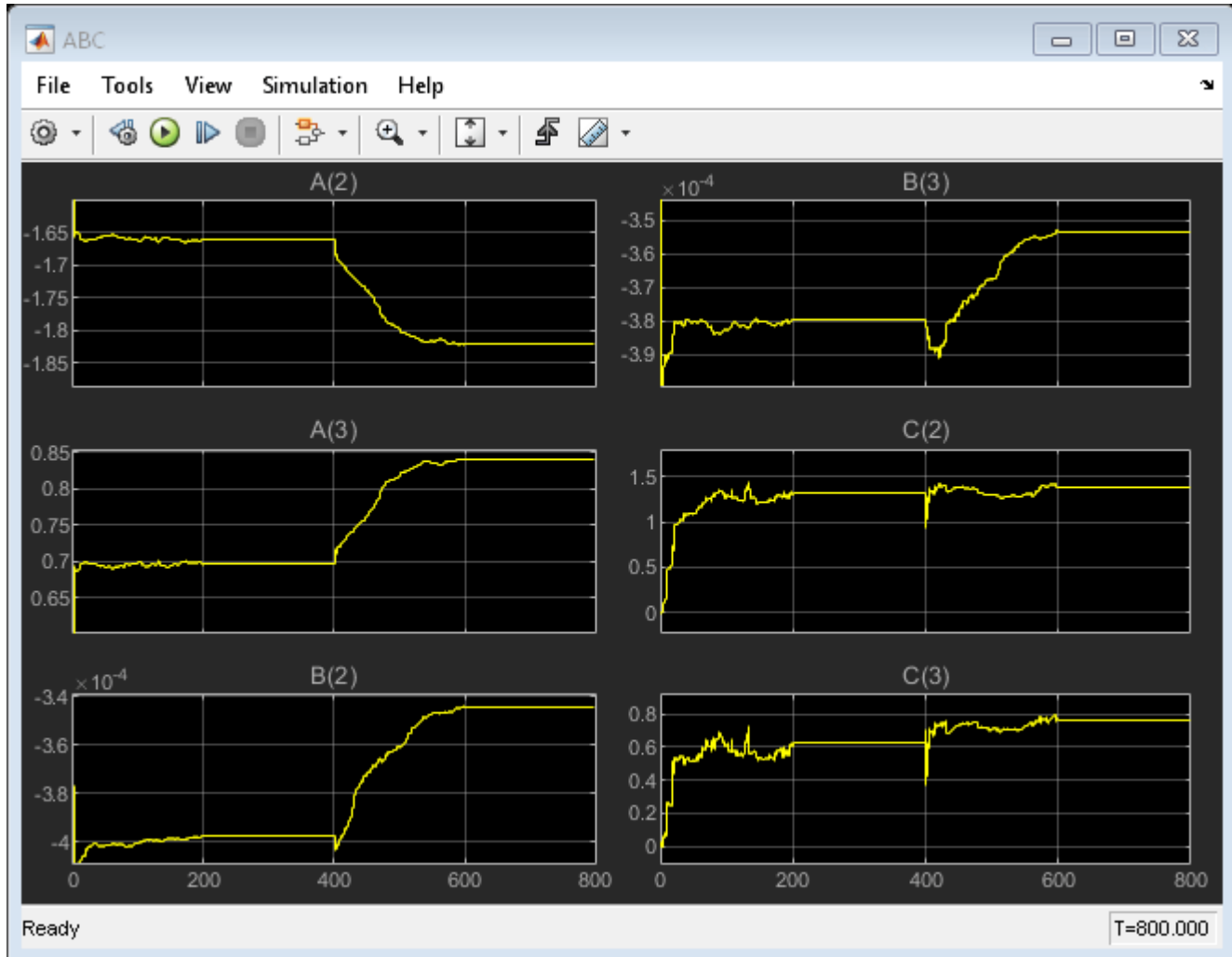

The parameter estimates quickly change from their initial values of 0 due to the high value chosen for the initial parameter covariance matrix. The parameters in the $A(q)$ and $B(q)$ polynomials approach their values at $t = 200$ rapidly. However, the parameters in the $C(q)$ polynomial show some fluctuations. One reason behind these fluctuations is that the disturbance $T_f(t)$ to CSTR output $C_A(t)$ is not fully modelled by the ARMAX structure. The error model $C(q)$ is not important for the control problem studied here since the $T_j(t)$ to $C_A(t)$ relationship is captured by the transfer function $\frac{B(q)}{A(q)}$. Therefore, the fluctuation in $C(q)$ is not a concern for this identification and control problem.

The parameter estimates are held constant for $t \in [200, 400)$ since the estimator block was disabled for this interval (0 signal to the Enable inport). The parameter estimation is enabled at $t = 400$ when the CSTR tank starts switching to its new operating point. The parameters of $A(q)$ and $B(q)$ converge to their new values by $t = 600$, and then held constant by setting the Enable port to 0. The

convergence of $A(q)$ and $B(q)$ is slower at this operating point. This slow convergence is because of the smaller eigenvalues of the parameter covariance matrix at t=400 compared to the initial 1e4 values set at t=0. The parameter covariance, which is a measure of confidence in the estimates, is updated with each time step. The algorithm quickly changed the parameter estimates when the confidence in estimates were low at t=0. The improved parameter estimates capture the system behavior better, resulting in smaller one-step-ahead prediction errors and smaller eigenvalues in the parameter covariance matrix (increased confidence). The system behavior changes in t=400. However, the block is slower to change the parameter estimates due to the increased confidence in the estimates. You can use the External Reset option of the Recursive Polynomial Model Estimator block to provide a new value for parameter covariance at t=400. To see the value of the parameter covariance, select the **Output parameter covariance matrix** check box in the Recursive Polynomial Model Estimator block.

Validating the Estimated Model

The `Error` output of the `Recursive Polynomial Model Estimator` block gives the one-step-ahead error for the estimated model.

```
close_system('iddemo_cstr/ABC');
open_system('iddemo_cstr/1-Step Ahead Prediction Error');
```
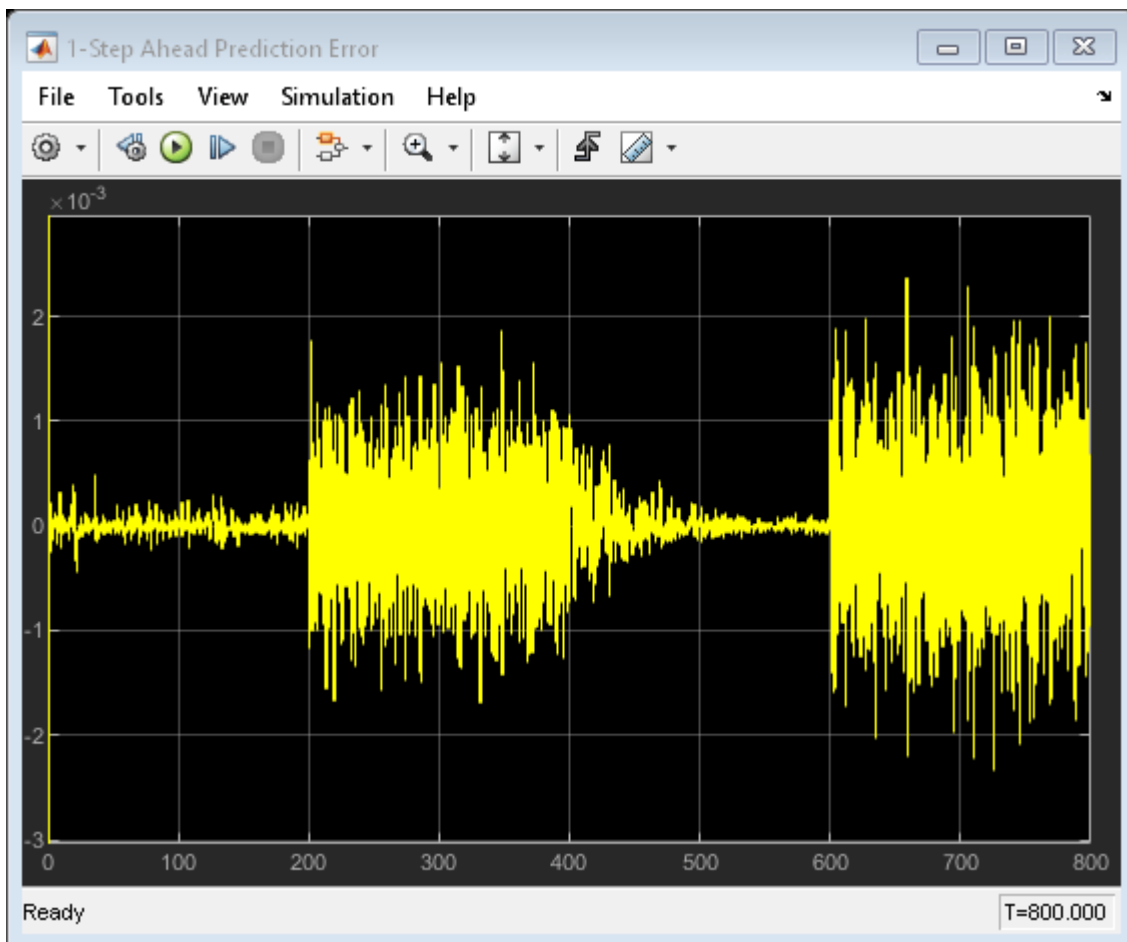
The one-step-ahead error is higher when there are no extra perturbations injected in the $T_j(t)$ channel for system identification. These higher errors may be caused by the lack of sufficient information in the $T_j(t)$ input channel that the estimator block relies on. However, even this higher error is low and bounded when compared to the measured fluctuations in $C_A(t)$. This gives confidence in the estimated parameter values.

A more rigorous check of the estimated model is to simulate the estimated model and compare with the actual model output. The `iddemo_cstr/Time-Varying` ARMAX block implements the time-varying ARMAX model estimated by the Online Polynomial Model Estimator block. The error between the output of the CSTR system and the estimated time-varying ARMAX model output is:

```
close_system('iddemo_cstr/1-Step Ahead Prediction Error');
open_system('iddemo_cstr/Simulation Error');
```
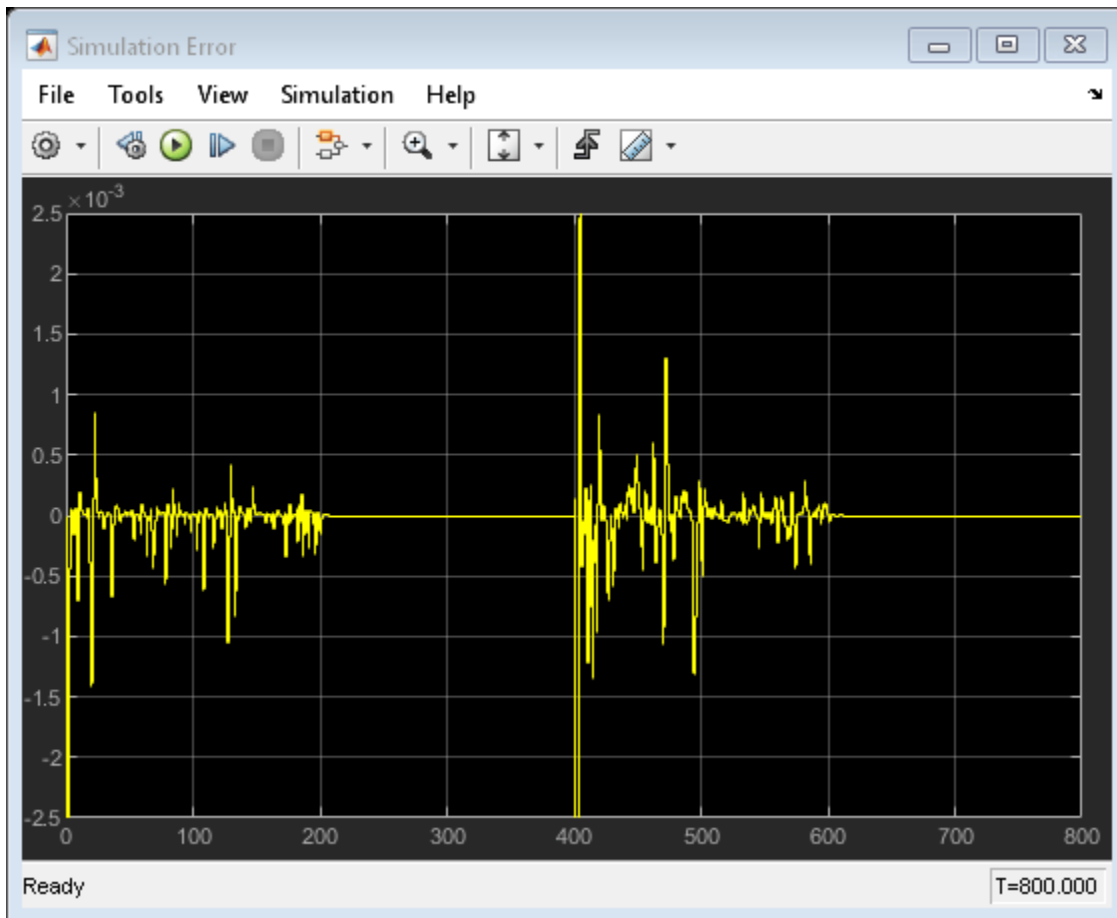

The simulation error is again bounded and low when compared to the fluctuations in the $C_A(t)$. This further provides confidence that the estimated linear models are able to predict the nonlinear CSTR model behavior.

The identified models can be further analyzed in MATLAB. The model estimates for the operating points $C_A = 1.5[kgmol/m^3]$ and $C_A = 2[kgmol/m^3]$ can be obtained by looking at the estimated A(q), B(q), and C(q) polynomials at $t = 200$ and $t = 600$ respectively. Bode plots of these models are:

```
Ts = 0.1;
tidx = find(t>=200,1);
P200 = idpoly(AHat(:,:,tidx),BHat(:,:,tidx),CHat(:,:,tidx),1,1,[],Ts);
tidx = find(t>=600,1);
P600 = idpoly(AHat(:,:,tidx),BHat(:,:,tidx),CHat(:,:,tidx),1,1,[],Ts);
bodemag(P200,'b',P600,'r--',{10^-1,20});
legend('Estimated Model at C_A=1.5 [kgmol/m^3]', ...
       'Estimated Model at C_A=2.0 [kgmol/m^3]', ...
       'Location', 'Best');
```
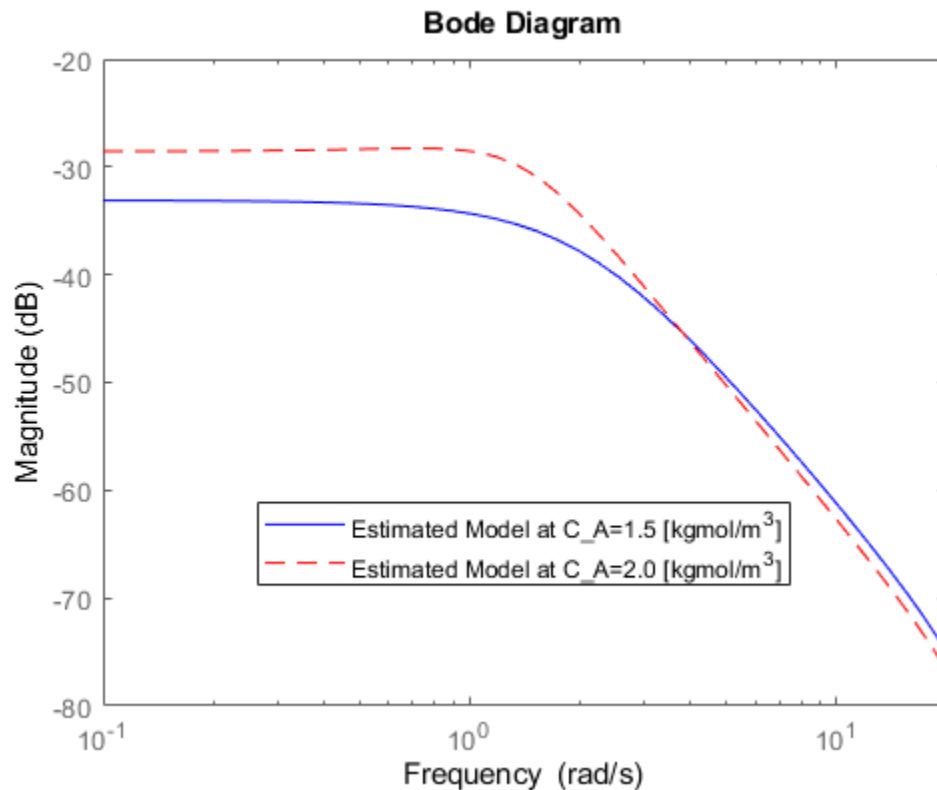

The estimated model has a higher gain at higher concentration levels. This is in agreement with prior knowledge about the nonlinear CSTR plant. The transfer function at $C_A(t) = 2[kgmol/m^3]$ has a $6dB$ higher gain (double the amplitude) at low frequencies.

Summary

You estimated two ARMAX models to capture the behavior of the nonlinear CSTR plant at two operating conditions. The estimation was done during closed-loop operation with an adaptive controller. You looked at two signals to validate the estimation results: One step ahead prediction errors and the errors between the CSTR plant output and the simulation of the estimation model. Both of these errors signals were bounded and small compared to the CSTR plant output. This provided confidence in the estimated ARMAX model parameters.

```
bdclose('iddemo_cstr');
```

Estimate Parameters of System Using Simulink Recursive Estimator Block

This example shows how to estimate the parameters of a two-parameter system and compare the measured and estimated outputs.

This example is the Simulink version of the command-line parameter-estimation example provided in `recursiveLS`.

The system has two parameters and is represented as:

$$y(t) = a_1 u(t) + a_2 u(t-1)$$

Here,

- u and y are the real-time input and output data, respectively.

- $u(t)$ and $u(t-1)$ are the regressors, H, of the system.

- a_1 and a_2 are the parameters, θ, of the system.

Load the data and extract the input, output, and time information.

To directly compare this example with the command-line recursiveLS example, shift the time vector by one position. This shift synchronizes the Simulink input (time starts at 0) with the command-line input (time starts at the first sample time).

```
load iddata3
input = z3.u;
output = z3.y;
time = z3.SamplingInstants-1;
```

Construct Input and Output signals for Simulink.

```
input_sig = timeseries(input,time);
output_sig = timeseries(output,time);
```

Open a preconfigured Simulink model based on the Recursive Least Squares Estimator block. In this model:

- The **input_sig** and **output_sig** blocks import `input_sig` and `output_sig`.

- The model constructs the **Regressors** signal by routing `input_sig` through a delay-line block, and then multiplexing the delayed signal with the original signal.

- An **Error** port provides the estimated error signal. A **Sum** block subtracts this error from `input_sig` to produce the estimated output.

- A **Mux** block then combines the measured (`output_sig`) and estimated output signals so that you can view them together on the scope.

```
rls = 'ex_RLS_Estimator_Block_sb_inf';
open_system(rls)
```
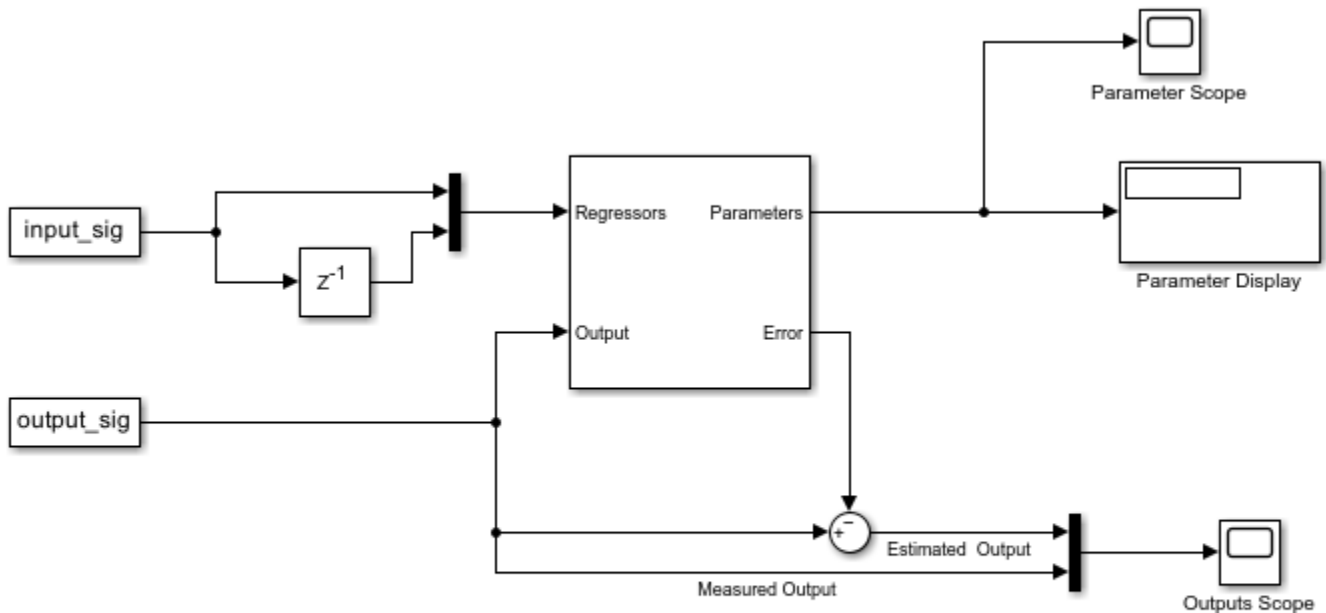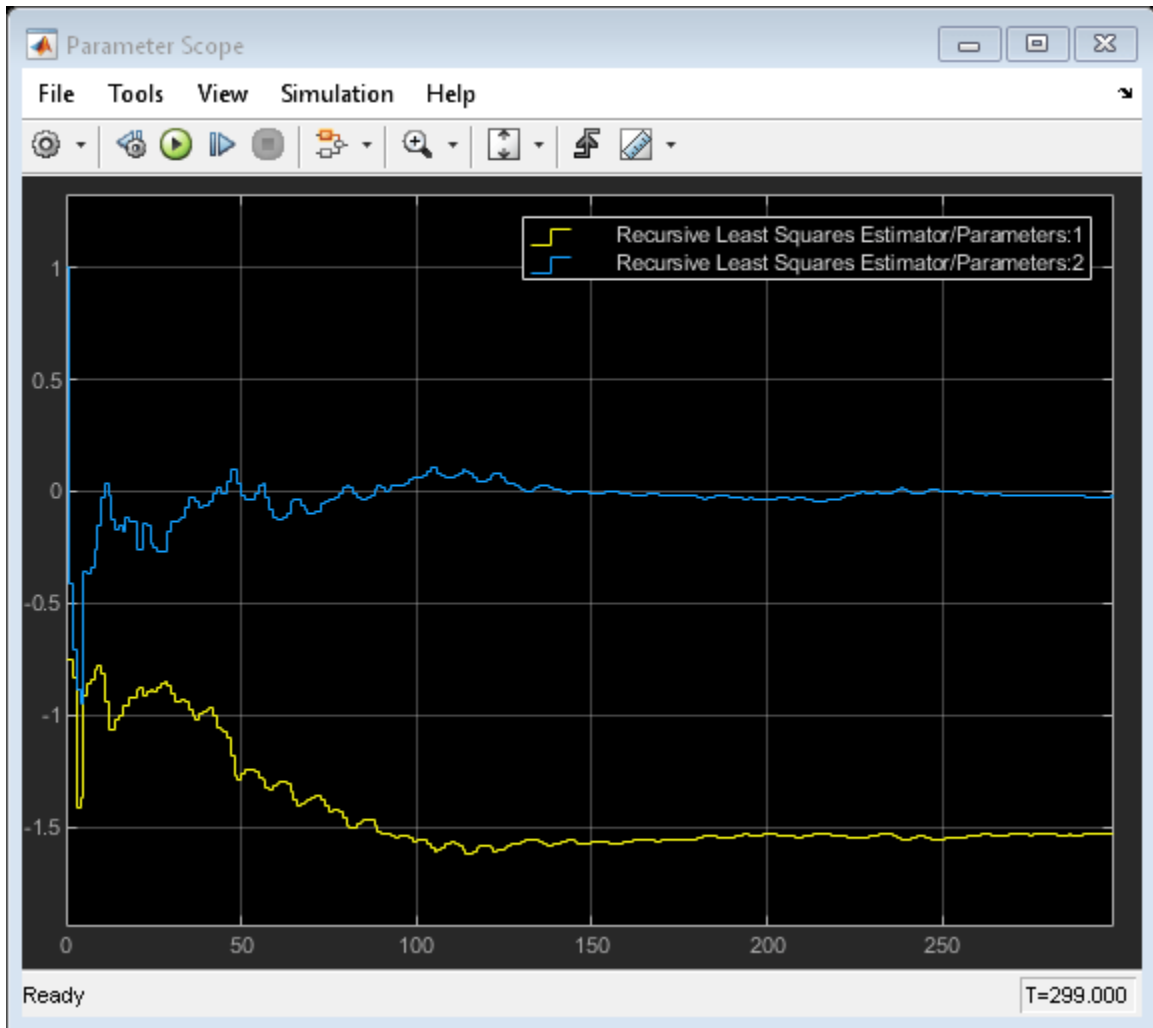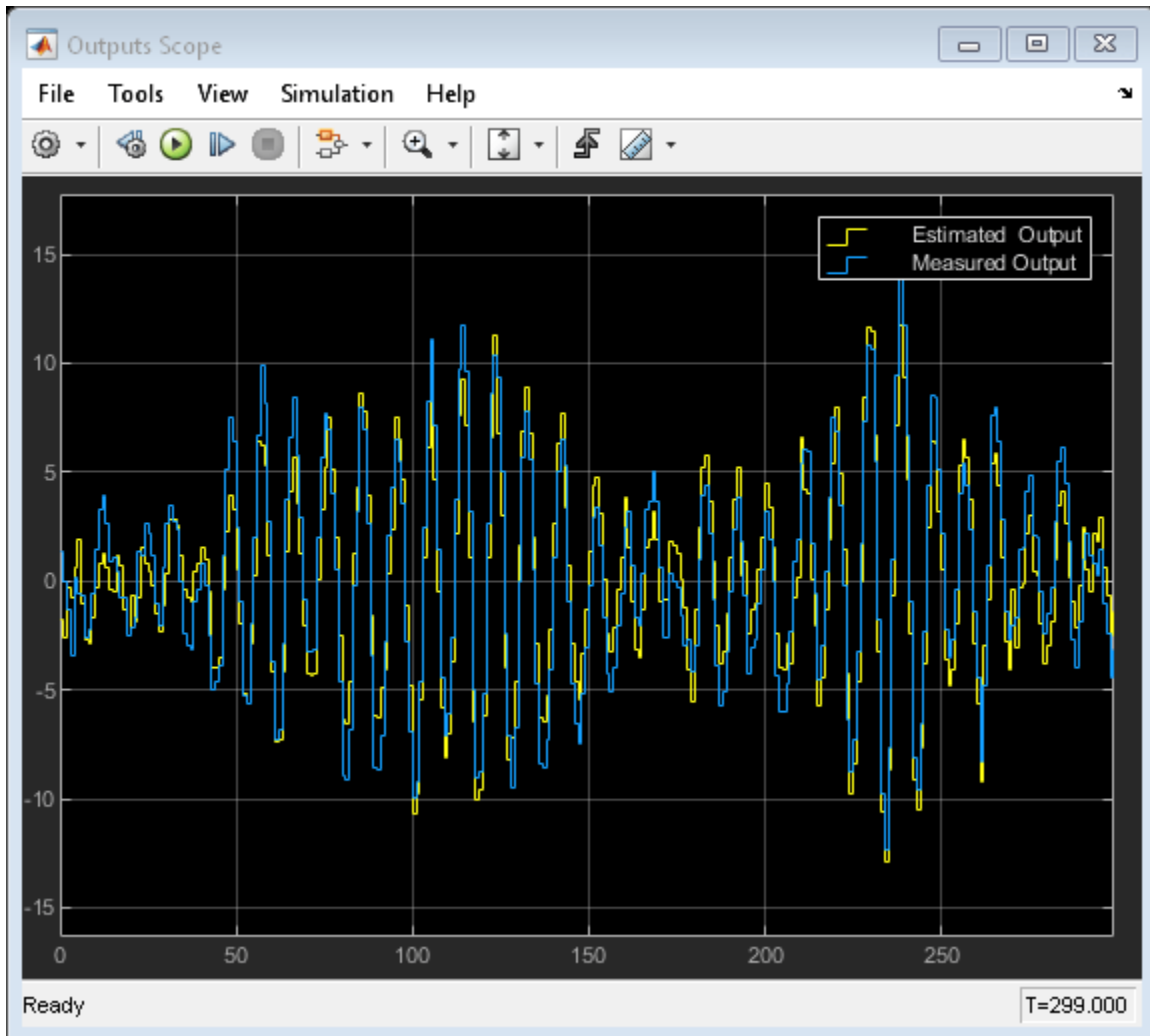
Copyright 2012-2021 The MathWorks Inc.

Run the simulation. The **Parameter Scope** shows the progression of the estimation. The adjacent **Parameter Display** displays the final numerical values. The **Outputs Scope** plots the measured and estimated outputs together.

```
sim(rls)
open_system([rls '/Parameter Scope'])
open_system([rls '/Outputs Scope'])
```

You can explore other block configurations by modifying the model parameters. For instance,

- Change **Estimation Method** to see how your algorithm choices affect your results.
- Change **History** to `finite` and your **Window Length** to `30` to apply the sliding-window algorithm.

See Also

Recursive Least Squares Estimator | Recursive Polynomial Model Estimator | `recursiveLS`

More About

- "Estimate Parameters of System Using Simulink Recursive Estimator Block" on page 16-79
- "Online Recursive Least Squares Estimation" on page 16-61

Use Frame-Based Data for Recursive Estimation in Simulink

This example shows how to use frame-based signals with the **Recursive Least Squares Estimator** block in Simulink®. Machine interfaces often provide sensor data in frames containing multiple samples, rather than in individual samples.

The recursive estimation blocks in the System Identification Toolbox™ accept these frames directly when you set **Input Processing** to `Frame-based`.

The blocks use the same estimation algorithms for sample-based and frame-based input processing. The estimation results are identical. There are some special considerations, however, for working with frame-based inputs in Simulink, and for visualizing the results.

This example is the frame-based version of the sample-based example in "Estimate Parameters of System Using Simulink Recursive Estimator Block" on page 16-79. The function reference for `recursiveLS` provides the equivalent sample-based command-line example.

System Description

The system has two parameters and is represented as:

$$y(t) = a_1 u(t) + a_2 u(t-1)$$

Here,

- u and y are the real-time input and output data samples, respectively
- a_1 and a_2 are the parameters θ.
- $u(t)$ and $u(t-1)$ are the regressor samples. The regressor array H_s is the row vector $H_s = \begin{bmatrix} u(t) & u(t-1) \end{bmatrix}$.
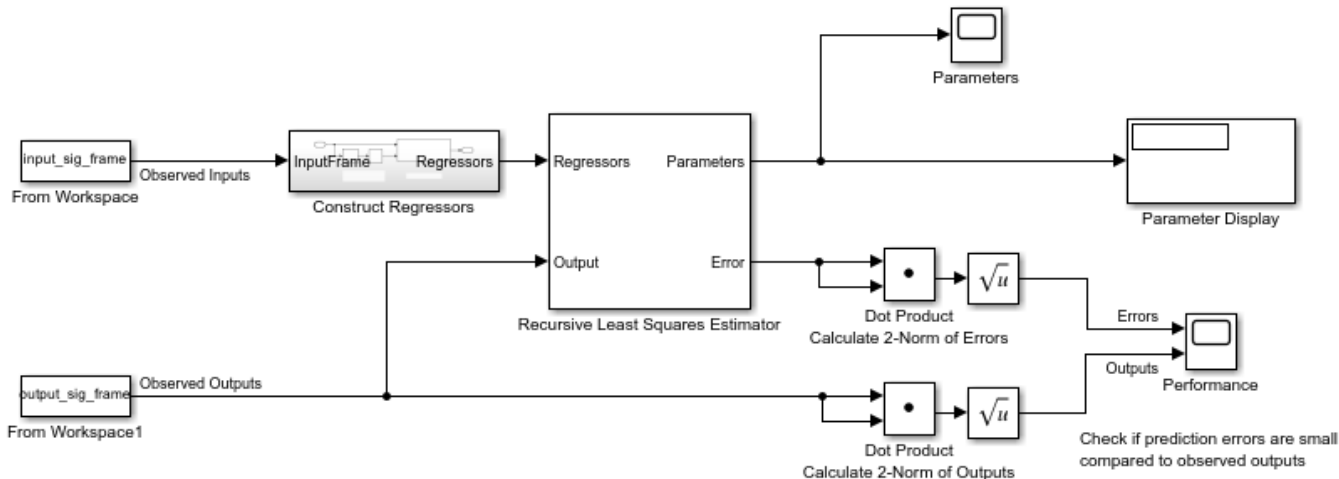
Frame-Based Inputs

Data frames contain multiple data samples stacked in rows, with the last row containing the most recent data. For this example, each frame contains 10 samples. The observed inputs and outputs therefore arrive as $[u(1)...u(10)]^T$ and $[y(1)...y(10)]^T$, where $u(i)$ and $y(i)$ are the positions within the frame for a given time t, and are equivalent to $u(t-10+i)$ and $y(t-10+i)$.

The regressor frame H_f also uses this stacking. The $u(0)$ element in the upper right corner is the last sample from the previous input frame.

$$H_f = \begin{bmatrix} u(1) & u(0) \\ u(2) & u(1) \\ \vdots & \vdots \\ u(10) & u(9) \end{bmatrix}$$

Open a preconfigured Simulink model based on the Recursive Least Squares Estimator block.

```
rlsfb = 'ex_RLS_Estimator_Block_fb';
open_system(rlsfb)
```

Observed Inputs and Outputs

Load the frame-based input and output signals into the workspace. Each signal consists of 30 frames, each frame containing ten individual time samples. In this implementation, the sample time is 1 second, and the frame time is 10 seconds.
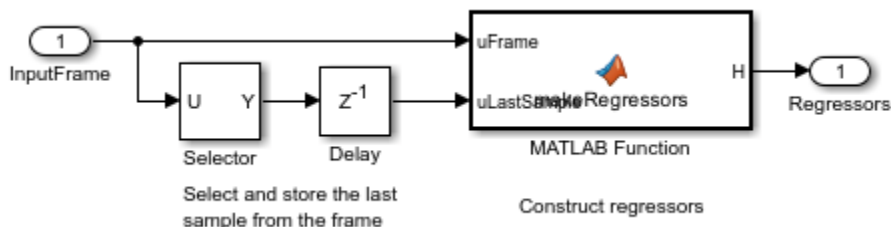
```
load iddata3_frames input_sig_frame output_sig_frame
```

From Workspace blocks import signals that you loaded. In your application, this data may be arriving directly from your hardware.

Regressors

The model constructs the **Regressors** frame Hframe in the **Construct Regressors** subsystem, using input_sig_frame.

```
open_system([rlsfb '/Construct Regressors'])
```


As **Frame-Based Inputs** described, the frame-based regressor signal consists of the all elements of the current frame of observed inputs, along with the last sample from the previous frame. The **Selector** block extracts the last sample from the observed inputs frame, and the **Delay** block stores this value for the next frame step.

Construct Regressors, a **MATLAB Function** block, constructs the signal Hframe. The first column of Hframe contains the current input frame. The second column contains the current frame, shifted by one position. The first element of the second column, representing the oldest sample, is from the previous observed inputs frame.

Recursive Least Squares Estimator Block

The block configuration includes frame-based input processing with a sample time of 10.

Error Visualization

The **Error** port provides the prediction errors in vector-valued frame $[e(1)...e(10)]^T$, where $e(i)$ is the position within the frame at a given time t and is equivalent to $e(t - 10 + i)$. These prediction errors can be compared to observed outputs frame $[y(1)...y(10)]^T$ in multiple ways. This example compares the 2-norms of these vectors. An error vector with a small norm relative to the observed outputs norm suggests successful estimation. The scope displays a plot of the comparison.

Results

Run the simulation. The **Parameters** scope shows the progression of the $a_1(t)$ and $a_2(t)$ parameter estimates. The adjacent **Parameter Display** displays the current values of $a_1(t)$ and $a_2(t)$. The scope **Performance** plots the 2-norms of the observed outputs and prediction error frames, as described in error as it compares with the original measurement signal as described in **Error Visualization**.

```
sim(rlsfb)
open_system([rlsfb '/Parameters'])
open_system([rlsfb '/Performance'])
```
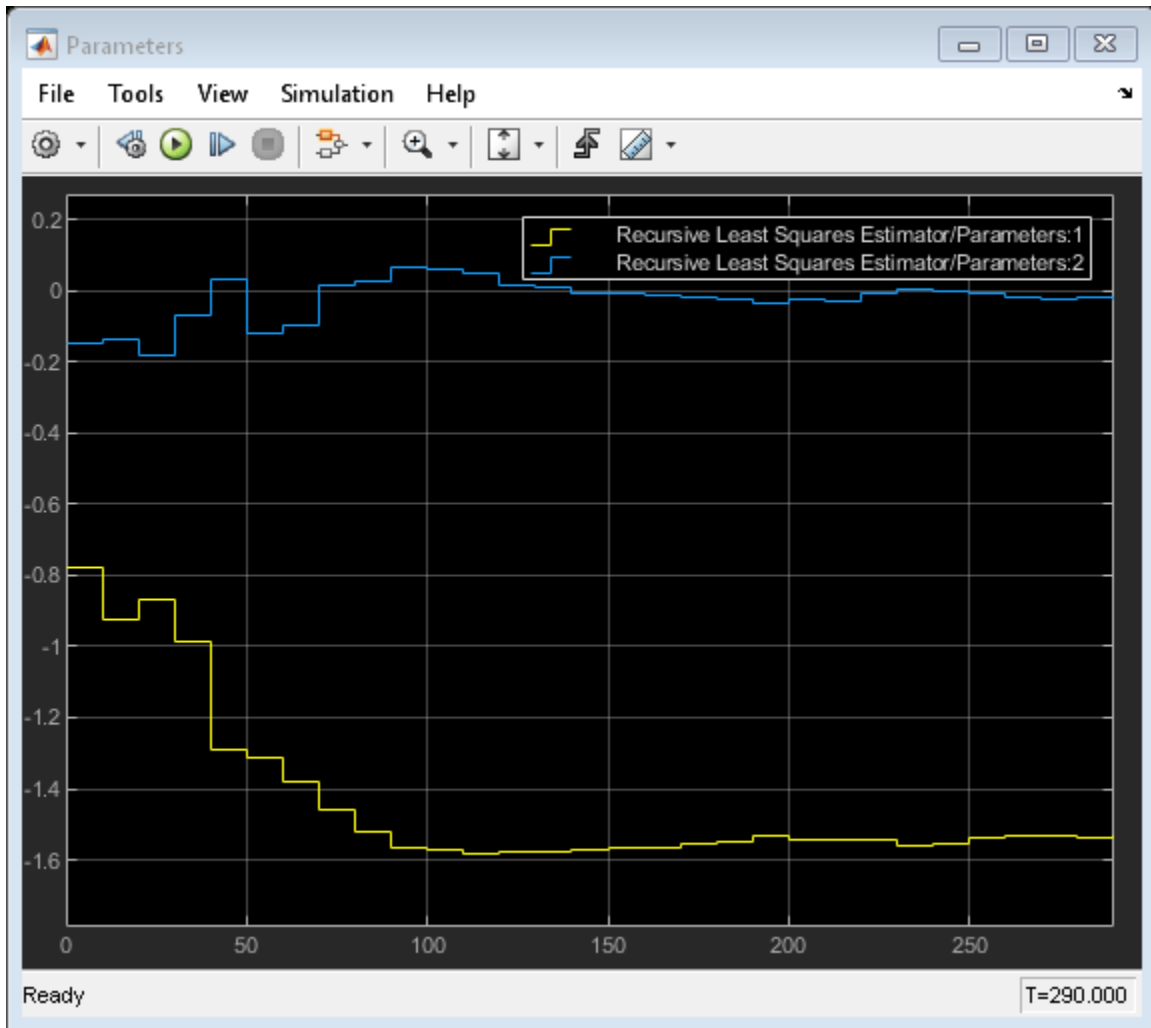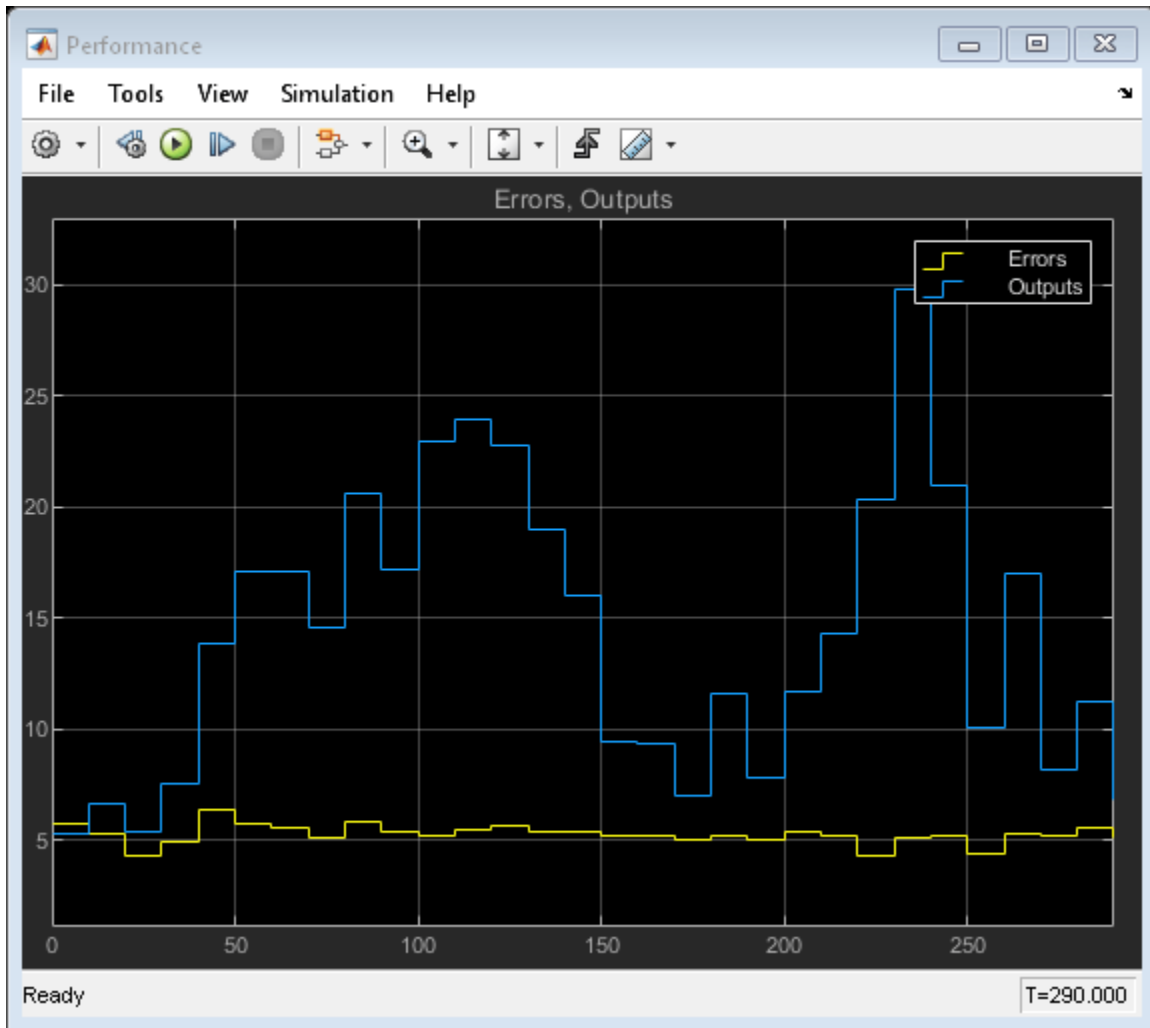
The **Parameters** plot shows that parameter estimates converge at approximately T = 150. The **Performance** plot shows that the prediction errors are small compared with the observed outputs. These results suggest a successful estimation. The final parameter estimates in the Parameter display are identical to those in the command-line and the sample-based Simulink examples linked at the beginning of this example.

See Also

Recursive Least Squares Estimator | Recursive Polynomial Model Estimator | `recursiveLS`

More About

*   "Recursive Algorithms for Online Parameter Estimation" on page 16-16
*   "Estimate Parameters of System Using Simulink Recursive Estimator Block" on page 16-79
*   "Online Recursive Least Squares Estimation" on page 16-61

State Estimation Using Time-Varying Kalman Filter

This example shows how to estimate states of linear systems using time-varying Kalman filters in Simulink®. You use the Kalman Filter block from the `System Identification Toolbox/ Estimators` library to estimate the position and velocity of a ground vehicle based on noisy position measurements such as GPS sensor measurements. The plant model in Kalman filter has time-varying noise characteristics.

Introduction

You want to estimate the position and velocity of a ground vehicle in the north and east directions. The vehicle can move freely in the two-dimensional space without any constraints. You design a multi-purpose navigation and tracking system that can be used for any object and not just a vehicle.
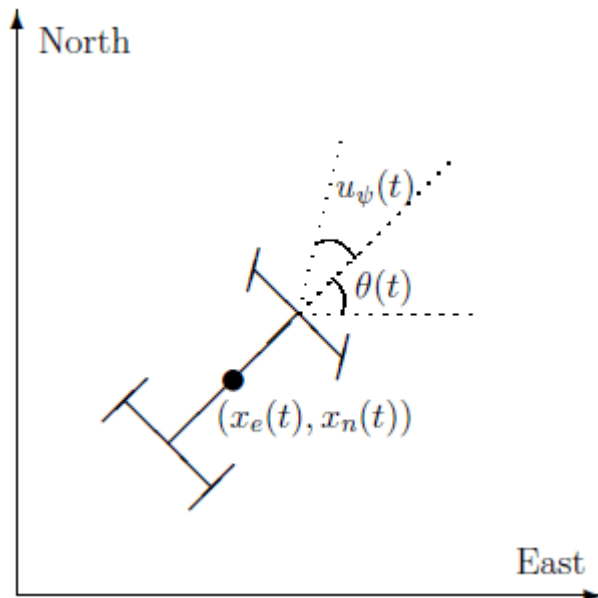

$x_e(t)$ and $x_n(t)$ are the vehicle's east and north positions from the origin, $\theta(t)$ is the vehicle orientation from east and $u_\psi(t)$ is the steering angle of the vehicle. t is the continuous-time variable.

The Simulink model consists of two main parts: Vehicle model and the Kalman filter. These are explained further in the following sections.

```
open_system('ctrlKalmanNavigationExample');
```
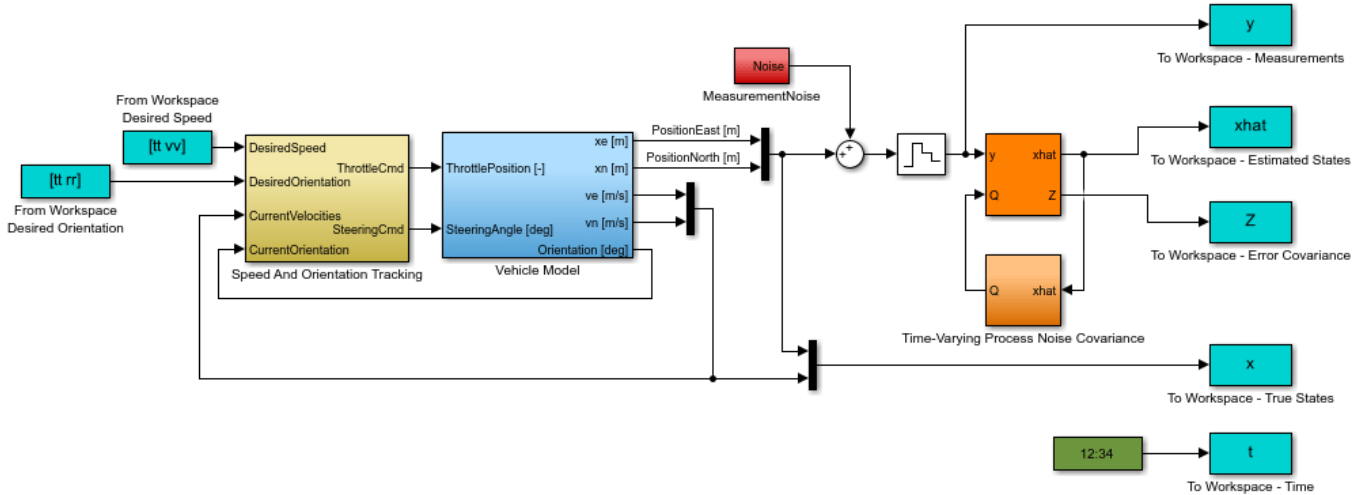
Copyright 2014 The MathWorks, Inc.

Vehicle Model

The tracked vehicle is represented with a simple point-mass model:

$$\frac{d}{dt} \begin{bmatrix} x_e(t) \\ x_n(t) \\ s(t) \\ \theta(t) \end{bmatrix} = \begin{bmatrix} s(t)\cos(\theta(t)) \\ s(t)\sin(\theta(t)) \\ (P\,\frac{u_T(t)}{s(t)} - A\,C_d\,s(t)^2)/m \\ s(t)\tan(u_\psi(t))/L \end{bmatrix}$$

where the vehicle states are:

$x_e(t)$ East position $[m]$
$x_n(t)$ North position $[m]$
$s(t)$ Speed $[m/s]$
$\theta(t)$ Orientation from east $[deg]$

the vehicle parameters are:

$P = 100000$ Peak engine power $[W]$
$A = 1$ Frontal area $[m^2]$
$C_d = 0.3$ Drag coefficient $[Unitless]$
$m = 1250$ Vehicle mass $[kg]$
$L = 2.5$ Wheelbase length $[m]$

and the control inputs are:

$u_T(t)$ Throttle position in the range of -1 and 1 $[Unitless]$
$u_\psi(t)$ Steering angle $[deg]$

The longitudinal dynamics of the model ignore tire rolling resistance. The lateral dynamics of the model assume that the desired steering angle can be achieved instantaneously and ignore the yaw moment of inertia.

16-89

The car model is implemented in the `ctrlKalmanNavigationExample/Vehicle Model` subsystem. The Simulink model contains two PI controllers for tracking the desired orientation and speed for the car in the `ctrlKalmanNavigationExample/Speed And Orientation Tracking` subsystem. This allows you to specify various operating conditions for the car and test the Kalman filter performance.

Kalman Filter Design

Kalman filter is an algorithm to estimate unknown variables of interest based on a linear model. This linear model describes the evolution of the estimated variables over time in response to model initial conditions as well as known and unknown model inputs. In this example, you estimate the following parameters/variables:

$$\hat{x}[n] = \begin{bmatrix} \hat{x}_e[n] \\ \hat{x}_n[n] \\ \hat{\dot{x}}_e[n] \\ \hat{\dot{x}}_n[n] \end{bmatrix}$$

where

$\hat{x}_e[n]$	East position estimate $[m]$
$\hat{x}_n[n]$	North position estimate $[m]$
$\hat{\dot{x}}_e[n]$	East velocity estimate $[m/s]$
$\hat{\dot{x}}_n[n]$	North velocity estimate $[m/s]$

The $\dot{x}$ terms denote velocities and not the derivative operator. n is the discrete-time index. The model used in the Kalman filter is of the form:

$$\begin{aligned} \hat{x}[n+1] &= A\hat{x}[n] + Gw[n] \\ y[n] &= C\hat{x}[n] + v[n] \end{aligned}$$

where $\hat{x}$ is the state vector, y is the measurements, w is the process noise, and v is the measurement noise. Kalman filter assumes that w and v are zero-mean, independent random variables with known variances $E[ww^T] = Q$, $E[vv^T] = R$, and $E[wv^T] = N$. Here, the A, G, and C matrices are:

$$A = \begin{bmatrix} 1 & 0 & T_s & 0 \\ 0 & 1 & 0 & T_s \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$G = \begin{bmatrix} T_s/2 & 0 \\ 0 & T_s/2 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

where $T_s = 1\ [s]$

The third row of A and G model the east velocity as a random walk: $\hat{\dot{x}}_e[n+1] = \hat{\dot{x}}_e[n] + w_1[n]$. In reality, position is a continuous-time variable and is the integral of velocity over time $\frac{d}{dt}\hat{x}_e = \hat{\dot{x}}_e$. The first row of the A and G represent a discrete approximation to this kinematic relationship: $(\hat{x}_e[n+1] - \hat{x}_e[n])/Ts = (\hat{\dot{x}}_e[n+1] + \hat{\dot{x}}_e[n])/2$. The second and fourth rows of the A and G represent the same relationship between the north velocity and position.

The C matrix represents that only position measurements are available. A position sensor, such as GPS, provides these measurements at the sample rate of 1Hz. The variance of the measurement noise v, the R matrix, is specified as $R = 50$. Since R is specified as a scalar, the Kalman filter block assumes that the matrix R is diagonal, its diagonals are 50 and is of compatible dimensions with y. If the measurement noise is Gaussian, R=50 corresponds to 68% of the position measurements being within $\pm\sqrt{50}\,m$ or the actual position in the east and north directions. However, this assumption is not necessary for the Kalman filter.

The elements of w capture how much the vehicle velocity can change over one sample time Ts. The variance of the process noise w, the Q matrix, is chosen to be time-varying. It captures the intuition that typical values of $w[n]$ are smaller when velocity is large. For instance, going from 0 to 10m/s is easier than going from 10 to 20m/s. Concretely, you use the estimated north and east velocities and a saturation function to construct Q[n]:

$$f_{sat}(z) = min(max(z, 25), 625)$$

$$Q[n] = \begin{bmatrix} 1 + \dfrac{250}{f_{sat}(\hat{\dot{x}}_e^2)} & 0 \\ 0 & 1 + \dfrac{250}{f_{sat}(\hat{\dot{x}}_n^2)} \end{bmatrix}$$

The diagonals of Q model the variance of w inversely proportional to the square of the estimated velocities. The saturation function prevents Q from becoming too large or small. The coefficient 250 is obtained from a least squares fit to 0-5, 5-10, 10-15, 15-20, 20-25m/s acceleration time data for a generic vehicle. Note that the diagonal Q choice represents a naive assumption that the velocity changes in the north and east direction are uncorrelated.

Kalman Filter Block Inputs and Setup

The 'Kalman Filter' block is in the `System Identification Toolbox/Estimators` library in Simulink. It is also in `Control System Toolbox` library. Configure the block parameters for discrete-time state estimation. Specify the following **Filter Settings** parameters:

- **Time domain:** Discrete-time. Choose this option to estimate discrete-time states.

- Select the **Use current measurement y[n] to improve xhat[n]** check box. This implements the "current estimator" variant of the discrete-time Kalman filter. This option improves the estimation accuracy and is more useful for slow sample times. However, it increases the computational cost. In addition, this Kalman filter variant has direct feedthrough, which leads to an algebraic loop if the Kalman filter is used in a feedback loop that does not contain any delays (the feedback loop itself also has direct feedthrough). The algebraic loop can further impact the simulation speed.

Click the **Options** tab to set the block inport and outport options:

- Unselect the **Add input port u** check box. There are no known inputs in the plant model.

- Select the **Output state estimation error covariance Z** check box. The Z matrix provides information about the filter's confidence in the state estimates.

Kalman Filter

Estimate the states of a discrete-time or continuous-time linear system. Time-varying systems are supported.

Filter Settings

Time domain: Discrete-Time ▾

☑ Use the current measurement y[n] to improve xhat[n]

| Model Parameters | Options |

Additional Inports

☐ Add input port u

☐ Add input port Enable to control measurement updates

External reset:

None ▾

Additional Outports

☐ Output estimated model output y

☑ Output state estimation error covariance Z

Sample time (-1 for inherited): Ts

| OK | Cancel | Help | Apply |

Click **Model Parameters** to specify the plant model and noise characteristics:

- **Model source:** Individual A, B, C, D matrices.

- **A**: A. The A matrix is defined earlier in this example.

- **C**: C. The C matrix is defined earlier in this example.

- **Initial Estimate Source**: Dialog

- **Initial states x[0]**: 0. This represents an initial guess of 0 for the position and velocity estimates at t=0s.

- **State estimation error covariance P[0]**: 10. Assume that the error between your initial guess x[0] and its actual value is a random variable with a standard deviation $\sqrt{10}$.

- Select the **Use G and H matrices (default G=I and H=0)** check box to specify a non-default G matrix.

- **G**: G. The G matrix is defined earlier in this example.

- **H**: 0. The process noise does not impact the measurements y entering the Kalman filter block.

- Unselect the **Time-invariant Q** check box. The Q matrix is time-varying and is supplied through the block inport Q. The block uses a time-varying Kalman filter due to this setting. You can select this option to use a time-invariant Kalman filter. A time-invariant Kalman filter performs slightly worse for this problem, but is easier to design and has a lower computational cost.

- **R**: R. This is the covariance of the measurement noise $v[n]$. The R matrix is defined earlier in this example.

- **N**: 0. Assume that there is no correlation between process and measurement noises.

- **Sample time (-1 for inherited)**: Ts, which is defined earlier in this example.

Kalman Filter

Estimate the states of a discrete-time or continuous-time linear system. Time-varying systems are supported.

Filter Settings

Time domain: Discrete-Time ▼

☑ Use the current measurement y[n] to improve xhat[n]

| Model Parameters | Options |

System Model

Model source: Individual A, B, C, D matrices ▼

A: [1 0 Ts 0; 0 1 0 Ts; 0 0 1 0; 0 0 0 1]

C: [1 0 0 0; 0 1 0 0]

Initial Estimates

Source: Dialog ▼

Initial states x[0]: 0

State estimation error covariance P[0] 10

Noise Characteristics

☑ Use G and H matrices (default G=I and H=0)

G: [Ts/2 0; 0 Ts/2; 1 0; 0 1]   ☑ Time-invariant G

H: 0   ☑ Time-invariant H

Q: 0.05   ☐ Time-invariant Q

R: 50   ☑ Time-invariant R

N: 0   ☑ Time-invariant N

Sample time (-1 for inherited): Ts

| OK | Cancel | Help | Apply |

Results

Test the performance of the Kalman filter by simulating a scenario where the vehicle makes the following maneuvers:

- At t = 0 the vehicle is at $x_e(0) = 0$, $x_n(0) = 0$ and is stationary.

- Heading east, it accelerates to 25m/s. It decelerates to 5m/s at t=50s.

- At t = 100s, it turns toward north and accelerates to 20m/s.

- At t = 200s, it makes another turn toward west. It accelerates to 25m/s.

- At t = 260s, it decelerates to 15m/s and makes a constant speed 180 degree turn.

Simulate the Simulink model. Plot the actual, measured and Kalman filter estimates of vehicle position.

```
sim('ctrlKalmanNavigationExample');

figure;
% Plot results and connect data points with a solid line.
plot(x(:,1),x(:,2),'bx',...
    y(:,1),y(:,2),'gd',...
    xhat(:,1),xhat(:,2),'ro',...
    'LineStyle','-');
title('Position');
xlabel('East [m]');
ylabel('North [m]');
legend('Actual','Measured','Kalman filter estimate','Location','Best');
axis tight;
```
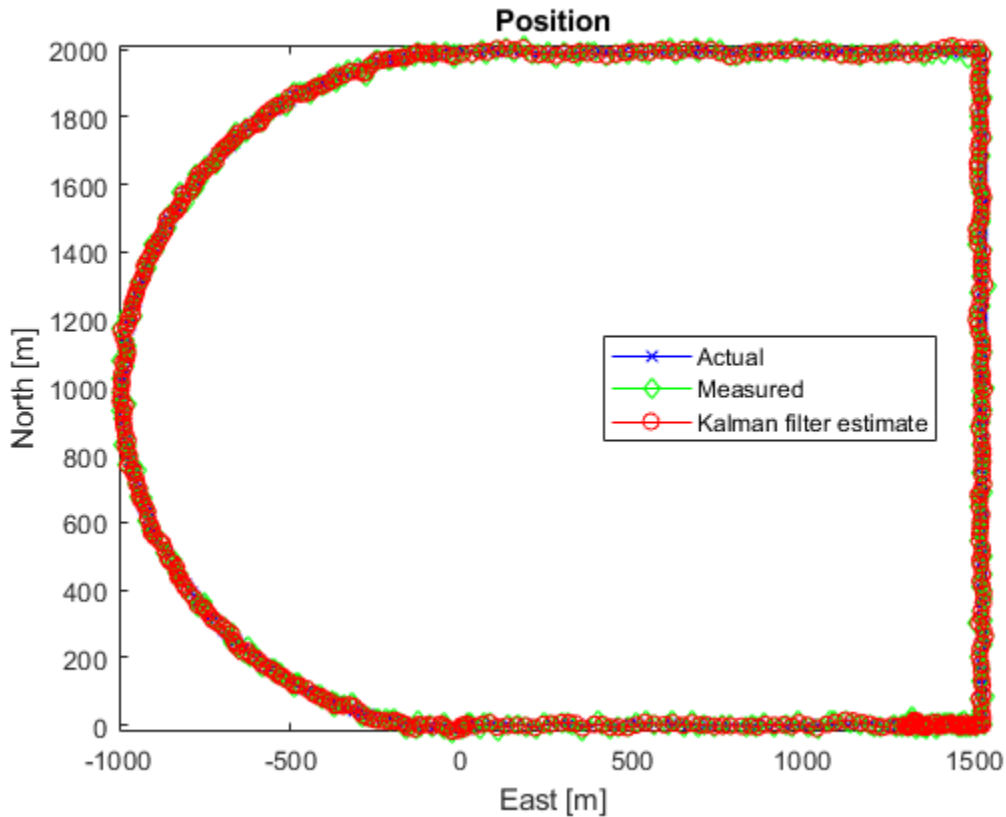
The error between the measured and actual position as well as the error between the Kalman filter estimate and actual position is:

```matlab
% East position measurement error [m]
n_xe = y(:,1)-x(:,1);
% North position measurement error [m]
n_xn = y(:,2)-x(:,2);
% Kalman filter east position error [m]
e_xe = xhat(:,1)-x(:,1);
% Kalman filter north position error [m]
e_xn = xhat(:,2)-x(:,2);

figure;
% East Position Errors
subplot(2,1,1);
plot(t,n_xe,'g',t,e_xe,'r');
ylabel('Position Error - East [m]');
xlabel('Time [s]');
legend(sprintf('Meas: %.3f',norm(n_xe,1)/numel(n_xe)),sprintf('Kalman f.: %.3f',norm(e_xe,1)/nume
axis tight;
% North Position Errors
subplot(2,1,2);
plot(t,y(:,2)-x(:,2),'g',t,xhat(:,2)-x(:,2),'r');
ylabel('Position Error - North [m]');
xlabel('Time [s]');
legend(sprintf('Meas: %.3f',norm(n_xn,1)/numel(n_xn)),sprintf('Kalman f: %.3f',norm(e_xn,1)/nume
axis tight;
```
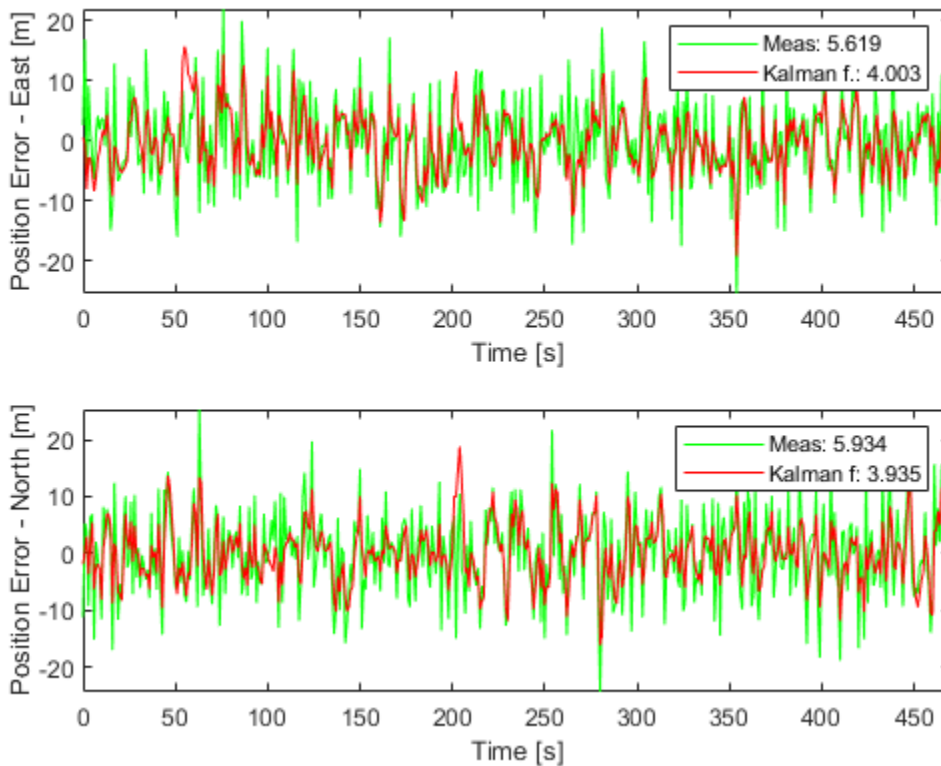
The plot legends show the position measurement and estimation error ($||x_e - \hat{x}_e||_1$ and $||x_n - \hat{x}_n||_1$) normalized by the number of data points. The Kalman filter estimates have about 25% percent less error than the raw measurements.

The actual velocity in the east direction and its Kalman filter estimate is shown below in the top plot. The bottom plot shows the estimation error.

```
e_ve = xhat(:,3)-x(:,3); % [m/s] Kalman filter east velocity error
e_vn = xhat(:,4)-x(:,4); % [m/s] Kalman filter north velocity error
figure;
% Velocity in east direction and its estimate
subplot(2,1,1);
plot(t,x(:,3),'b',t,xhat(:,3),'r');
ylabel('Velocity - East [m/s]');
xlabel('Time [s]');
legend('Actual','Kalman filter','Location','Best');
axis tight;
subplot(2,1,2);
% Estimation error
plot(t,e_ve,'r');
ylabel('Velocity Error - East [m/s]');
xlabel('Time [s]');
legend(sprintf('Kalman filter: %.3f',norm(e_ve,1)/numel(e_ve)));
axis tight;
```
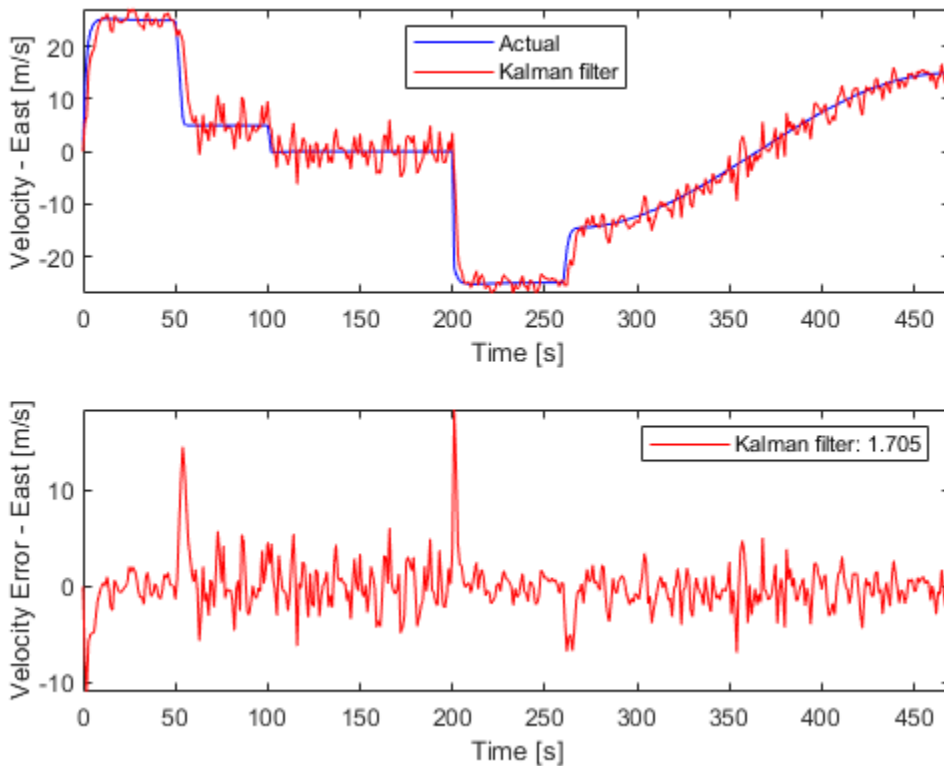
The legend on the error plot shows the east velocity estimation error $\|\dot{x}_e - \hat{\dot{x}}_e\|_1$ normalized by the number of data points.

The Kalman filter velocity estimates track the actual velocity trends correctly. The noise levels decrease when the vehicle is traveling at high velocities. This is in line with the design of the Q matrix. The large two spikes are at t=50s and t=200s. These are the times when the car goes through sudden deceleration and a sharp turn, respectively. The velocity changes at those instants are much larger than the predictions from the Kalman filter, which is based on its Q matrix input. After a few time-steps, the filter estimates catch up with the actual velocity.

Summary

You estimated the position and velocity of a vehicle using the Kalman filter block in Simulink. The process noise dynamics of the model were time-varying. You validated the filter performance by simulating various vehicle maneuvers and randomly generated measurement noise. The Kalman filter improved the position measurements and provided velocity estimates for the vehicle.

```
bdclose('ctrlKalmanNavigationExample');
```

Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter

This example shows how to use the unscented Kalman filter and particle filter algorithms for nonlinear state estimation for the van der Pol oscillator.

This example also uses the Signal Processing Toolbox™.

Introduction

System Identification Toolbox™ offers three commands for nonlinear state estimation:

- `extendedKalmanFilter`: First-order, discrete-time extended Kalman filter
- `unscentedKalmanFilter`: Discrete-time unscented Kalman filter
- `particleFilter`: Discrete-time particle filter

A typical workflow for using these commands is as follows:

1   Model your plant and sensor behavior.
2   Construct and configure the `extendedKalmanFilter`, `unscentedKalmanFilter` or `particleFilter` object.
3   Perform state estimation by using the `predict` and `correct` commands with the object.
4   Analyze results to gain confidence in filter performance
5   Deploy the filter on your hardware. You can generate code for these filters using MATLAB® Coder™.

This example first uses the `unscentedKalmanFilter` command to demonstrate this workflow. Then it demonstrates the use of `particleFilter`.

Plant Modeling and Discretization

The unscented Kalman filter (UKF) algorithm requires a function that describes the evolution of states from one time step to the next. This is typically called the state transition function. `unscentedKalmanFilter` supports the following two function forms:

1   Additive process noise: $x[k] = f(x[k-1], u[k-1]) + w[k-1]$
2   Non-additive process noise: $x[k] = f(x[k-1], w[k-1], u[k-1])$

Here $\mathbf{f}(..)$ is the state transition function, $\mathbf{x}$ is the state, w is the process noise. u is optional and represents additional inputs to $\mathbf{f}$, for instance system inputs or parameters. $\mathbf{u}$ can be specified as zero or more function arguments. Additive noise means that the state and process noise is related linearly. If the relationship is nonlinear, use the second form. When you create the `unscentedKalmanFilter` object, you specify $\mathbf{f}(..)$ and also whether the process noise is additive or non-additive.

The system in this example is the van der Pol oscillator with mu=1. This 2-state system is described with the following set of nonlinear ordinary differential equations (ODE):

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = (1 - x_1^2)x_2 - x_1$$

Denote this equation as $\dot{x} = f_c(x)$, where $x = [x_1; x_2]$. The process noise w does not appear in the system model. You can assume it is additive for simplicity.

`unscentedKalmanFilter` requires a discrete-time state transition function, but the plant model is continuous-time. You can use a discrete-time approximation to the continuous-time model. Euler discretization is one common approximation method. Assume that your sample time is T_s, and denote the continuous-time dynamics you have as $\dot{x} = f_c(x)$. Euler discretization approximates the derivative operator as $\dot{x} \approx \dfrac{x[k+1] - x[k]}{T_s}$. The resulting discrete-time state-transition function is:

$$x[k+1] = x[k] + f_c(x[k]) \, T_s$$
$$= f(x[k])$$

The accuracy of this approximation depends on the sample time T_s. Smaller T_s values provide better approximations. Alternatively, you can use a different discretization method. For instance, higher order Runge-Kutta family of methods provide a higher accuracy at the expense of more computational cost, given a fixed sample time T_s.

Create this state-transition function and save it in a file named vdpStateFcn.m. Use the sample time $T_s = 0.05s$. You provide this function to the `unscentedKalmanFilter` during object construction.

```
type vdpStateFcn
```

```
function x = vdpStateFcn(x)
% vdpStateFcn Discrete-time approximation to van der Pol ODEs for mu = 1.
% Sample time is 0.05s.
%
% Example state transition function for discrete-time nonlinear state
% estimators.
%
% xk1 = vdpStateFcn(xk)
%
% Inputs:
%    xk - States x[k]
%
% Outputs:
%    xk1 - Propagated states x[k+1]
%
% See also extendedKalmanFilter, unscentedKalmanFilter

%   Copyright 2016 The MathWorks, Inc.

%#codegen

% The tag %#codegen must be included if you wish to generate code with
% MATLAB Coder.

% Euler integration of continuous-time dynamics x'=f(x) with sample time dt
dt = 0.05; % [s] Sample time
x = x + vdpStateFcnContinuous(x)*dt;
end

function dxdt = vdpStateFcnContinuous(x)
%vdpStateFcnContinuous Evaluate the van der Pol ODEs for mu = 1
dxdt = [x(2); (1-x(1)^2)*x(2)-x(1)];
end
```

Sensor Modeling

`unscentedKalmanFilter` also needs a function that describes how the model states are related to sensor measurements. `unscentedKalmanFilter` supports the following two function forms:

1   Additive measurement noise: $y[k] = h(x[k], u[k]) + v[k]$
2   Non-additive measurement noise: $y[k] = h(x[k], v[k], u[k])$

h(..) is the measurement function, **v** is the measurement noise. **u** is optional and represents additional inputs to **h**, for instance system inputs or parameters. **u** can be specified as zero or more function arguments. You can add additional system inputs following the **u** term. These inputs can be different than the inputs in the state transition function.

For this example, assume you have measurements of the first state x_1 within some percentage error:

$$y[k] = x_1[k] \, (1 + v[k])$$

This falls into the category of non-additive measurement noise because the measurement noise is not simply added to a function of states. You want to estimate both x_1 and x_2 from the noisy measurements.

Create this state transition function and save it in a file named vdpMeasurementNonAdditiveNoiseFcn.m. You provide this function to the `unscented` during object construction.

`type vdpMeasurementNonAdditiveNoiseFcn`

```
function yk = vdpMeasurementNonAdditiveNoiseFcn(xk,vk)
% vdpMeasurementNonAdditiveNoiseFcn Example measurement function for discrete
% time nonlinear state estimators with non-additive measurement noise.
%
% yk = vdpNonAdditiveMeasurementFcn(xk,vk)
%
% Inputs:
%    xk - x[k], states at time k
%    vk - v[k], measurement noise at time k
%
% Outputs:
%    yk - y[k], measurements at time k
%
% The measurement is the first state with multiplicative noise
%
% See also extendedKalmanFilter, unscentedKalmanFilter

%   Copyright 2016 The MathWorks, Inc.

%#codegen

% The tag %#codegen must be included if you wish to generate code with
% MATLAB Coder.

yk = xk(1)*(1+vk);
end
```

Unscented Kalman Filter Construction

Construct the filter by providing function handles to the state transition and measurement functions, followed by your initial state guess. The state transition model has additive noise. This is the default

setting in the filter, hence you do not need to specify it. The measurement model has non-additive noise, which you must specify through setting the HasAdditiveMeasurementNoise property of the object as false. This must be done during object construction. If your application has non-additive process noise in the state transition function, specify the HasAdditiveProcessNoise property as false.

```
% Your initial state guess at time k, utilizing measurements up to time k-1: xhat[k|k-1]
initialStateGuess = [2;0]; % xhat[k|k-1]
% Construct the filter
ukf = unscentedKalmanFilter(...
    @vdpStateFcn,... % State transition function
    @vdpMeasurementNonAdditiveNoiseFcn,... % Measurement function
    initialStateGuess,...
    'HasAdditiveMeasurementNoise',false);
```

Provide your knowledge of the measurement noise covariance

```
R = 0.2; % Variance of the measurement noise v[k]
ukf.MeasurementNoise = R;
```

The ProcessNoise property stores the process noise covariance. It is set to account for model inaccuracies and the effect of unknown disturbances on the plant. We have the true model in this example, but discretization introduces errors. This example did not include any disturbances for simplicity. Set it to a diagonal matrix with less noise on the first state, and more on the second state to reflect the knowledge that the second state is more impacted by modeling errors.

```
ukf.ProcessNoise = diag([0.02 0.1]);
```

Estimation Using predict and correct Commands

In your application, the measurement data arriving from your hardware in real-time are processed by the filters as they arrive. This operation is demonstrated in this example by generating a set of measurement data first, and then feeding it to the filter one step at a time.

Simulate the van der Pol oscillator for 5 seconds with the filter sample time 0.05 [s] to generate the true states of the system.

```
T = 0.05; % [s] Filter sample time
timeVector = 0:T:5;
[~,xTrue]=ode45(@vdp1,timeVector,[2;0]);
```

Generate the measurements assuming that a sensor measures the first state, with a standard deviation of 45% error in each measurement.

```
rng(1); % Fix the random number generator for reproducible results
yTrue = xTrue(:,1);
yMeas = yTrue .* (1+sqrt(R)*randn(size(yTrue))); % sqrt(R): Standard deviation of noise
```

Pre-allocate space for data that you will analyze later

```
Nsteps = numel(yMeas); % Number of time steps
xCorrectedUKF = zeros(Nsteps,2); % Corrected state estimates
PCorrected = zeros(Nsteps,2,2); % Corrected state estimation error covariances
e = zeros(Nsteps,1); % Residuals (or innovations)
```

Perform online estimation of the states x using the correct and predict commands. Provide generated data to the filter one time step at a time.

```matlab
for k=1:Nsteps
    % Let k denote the current time.
    %
    % Residuals (or innovations): Measured output - Predicted output
    e(k) = yMeas(k) - vdpMeasurementFcn(ukf.State); % ukf.State is x[k|k-1] at this point
    % Incorporate the measurements at time k into the state estimates by
    % using the "correct" command. This updates the State and StateCovariance
    % properties of the filter to contain x[k|k] and P[k|k]. These values
    % are also produced as the output of the "correct" command.
    [xCorrectedUKF(k,:), PCorrected(k,:,:)] = correct(ukf,yMeas(k));
    % Predict the states at next time step, k+1. This updates the State and
    % StateCovariance properties of the filter to contain x[k+1|k] and
    % P[k+1|k]. These will be utilized by the filter at the next time step.
    predict(ukf);
end
```

Unscented Kalman Filter Results and Validation

Plot the true and estimated states over time. Also plot the measured value of the first state.

```matlab
figure();
subplot(2,1,1);
plot(timeVector,xTrue(:,1),timeVector,xCorrectedUKF(:,1),timeVector,yMeas(:));
legend('True','UKF estimate','Measured')
ylim([-2.6 2.6]);
ylabel('x_1');
subplot(2,1,2);
plot(timeVector,xTrue(:,2),timeVector,xCorrectedUKF(:,2));
ylim([-3 1.5]);
xlabel('Time [s]');
ylabel('x_2');
```
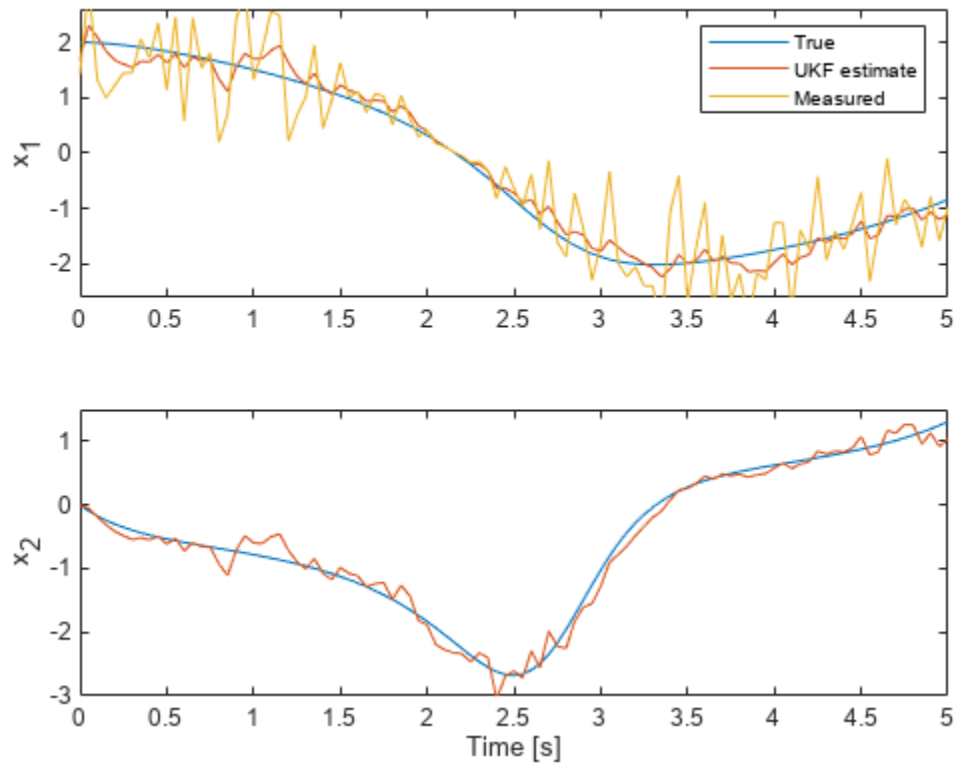
The top plot shows the true, estimated, and the measured value of the first state. The filter utilizes the system model and noise covariance information to produce an improved estimate over the measurements. The bottom plot shows the second state. The filter is successful in producing a good estimate.

The validation of unscented and extended Kalman filter performance is typically done using extensive Monte Carlo simulations. These simulations should test variations of process and measurement noise realizations, plant operating under various conditions, initial state and state covariance guesses. The key signal of interest used for validating the state estimation is the residuals (or innovations). In this example, you perform residual analysis for a single simulation. Plot the residuals.

```
figure();
plot(timeVector, e);
xlabel('Time [s]');
ylabel('Residual (or innovation)');
```
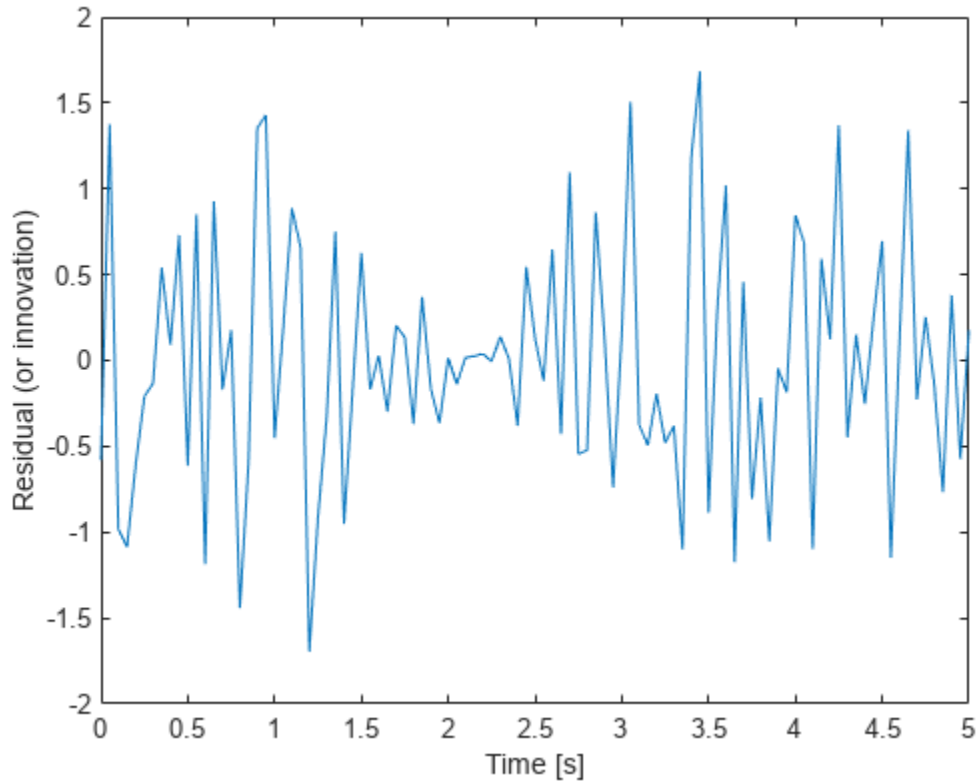
The residuals should have:

1 Small magnitude

2 Zero mean

3 No autocorrelation, except at zero lag

The mean value of the residuals is:

```
mean(e)
```

```
ans = -0.0012
```

This is small relative to the magnitude of the residuals. The autocorrelation of the residuals can be calculated with the xcorr function in the Signal Processing Toolbox.

```
[xe,xeLags] = xcorr(e,'coeff'); % 'coeff': normalize by the value at zero lag
% Only plot non-negative lags
idx = xeLags>=0;
figure();
plot(xeLags(idx),xe(idx));
xlabel('Lags');
ylabel('Normalized correlation');
title('Autocorrelation of residuals (innovation)');
```
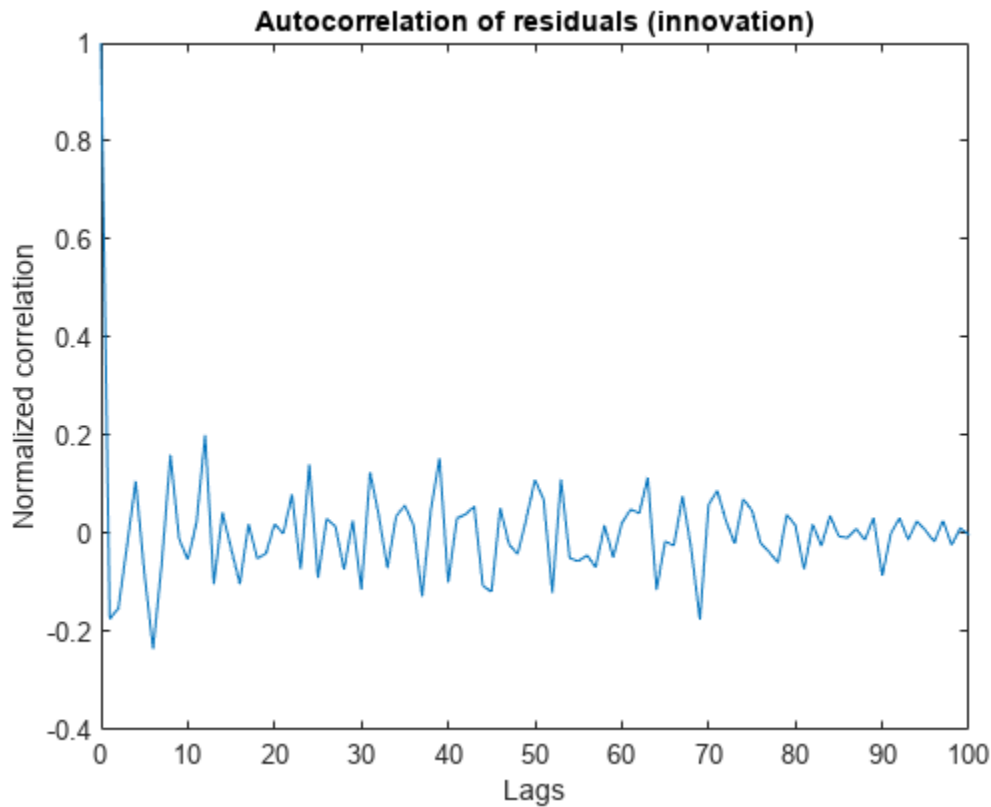
The correlation is small for all lags except 0. The mean correlation is close to zero, and the correlation does not show any significant non-random variations. These characteristics increase confidence in filter performance.

In reality the true states are never available. However, when performing simulations, you have access to real states and can look at the errors between estimated and true states. These errors must satisfy similar criteria to the residual:

1   Small magnitude

2   Variance within filter error covariance estimate

3   Zero mean

4   Uncorrelated.

First, look at the error and the 1σ uncertainty bounds from the filter error covariance estimate.

```
eStates = xTrue-xCorrectedUKF;
figure();
subplot(2,1,1);
plot(timeVector,eStates(:,1),...                    % Error for the first state
    timeVector, sqrt(PCorrected(:,1,1)),'r', ... % 1-sigma upper-bound
    timeVector, -sqrt(PCorrected(:,1,1)),'r');   % 1-sigma lower-bound
xlabel('Time [s]');
ylabel('Error for state 1');
title('State estimation errors');
subplot(2,1,2);
plot(timeVector,eStates(:,2),...                    % Error for the second state
```

```
    timeVector,sqrt(PCorrected(:,2,2)),'r', ...    % 1-sigma upper-bound
    timeVector,-sqrt(PCorrected(:,2,2)),'r');      % 1-sigma lower-bound
xlabel('Time [s]');
ylabel('Error for state 2');
legend('State estimate','1-sigma uncertainty bound',...
    'Location','Best');
```
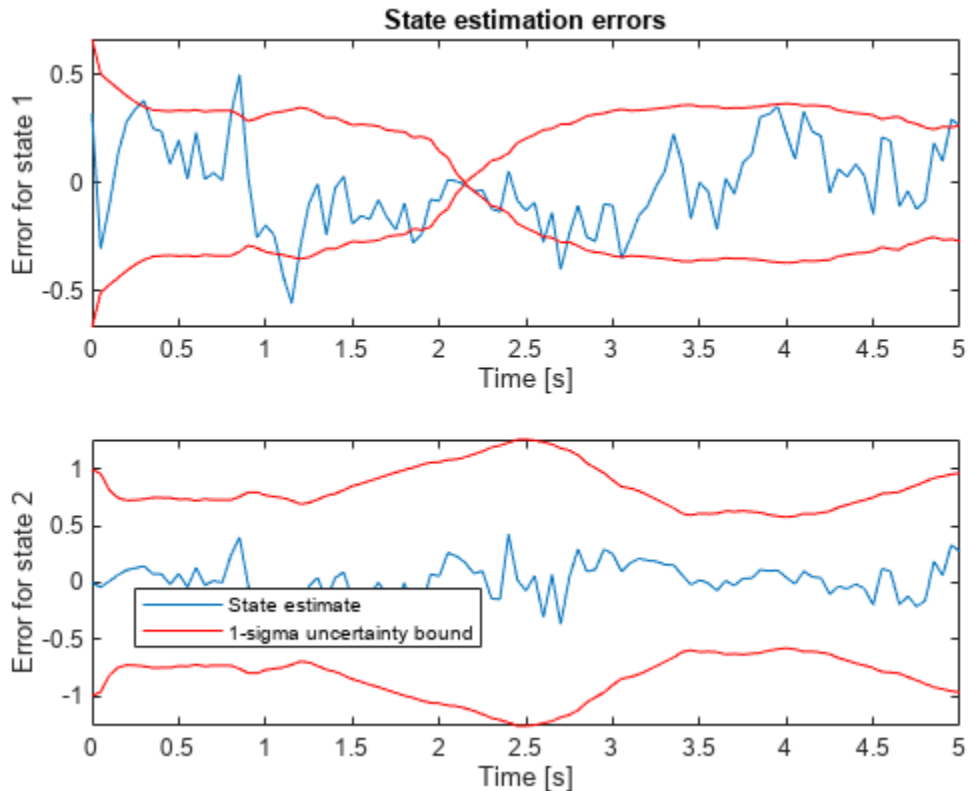

The error bound for state 1 approaches 0 at t=2.15 seconds because of the sensor model (MeasurementFcn). It has the form $x_1[k]*(1 + v[k])$. At t=2.15 seconds the true and estimated states are near zero, which implies the measurement error in absolute terms is also near zero. This is reflected in the state estimation error covariance of the filter.

Calculate what percentage of the points are beyond the 1-sigma uncertainty bound.

```
distanceFromBound1 = abs(eStates(:,1))-sqrt(PCorrected(:,1,1));
percentageExceeded1 = nnz(distanceFromBound1>0) / numel(eStates(:,1));
distanceFromBound2 = abs(eStates(:,2))-sqrt(PCorrected(:,2,2));
percentageExceeded2 = nnz(distanceFromBound2>0) / numel(eStates(:,2));
[percentageExceeded1 percentageExceeded2]
```

ans = *1×2*

    0.1386   0

The first state estimation errors exceed the 1σ uncertainty bound approximately 14% of the time steps. Less than 30% of the errors exceeding the 1-sigma uncertainty bound implies good estimation.

This criterion is satisfied for both states. The 0% percentage for the second state means that the filter is conservative: most likely the combined process and measurement noises are too high. Likely a better performance can be obtained by tuning these parameters.
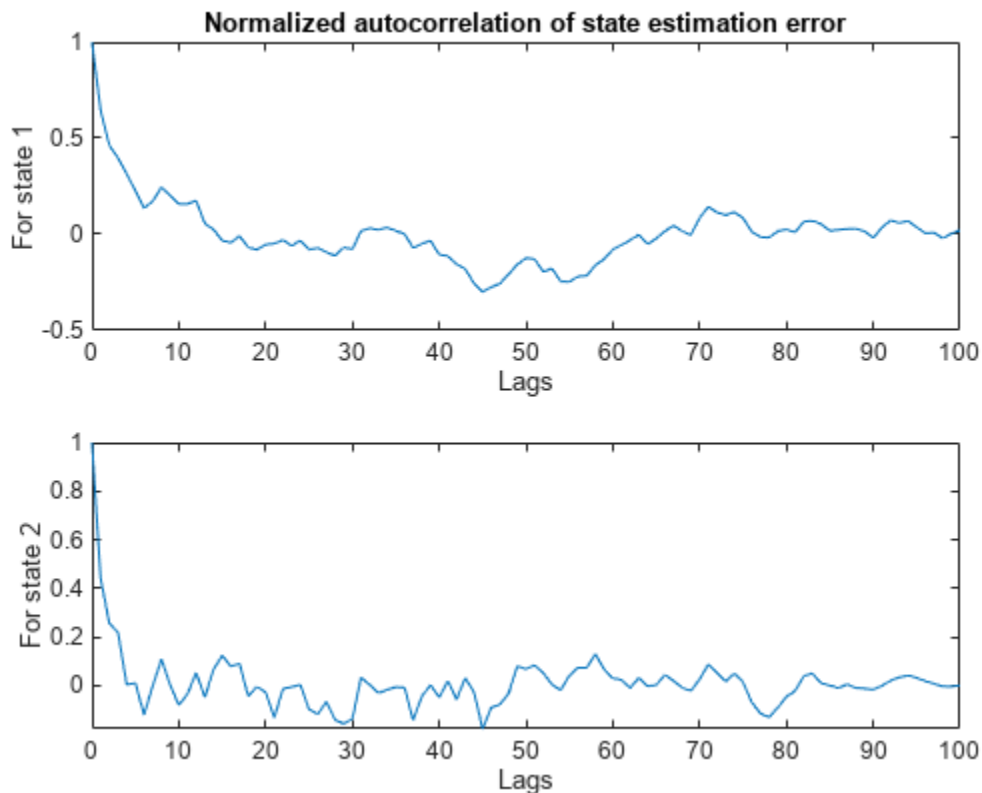
Calculate the mean autocorrelation of state estimation errors. Also plot the autocorrelation.

```
mean(eStates)
```

```
ans = 1×2

   -0.0103    0.0201
```

```
[xeStates1,xeStatesLags1] = xcorr(eStates(:,1),'coeff'); % 'coeff': normalize by the value at ze
[xeStates2,xeStatesLags2] = xcorr(eStates(:,2),'coeff'); % 'coeff'
% Only plot non-negative lags
idx = xeStatesLags1>=0;
figure();
subplot(2,1,1);
plot(xeStatesLags1(idx),xeStates1(idx));
xlabel('Lags');
ylabel('For state 1');
title('Normalized autocorrelation of state estimation error');
subplot(2,1,2);
plot(xeStatesLags2(idx),xeStates2(idx));
xlabel('Lags');
ylabel('For state 2');
```

The mean value of the errors is small relative to the value of the states. The autocorrelation of state estimation errors shows little non-random variations for small lag values, but these are much smaller than the normalized peak value 1. Combined with the fact that the estimated states are accurate, this behavior of the residuals can be considered as satisfactory results.

Particle Filter Construction

Unscented and extended Kalman filters aim to track the mean and covariance of the posterior distribution of the state estimates by different approximation methods. These methods may not be sufficient if the nonlinearities in the system are severe. In addition, for some applications, just tracking the mean and covariance of the posterior distribution of the state estimates may not be sufficient. Particle filters can address these problems by tracking the evolution of many state hypotheses (particles) over time, at the expense of higher computational cost. The computational cost and estimation accuracy increases with the number of particles.

The `particleFilter` command in System Identification Toolbox implements a discrete-time particle filter algorithm. This section walks you through constructing a particle filter for the same van der Pol oscillator used earlier in this example, and highlights the similarities and differences with the unscented Kalman filter.

The state transition function you provide to `particleFilter` must perform two tasks. One, sampling the process noise from any distribution appropriate for your system. Two, calculating the time propagation of all particles (state hypotheses) to the next step, including the effect of process noise you calculated in step one.

type vdpParticleFilterStateFcn

```
function particles = vdpParticleFilterStateFcn(particles)
% vdpParticleFilterStateFcn Example state transition function for particle
%                           filter
%
% Discrete-time approximation to van der Pol ODEs for mu = 1.
% Sample time is 0.05s.
%
% predictedParticles = vdpParticleFilterStateFcn(particles)
%
% Inputs:
%    particles - Particles at current time. Matrix with dimensions
%                [NumberOfStates NumberOfParticles] matrix
%
% Outputs:
%    predictedParticles - Predicted particles for the next time step
%
% See also particleFilter

%   Copyright 2017 The MathWorks, Inc.

%#codegen

% The tag %#codegen must be included if you wish to generate code with
% MATLAB Coder.

[numberOfStates, numberOfParticles] = size(particles);

% Time-propagate each particle
%
% Euler integration of continuous-time dynamics x'=f(x) with sample time dt
```

```
dt = 0.05; % [s] Sample time
for kk=1:numberOfParticles
    particles(:,kk) = particles(:,kk) + vdpStateFcnContinuous(particles(:,kk))*dt;
end

% Add Gaussian noise with variance 0.025 on each state variable
processNoise = 0.025*eye(numberOfStates);
particles = particles + processNoise * randn(size(particles));
end

function dxdt = vdpStateFcnContinuous(x)
%vdpStateFcnContinuous Evaluate the van der Pol ODEs for mu = 1
dxdt = [x(2); (1-x(1)^2)*x(2)-x(1)];
end
```

There are differences between the state transition function you supply to `unscentedKalmanFilter` and `particleFilter`. The state transition function you used for unscented Kalman filter just described propagation of one state hypothesis to the next time step, instead of a set of hypotheses. In addition, the process noise distribution was defined in `ProcessNoise` property of the `unscentedKalmanFilter`, just by its covariance. Particle filter can consider arbitrary distributions that may require more statistical properties to be defined. This arbitrary distribution and its parameters are fully defined in the state transition function you provide to the `particleFilter`.

The measurement likelihood function you provide to `particleFilter` must also perform two tasks. One, calculating measurement hypotheses from particles. Two, calculating the likelihood of each particle from the sensor measurement and the hypotheses calculated in step one.

type vdpExamplePFMeasurementLikelihoodFcn

```
function likelihood = vdpExamplePFMeasurementLikelihoodFcn(particles,measurement)
% vdpExamplePFMeasurementLikelihoodFcn Example measurement likelihood function
%
% The measurement is the first state.
%
% likelihood = vdpParticleFilterMeasurementLikelihoodFcn(particles, measurement)
%
% Inputs:
%    particles - NumberOfStates-by-NumberOfParticles matrix that holds
%                the particles
%
% Outputs:
%    likelihood - A vector with NumberOfParticles elements whose n-th
%                 element is the likelihood of the n-th particle
%
% See also extendedKalmanFilter, unscentedKalmanFilter

%   Copyright 2017 The MathWorks, Inc.

%#codegen

% The tag %#codegen must be included if you wish to generate code with
% MATLAB Coder.

% Validate the sensor measurement
numberOfMeasurements = 1; % Expected number of measurements
validateattributes(measurement, {'double'}, {'vector', 'numel', numberOfMeasurements}, ...
    'vdpExamplePFMeasurementLikelihoodFcn', 'measurement');
```

```
% The measurement is first state. Get all measurement hypotheses from particles
predictedMeasurement = particles(1,:);

% Assume the ratio of the error between predicted and actual measurements
% follow a Gaussian distribution with zero mean, variance 0.2
mu = 0; % mean
sigma = 0.2 * eye(numberOfMeasurements); % variance

% Use multivariate Gaussian probability density function, calculate
% likelihood of each particle
numParticles = size(particles,2);
likelihood = zeros(numParticles,1);
C = det(2*pi*sigma) ^ (-0.5);
for kk=1:numParticles
    errorRatio = (predictedMeasurement(kk)-measurement)/predictedMeasurement(kk);
    v = errorRatio-mu;
    likelihood(kk) = C * exp(-0.5 * (v' / sigma * v) );
end
end
```

Now construct the filter, and initialize it with 1000 particles around the mean [2; 0] with 0.01 covariance. The covariance is small because you have high confidence in your guess [2; 0].

```
pf = particleFilter(@vdpParticleFilterStateFcn,@vdpExamplePFMeasurementLikelihoodFcn);
initialize(pf, 1000, [2;0], 0.01*eye(2));
```

Optionally, pick the state estimation method. This is set by the `StateEstimationMethod` property of `particleFilter`, which can take the value `'mean'` (default) or `'maxweight'`. When `StateEstimationMethod` is `'mean'`, the object extracts a weighted mean of the particles from the `Particles` and `Weights` properties as the state estimate. `'maxweight'` corresponds to choosing the particle (state hypothesis) with the maximum weight value in `Weights` as the state estimate. Alternatively, you can access `Particles` and `Weights` properties of the object and extract your state estimate via an arbitrary method of your choice.

```
pf.StateEstimationMethod
```

```
ans =
'mean'
```

`particleFilter` lets you specify various resampling options via its `ResamplingPolicy` and `ResamplingMethod` properties. This example uses the default settings in the filter. See the `particleFilter` documentation for further details on resampling.

```
pf.ResamplingMethod
```

```
ans =
'multinomial'
```

```
pf.ResamplingPolicy
```

```
ans =
  particleResamplingPolicy with properties:

               TriggerMethod: 'ratio'
            SamplingInterval: 1
    MinEffectiveParticleRatio: 0.5000
```

Start the estimation loop. This represents measurements arriving over time, step by step.

```
% Estimate
xCorrectedPF = zeros(size(xTrue));
for k=1:size(xTrue,1)
    % Use measurement y[k] to correct the particles for time k
    xCorrectedPF(k,:) = correct(pf,yMeas(k)); % Filter updates and stores Particles[k|k], Weights
    % The result is x[k|k]: Estimate of states at time k, utilizing
    % measurements up to time k. This estimate is the mean of all particles
    % because StateEstimationMethod was 'mean'.
    %
    % Now, predict particles at next time step. These are utilized in the
    % next correct command
    predict(pf); % Filter updates and stores Particles[k+1|k]
end
```

Plot the state estimates from particle filter:

```
figure();
subplot(2,1,1);
plot(timeVector,xTrue(:,1),timeVector,xCorrectedPF(:,1),timeVector,yMeas(:));
legend('True','Particlte filter estimate','Measured')
ylim([-2.6 2.6]);
ylabel('x_1');
subplot(2,1,2);
plot(timeVector,xTrue(:,2),timeVector,xCorrectedPF(:,2));
ylim([-3 1.5]);
xlabel('Time [s]');
ylabel('x_2');
```
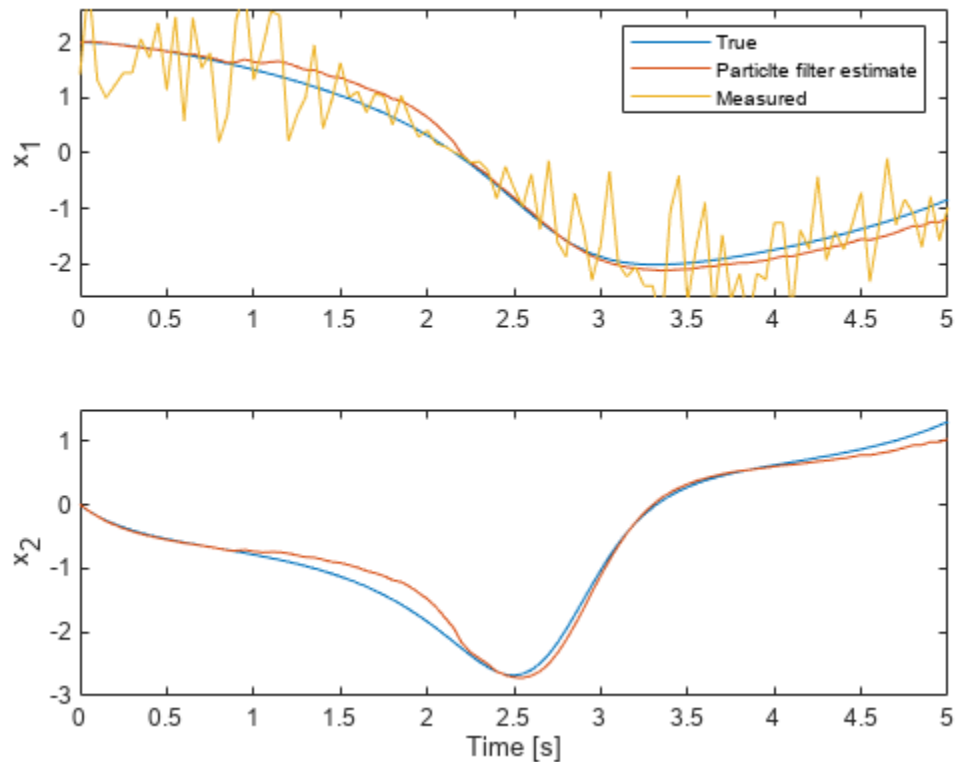
The top plot shows the true value, particle filter estimate, and the measured value of the first state. The filter utilizes the system model and noise information to produce an improved estimate over the measurements. The bottom plot shows the second state. The filter is successful in producing a good estimate.

The validation of the particle filter performance involves performing statistical tests on residuals, similar to those that were performed earlier in this example for unscented Kalman filter results.

Summary

This example has shown the steps of constructing and using an unscented Kalman filter and a particle filter for state estimation of a nonlinear system. You estimated states of a van der Pol oscillator from noisy measurements, and validated the estimation performance.

See Also

extendedKalmanFilter | unscentedKalmanFilter | particleFilter

More About

- "Extended and Unscented Kalman Filter Algorithms for Online State Estimation" on page 16-27
- "Validate Online State Estimation at the Command Line" on page 16-34
- "Troubleshoot Online State Estimation" on page 16-42
- "Generate Code for Online State Estimation in MATLAB" on page 16-39

External Websites

- Understanding Kalman Filters: Nonlinear State Estimators — MATLAB Video Series

Estimate States of Nonlinear System with Multiple, Multirate Sensors

This example shows how to perform nonlinear state estimation in Simulink™ for a system with multiple sensors operating at different sample rates. The Extended Kalman Filter block in System Identification Toolbox™ is used to estimate the position and velocity of an object using GPS and radar measurements.

Introduction

The toolbox has three Simulink blocks for nonlinear state estimation:

- Extended Kalman Filter: Implements the first-order discrete-time extended Kalman filter algorithm.
- Unscented Kalman Filter: Implements the discrete-time unscented Kalman filter algorithm.
- Particle Filter: Implements a discrete-time particle filter algorithm.

These blocks support state estimation using multiple sensors operating at different sample rates. A typical workflow for using these blocks is as follows:

1. Model your plant and sensor behavior using MATLAB or Simulink functions.
2. Configure the parameters of the block.
3. Simulate the filter and analyze results to gain confidence in filter performance.
4. Deploy the filter on your hardware. You can generate code for these filters using Simulink Coder™ software.

This example uses the Extended Kalman Filter block to demonstrate the first two steps of this workflow. The last two steps are briefly discussed in the **Next Steps** section. The goal in this example is to estimate the states of an object using noisy measurements provided by a radar and a GPS sensor. The states of the object are its position and velocity, which are denoted as xTrue in the Simulink model.

If you are interested in the Particle Filter block, please see the example "Parameter and State Estimation in Simulink Using Particle Filter Block".

```
addpath(fullfile(matlabroot,'examples','ident','main')) % add example data
open_system('multirateEKFExample');
```
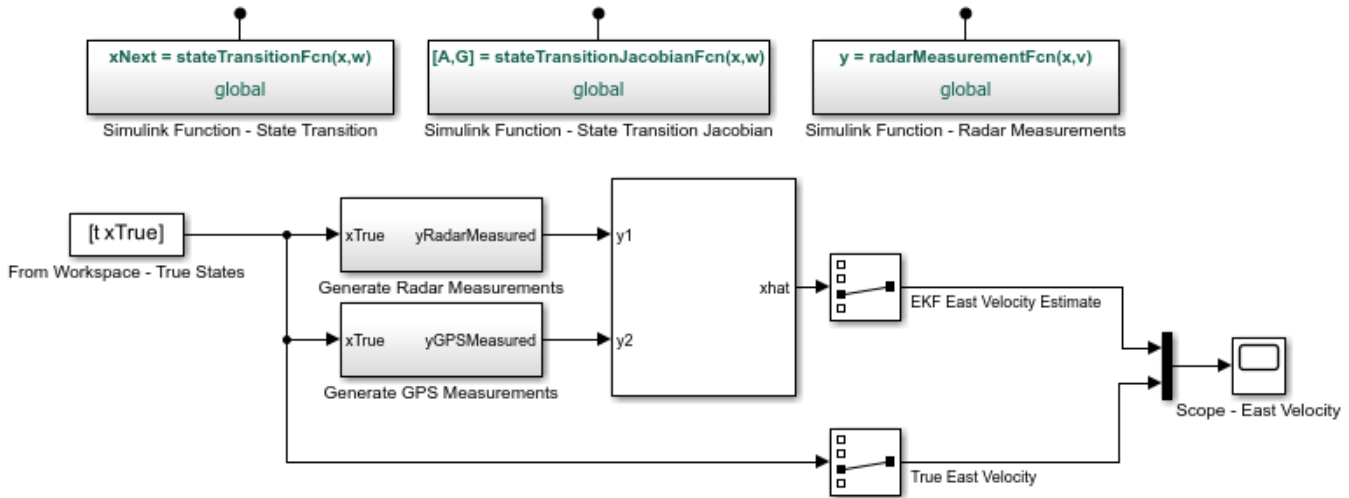
Copyright 2016-2017 The MathWorks, Inc.

Plant Modeling

The extended Kalman filter (EKF) algorithm requires a state transition function that describes the evolution of states from one time step to the next. The block supports the following two function forms:

- Additive process noise: $x[k+1] = f(x[k], u[k]) + w[k]$

- Nonadditive process noise: $x[k+1] = f(x[k], w[k], u[k])$

Here f(..) is the state transition function, x is the state, and w is the process noise. u is optional, and represents additional inputs to f, for instance system inputs or parameters. Additive noise means that the next state $x[k+1]$ and process noise $w[k]$ are related linearly. If the relationship is nonlinear, use the nonadditive form.

The function f(...) can be a MATLAB Function that comply with the restrictions of MATLAB Coder™, or a Simulink Function block. After you create f(...), you specify the function name and whether the process noise is additive or nonadditive in the Extended Kalman Filter block.

In this example, you are tracking the north and east positions and velocities of an object on a 2-dimensional plane. The estimated quantities are:

$$\hat{x}[k] = \begin{bmatrix} \hat{x}_e[k] \\ \hat{x}_n[k] \\ \hat{v}_e[k] \\ \hat{v}_n[k] \end{bmatrix} \quad \begin{matrix} \text{East position estimate } [m] \\ \text{North position estimate } [m] \\ \text{East velocity estimate } [m/s] \\ \text{North velocity estimate } [m/s] \end{matrix}$$

Here k is the discrete-time index. The state transition equation used is of the nonadditive form $\hat{x}[k+1] = A\hat{x}[k] + Gw[k]$, where $\hat{x}$ is the state vector, and w is the process noise. The filter assumes that w is a zero-mean, independent random variable with known variance $E[ww^T]$. The A and G matrices are:

$$A = \begin{bmatrix} 1 & 0 & T_s & 0 \\ 0 & 1 & 0 & T_s \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad G = \begin{bmatrix} T_s/2 & 0 \\ 0 & T_s/2 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

where T_s is the sample time. The third row of A and G model the east velocity as a random walk: $\hat{v}_e[k+1] = \hat{v}_e[k] + w_1[k]$. In reality, position is a continuous-time variable and is the integral of velocity over time $\frac{d}{dt}\hat{x}_e = \hat{v}_e$. The first row of A and G represent a discrete approximation to this kinematic relationship: $(\hat{x}_e[k+1] - \hat{x}_e[k])/T_s = (\hat{v}_e[k+1] + \hat{v}_e[k])/2$. The second and fourth rows of A and G represent the same relationship between the north velocity and position. This state transition model is linear, but the radar measurement model is nonlinear. This nonlinearity necessitates the use of a nonlinear state estimator such as the extended Kalman filter.

In this example you implement the state transition function using a Simulink Function block. To do so,

- Add a `Simulink Function` block to your model from the `Simulink/User-Defined Functions` library

- Click on the name shown on the Simulink Function block. Edit the function name, and add or remove input and output arguments, as necessary. In this example the name for the state transition function is `stateTransitionFcn`. It has one output argument (xNext) and two input arguments (x, w).

Simulink Function - State Transition
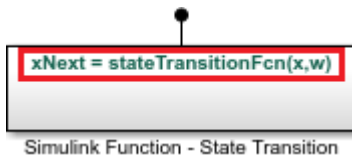
- Though it is not required in this example, you can use any signals from the rest of your Simulink model in the Simulink Function. To do so, add `Inport` blocks from the `Simulink/Sources` library. Note that these are different than the `ArgIn` and `ArgOut` blocks that are set through the signature of your function (xNext = stateTransitionFcn(x, w)).

- In the Simulink Function block, construct your function utilizing Simulink blocks.

- Set the dimensions for the input and output arguments x, w, and xNext in the **Signal Attributes** tab of the `ArgIn` and `ArgOut` blocks. The data type and port dimensions must be consistent with the information you provide in the `Extended Kalman Filter` block.
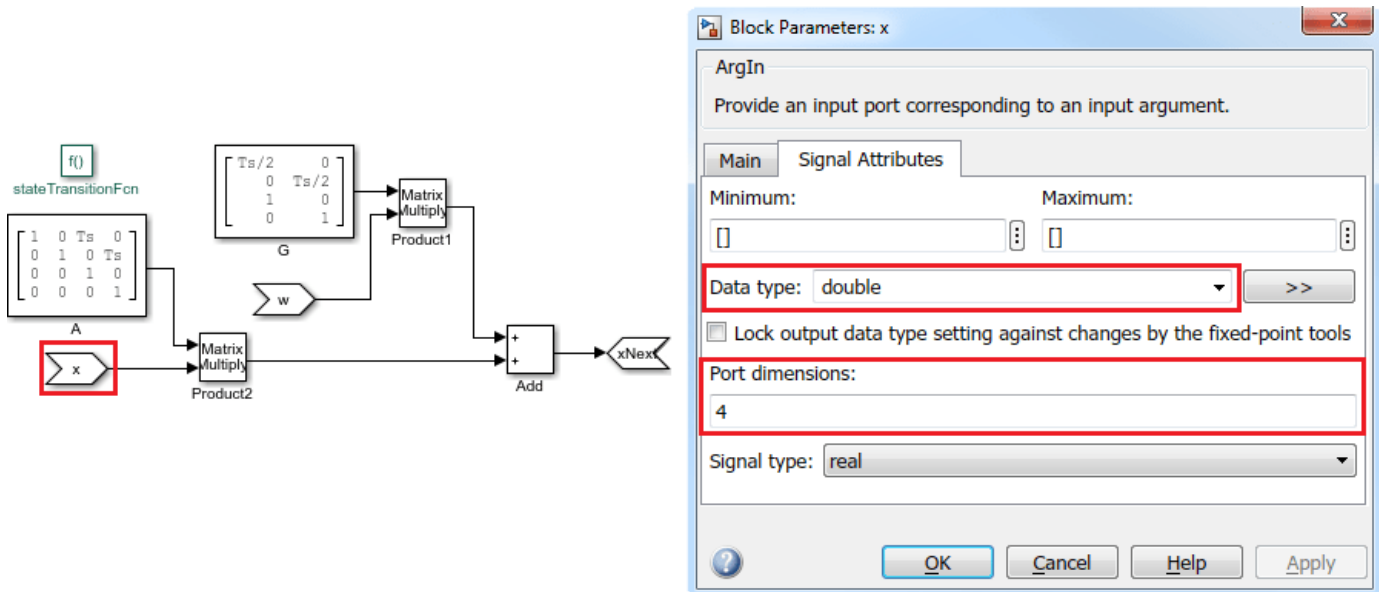
Analytical Jacobian of the state transition function is also implemented in this example. Specifying the Jacobian is optional. However, this reduces the computational burden, and in most cases increases the state estimation accuracy. Implement the Jacobian function as a Simulink function because the state transition function is a Simulink function.

```
open_system('multirateEKFExample/Simulink Function - State Transition Jacobian');
```
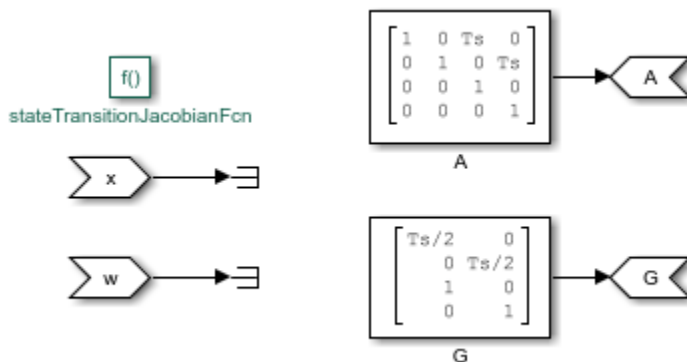

Sensor modeling - Radar

The Extended Kalman Filter block also needs a measurement function that describes how the states are related to measurements. The following two function forms are supported:

- Additive measurement noise: $y[k] = h(x[k], u[k]) + v[k]$

- Nonadditive measurement noise: $y[k] = h(x[k], v[k], u[k])$

Here h(..) is the measurement function, and v is the measurement noise. u is optional, and represents additional inputs to h, for instance system inputs or parameters. These inputs can differ from the inputs in the state transition function.

In this example a radar located at the origin measures the range and angle of the object at 20 Hz. Assume that both of the measurements have about 5% noise. This can be modeled by the following measurement equation:

$$y_{radar}[k] = \begin{bmatrix} \sqrt{x_n[k]^2 + x_e[k]^2}\,(1 + v_1[k]) \\ atan2(x_n[k], x_e[k])\,(1 + v_2[k]) \end{bmatrix}$$

Here $v_1[k]$ and $v_2[k]$ are the measurement noise terms, each with variance 0.05^2. That is, most of the measurements have errors less than 5%. The measurement noise is nonadditive because $v_1[k]$ and $v_2[k]$ are not simply added to the measurements, but instead they depend on the states x. In this example, the radar measurement equation is implemented using a Simulink Function block.

```
open_system('multirateEKFExample/Simulink Function - Radar Measurements');
```
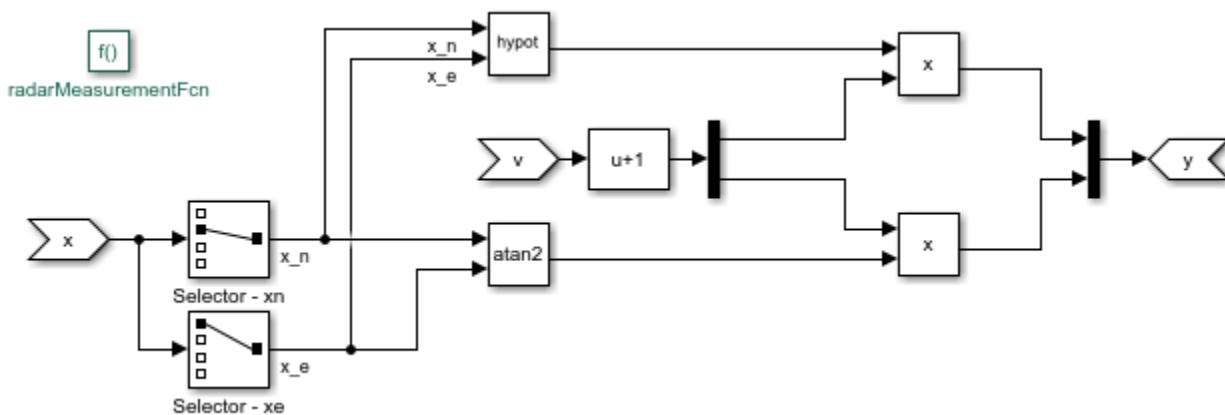

Sensor modeling - GPS

A GPS measures the east and north positions of the object at 1 Hz. Hence, the measurement equation for the GPS sensor is:

$$y_{GPS}[k] = \begin{bmatrix} x_e[k] \\ x_n[k] \end{bmatrix} + \begin{bmatrix} v_1[k] \\ v_2[k] \end{bmatrix}$$

Here $v_1[k]$ and $v_2[k]$ are measurement noise terms with the covariance matrix [10^2 0; 0 10^2]. That is, the measurements are accurate up to approximately 10 meters, and the errors are uncorrelated. The measurement noise is additive because the noise terms affect the measurements y_{GPS} linearly.

Create this function, and save it in a file named gpsMeasurementFcn.m. When the measurement noise is additive, you must not specify the noise terms in the function. You provide this function name and measurement noise covariance in the Extended Kalman Filter block.

```
type gpsMeasurementFcn
```

```
function y = gpsMeasurementFcn(x)
% gpsMeasurementFcn GPS measurement function for state estimation
%
% Assume the states x are:
```

```
%   [EastPosition; NorthPosition; EastVelocity; NorthVelocity]

%#codegen

% The %#codegen tag above is needed is you would like to use MATLAB Coder to
% generate C or C++ code for your filter

y = x([1 2]); % Position states are measured
end
```

Filter Construction

Configure the Extended Kalman Filter block to perform the estimation. You specify the state transition and measurement function names, initial state and state error covariance, and process and measurement noise characteristics.

In the **System Model** tab of the block dialog, specify the following parameters:

State Transition

1   Specify the state transition function, `stateTransitionFcn`, in **Function**. Since you have the Jacobian of this function, select **Jacobian**, and specify the Jacobian function, `stateTransitionJacobianFcn`.
2   Select `Nonadditive` in the **Process Noise** drop-down list because you explicitly stated how the process noise impacts the states in your function.
3   Specify the process noise covariance as [0.2 0; 0 0.2]. As explained in the **Plant Modeling** section of this example, process noise terms define the random walk of the velocities in each direction. The diagonal terms approximately capture how much the velocities can change over one sample time of the state transition function. The off-diagonal terms are set to zero, which is a naive assumption that velocity variations in the north and east directions are uncorrelated.

Initialization

1   Specify your best initial state estimate in **Initial state**. In this example, specify [100; 100; 0; 0].
2   Specify your confidence in your state estimate guess in **Initial covariance**. In this example, specify 10. The software interprets this value as the true state values are likely to be within $\pm\sqrt{10}$ of your initial estimate. You can specify a separate value for each state by setting `Initial covariance` as a vector. You can specify cross-correlations in this uncertainty by specifying it as a matrix.

Since there are two sensors, click **Add Measurement** to specify a second measurement function.

Measurement 1

1   Specify the name of your measurement function, `radarMeasurementFcn`, in **Function**.
2   Select `Nonadditive` in the **Measurement Noise** drop-down list because you explicitly stated how the process noise impacts the measurements in your function.
3   Specify the measurement noise covariance as [0.05^2 0; 0 0.05^2] per the discussion in the **Sensor Modeling - Radar** section.

Measurement 2

1   Specify the name of your measurement function, `gpsMeasurementFcn`, in **Function**.

2 This sensor model has additive noise. Therefore, specify the GPS measurement noise as `Additive` in the **Measurement Noise** drop-down list.

3 Specify the measurement noise covariance as [100 0; 0 100].

In the **Multirate** tab, since the two sensors are operating at different sample rates, perform the following configuration:

1   Select `Enable multirate operation`.

2   Specify the state transition sample time. The state transition sample time must be the smallest, and all measurement sample times must be an integer multiple of the state transition sample time. Specify **State Transition** sample time as 0.05, the sample time of the fastest measurement. Though not required in this example, it is possible to have a smaller sample time for state transition than all measurements. This means there will be some sample times without any measurements. For these sample times the filter generates state predictions using the state transition function.

3   Specify the **Measurement 1** sample time (Radar) as 0.05 seconds and **Measurement 2** (GPS) as 1 seconds.
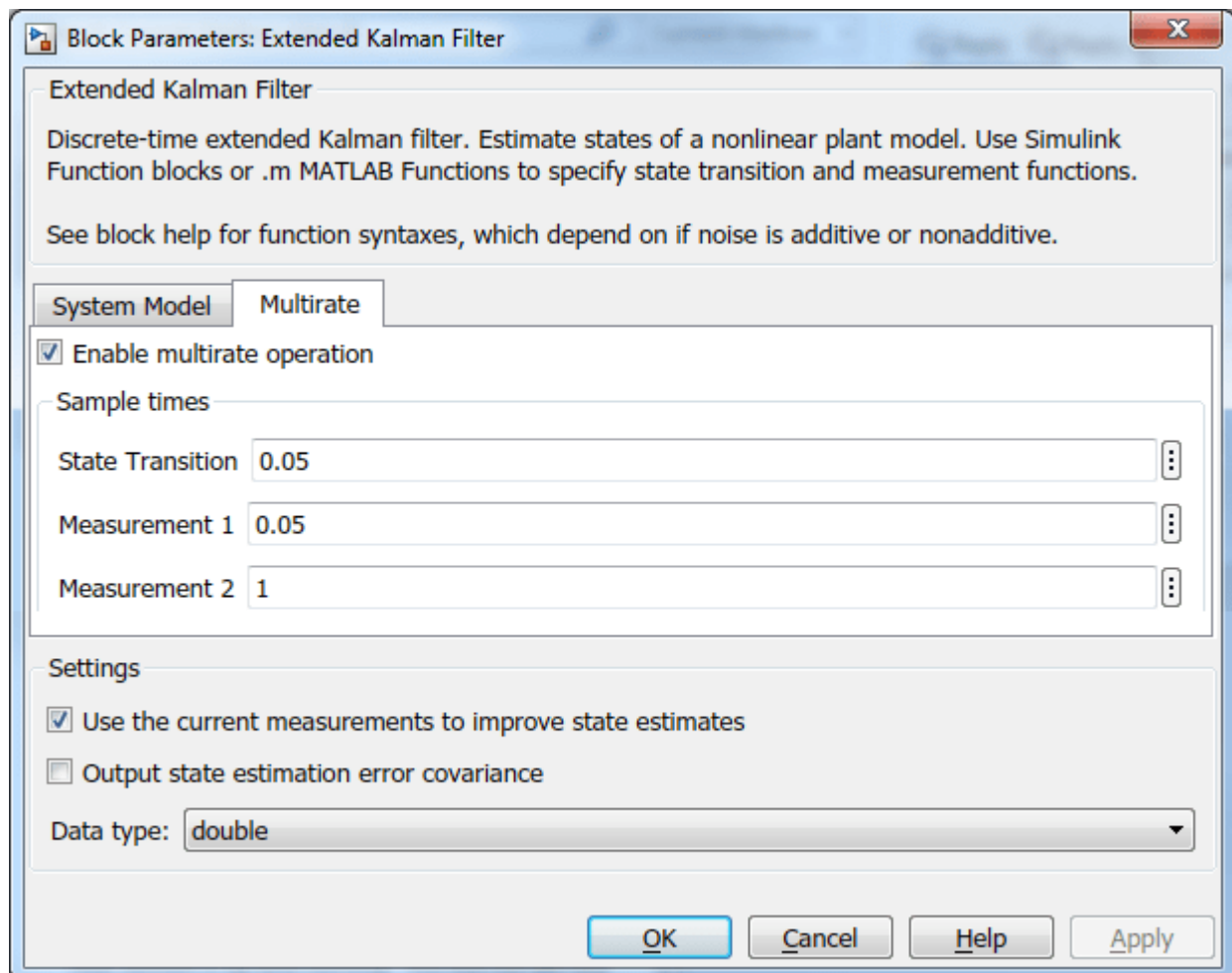
Simulation and Results

Test the performance of the Extended Kalman filter by simulating a scenario where the object travels in a square pattern with the following maneuvers:

- At t = 0, the object starts at $x_e(0) = 100$ [m], $x_n(0) = 100$ [m]

- It heads north at $\dot{x}_n = 50 \, [\text{m/s}]$ until t = 20 seconds.
- It heads east at $\dot{x}_n = 40 \, [\text{m/s}]$ between t = 20 and t = 45 seconds.
- It heads south at $\dot{x}_n = -25 \, [\text{m/s}]$ between t = 45 and t = 85 seconds.
- It heads west at $\dot{x}_e = -10 \, [\text{m/s}]$ between t = 85 and t = 185 seconds.

Generate the true state values corresponding to this motion:

```
Ts = 0.05; % [s] Sample rate for the true states
[t, xTrue] = generateTrueStates(Ts); % Generate position and velocity profile over 0-185 seconds
```

Simulate the model. For instance, look at the actual and estimated velocities in the east direction:

```
sim('multirateEKFExample');
open_system('multirateEKFExample/Scope - East Velocity');
```
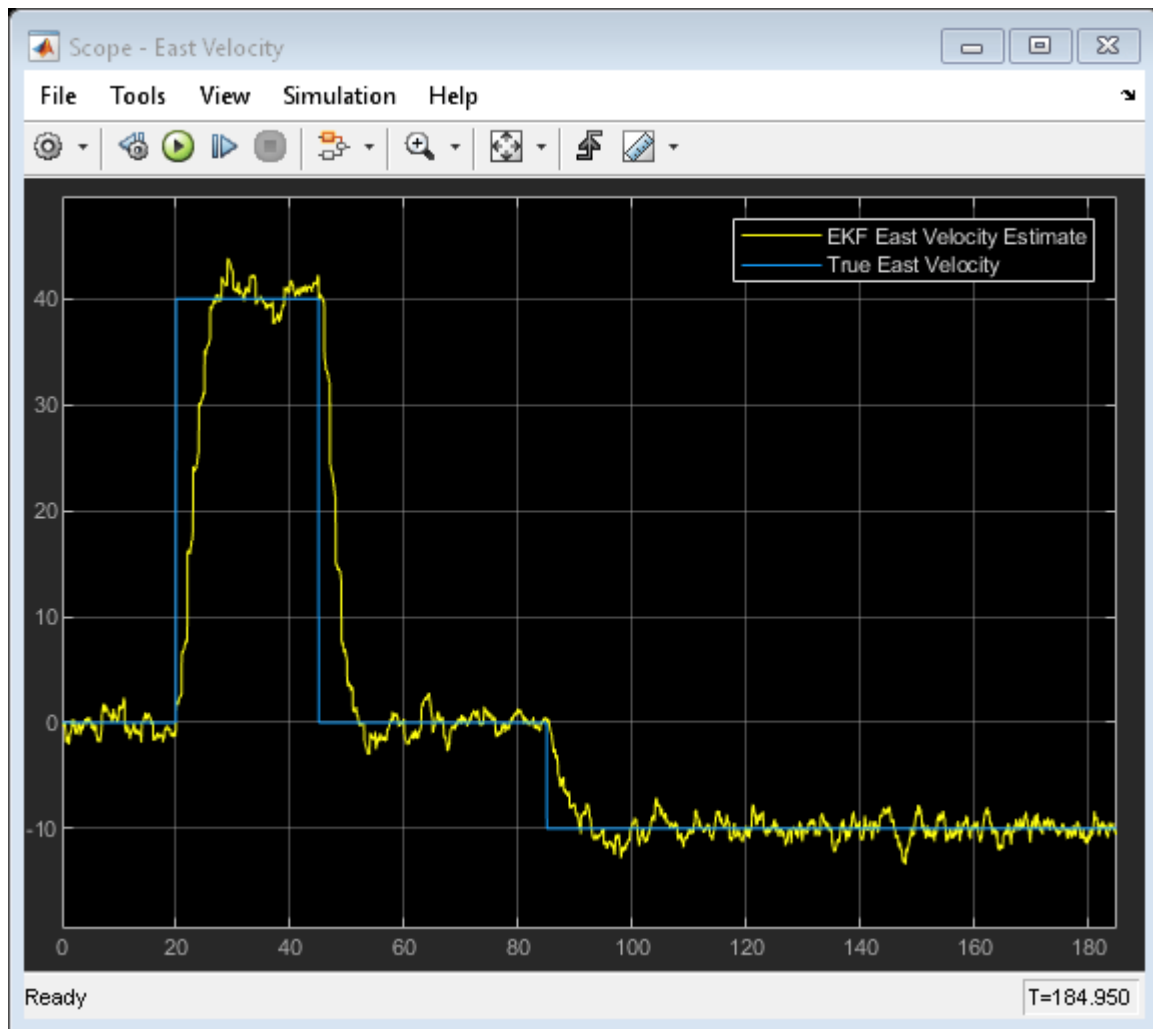

The plot shows the true velocity in the east direction, and its extended Kalman filter estimates. The filter successfully tracks the changes in velocity. The multirate nature of the filter is most apparent in the time range t = 20 to 30 seconds. The filter makes large corrections every second (GPS sample rate), while the corrections due to radar measurements are visible every 0.05 seconds.

Next Steps

1 Validate the state estimation: The validation of unscented and extended Kalman filter performance is typically done using extensive Monte Carlo simulations. For more information, see "Validate Online State Estimation in Simulink" on page 16-36.

2 Generate code: The Unscented and Extended Kalman Filter blocks support C and C++ code generation using Simulink Coder™ software. The functions you provide to these blocks must comply with the restrictions of MATLAB Coder™ software (if you are using MATLAB functions to model your system) and Simulink Coder software (if you are using Simulink Function blocks to model your system).

Summary

This example has shown how to use the Extended Kalman Filter block in System Identification Toolbox. You estimated position and velocity of an object from two different sensors operating at different sampling rates.

```
close_system('multirateEKFExample', 0);
rmpath(fullfile(matlabroot,'examples','ident','main')) % remove example data
```

See Also
Extended Kalman Filter | Unscented Kalman Filter | Particle Filter

More About
- "Extended and Unscented Kalman Filter Algorithms for Online State Estimation" on page 16-27
- "Validate Online State Estimation in Simulink" on page 16-36
- "Troubleshoot Online State Estimation" on page 16-42

Parameter and State Estimation in Simulink Using Particle Filter Block

This example demonstrates the use of Particle Filter block in System Identification Toolbox™. A discrete-time transfer function parameter estimation problem is reformulated and recursively solved as a state estimation problem.

Introduction

The System Identification Toolbox has three Simulink blocks for nonlinear state estimation:

- Particle Filter: Implements a discrete-time particle filter algorithm.
- Extended Kalman Filter: Implements the first-order discrete-time extended Kalman filter algorithm.
- Unscented Kalman Filter: Implements the discrete-time unscented Kalman filter algorithm.

These blocks support state estimation using multiple sensors operating at different sample rates. A typical workflow for using these blocks is as follows:

1 Model your plant and sensor behavior using MATLAB or Simulink functions.
2 Configure the parameters of the block.
3 Simulate the filter and analyze results to gain confidence in filter performance.
4 Deploy the filter on your hardware. You can generate code for these filters using Simulink Coder™ software.

This example uses the Particle Filter block to demonstrate the first two steps of this workflow. The last two steps are briefly discussed in the **Next Steps** section. The goal in this example is to estimate the parameters of a discrete-time transfer function (an output-error model) recursively, where the model parameters are updated at each time step as new information arrives.

If you are interested in the Extended Kalman Filter, see the example "Estimate States of Nonlinear System with Multiple, Multirate Sensors". The use of Unscented Kalman Filter follows similar steps to Extended Kalman Filter.

Add the example file folder to the MATLAB path.

```
addpath(fullfile(matlabroot,'examples','ident','main'));
```

Plant Modeling

Most state estimation algorithms rely on a plant model (state transition function) that describe the evolution of plant states from one time step to the next. This function is typically denoted as $x[k+1] = f(x[k], w[k], u[k])$ where x is the states, w is the process noise, u is the optional additional inputs, for instance system inputs or parameters. The particle filter block requires you to provide this function in a slightly different syntax, $X[k+1] = f_{pf}(X[k], u[k])$. The differences are:

- Particle filter works by following the trajectories of many state hypotheses (particles), and the block passes all state hypotheses to your function at once. Concretely, if your state vector x has N_s elements and you choose N_p particles to use, X has the dimensions $[N_s \ N_p]$ where each column is a state hypothesis.

- You calculate the impact of process noise w on the state hypotheses $X[k+1]$ in your function $f_{pf}(\cdots)$. The block does not make any assumptions on the probability distribution of the process noise w, and does not need w as an input.

The function $f_{pf}(\cdots)$ can be a MATLAB Function that comply with the restrictions of MATLAB Coder™, or a Simulink Function block. After you create $f_{pf}(\cdots)$, you specify the function name in the Particle Filter block.

In this example, you are reformulating a discrete-time transfer function parameter estimation problem as a state estimation problem. This transfer function may be representing the dynamics of a discrete-time process, or it may be representing some continuous-time dynamics coupled with a signal reconstructor such as zero-order hold. Assume that you are interested in estimating the parameters of a first-order discrete-time transfer function:

$$y[k] = \frac{20q^{-1}}{1 - 0.7q^{-1}}u[k] + e[k]$$

Here $y[k]$ is the plant output, $u[k]$ is the plant input, $e[k]$ is the measurement noise, q^{-1} is the time-delay operator so that $q^{-1}u[k] = u[t-1]$. Parametrize the transfer function as $\frac{n\,q^{-1}}{1 + d\,q^{-1}}$, where n and d are parameters to be estimated. The transfer function and the parameters can be represented in the necessary state-space form in multiple ways, by choice of the state vector. One choice is $x[k] = [\,y[k]; d[k]; n[k]\,]$ where the second and third states represent the parameter estimates. Then the transfer function can be equivalently written as

$$x[k+1] = \begin{bmatrix} -x_2[k]x_1[k] + x_3[k]u[k] \\ x_2[k] \\ x_3[k] \end{bmatrix}$$

The measurement noise term $e[k]$ is handled in sensor modeling. In this example, you implement the expression above in a MATLAB Function, in a vectorized form for computational efficiency:

type `pfBlockStateTransitionFcnExample`

```
function xNext = pfBlockStateTransitionFcnExample(x,u)
% pfBlockStateTransitionFcnExample State transition function for particle
%                                  filter, for estimating parameters of a
%                                  SISO, first order, discrete-time transfer
%                                  function model
%
% Inputs:
%    x - Particles, a NumberOfStates-by-NumberOfParticles matrix
%    u - System input, a scalar
%
% Outputs:
%    xNext - Predicted particles, with the same dimensions as input x

% Implement the state-transition function
%    xNext = [x(1)*x(2) + x(3)*u;
%             x(2);
```

```
%                 x(3)];
% in vectorized form (for all particles).
xNext = x;
xNext(1,:) = bsxfun(@times,x(1,:),-x(2,:)) + x(3,:)*u;

% Add a small process noise (relative to expected size of each state), to
% increase particle diversity
xNext = xNext + bsxfun(@times,[1; 1e-2; 1e-1],randn(size(xNext)));
end
```

Sensor modeling

The Particle Filter block requires you to provide a measurement likelihood function that calculates the likelihood (probability) of each state hypothesis. This function has the form $L[k] = h_{pf}(X[k], y[k], u[k])$. $L[k]$ is an N_p element vector, where N_p is the number of particles you choose. m^{th} element in $L[k]$ is the likelihood of the mth particle (column) in $X[k]$. $y[k]$ is the sensor measurement. $u[k]$ is an optional input argument, which can differ from the inputs of the state transition function.

The sensor measures the first state in this example. This example assumes that the errors between the actual and predicted measurements are distributed according to a Gaussian distribution, but any arbitrary probability distribution or some other method can be used to calculate the likelihoods. You create $h_{pf}(...)$, and specify the function name in the Particle Filter block.

```
type pfBlockMeasurementLikelihoodFcnExample;
```

```
function likelihood = pfBlockMeasurementLikelihoodFcnExample(particles, measurement)
% pfBlockMeasurementLikelihoodFcnExample Measurement likelihood function for particle filter
%
% The measurement is the first state
%
% Inputs:
%    particles   - A NumberOfStates-by-NumberOfParticles matrix
%    measurement - System output, a scalar
%
% Outputs:
%    likelihood - A vector with NumberOfParticles elements whose n-th
%                 element is the likelihood of the n-th particle

%#codegen

% Predicted measurement
yHat = particles(1,:);

% Calculate likelihood of each particle based on the error between actual
% and predicted measurement
%
% Assume error is distributed per multivariate normal distribution with
% mean value of zero, variance 1. Evaluate the corresponding probability
% density function
e = bsxfun(@minus, yHat, measurement(:)'); % Error
numberOfMeasurements = 1;
mu = 0; % Mean
Sigma = eye(numberOfMeasurements); % Variance
measurementErrorProd = dot((e-mu), Sigma \ (e-mu), 1);
```

```
c = 1/sqrt((2*pi)^numberOfMeasurements * det(Sigma));
likelihood = c * exp(-0.5 * measurementErrorProd);
end
```

Filter Construction

Configure the Particle Filter block for estimation. You specify the state transition and measurement likelihood function names, number of particles, and the initial distribution of these particles.

In the **System Model** tab of the block dialog, specify the following parameters:

State Transition

1   Specify the state transition function, `pfBlockStateTransitionFcnExample`, in **Function**. When you enter the function name and click **Apply**, the block detects that your function has an extra input, u, and creates the input port **StateTransitionFcnInputs**. You connect your system input to this port.

Initialization

1   Specify `10000` in **Number of particles**. Higher number of particles typically correspond to better estimation, at increased computational cost.

2   Specify `Gaussian` in **Distribution** to get an initial set of particles from a multivariate Gaussian distribution. Then specify `[0; 0; 0]` in **Mean** because you have three states and this is your best guess. Specify `diag([10 5 100])` in **Covariance** to specify a large variance (uncertainty) in your guess for the third state, and smaller variance for the first two. It is critical that this initial set of particles are spread wide enough (large variance) to cover the potential true state.

Measurement 1

1   Specify the name of your measurement likelihood function, `pfBlockMeasurementLikelihoodFcnExample`, in **Function**.

Sample Time

1   At the bottom of the block dialog, enter 1 in **Sample time**. If you have a different sample time among state transition and measurement likelihood functions, or if you have multiple sensors with different sample times, these can be configured in the **Block outputs, Multirate** tab.

The particle filter involves deleting the particles with low likelihoods and seeding new particles using the ones with higher likelihoods. This is controlled by the options under the **Resampling** group. This example uses the default settings.

By default, the block only outputs the mean of the state hypotheses, weighted by their likelihoods. To see all the particles, weights, or to choose a different method of extracting a state estimate, check out the options in the **Block outputs, Multirate** tab.

Simulation and Results

For a simple test, the true plant model is simulated with white noise inputs. The inputs and the noisy measurements from the plant are fed to the Particle Filter block. The following Simulink model represents this setup.

```
open_system('pfBlockExample')
```
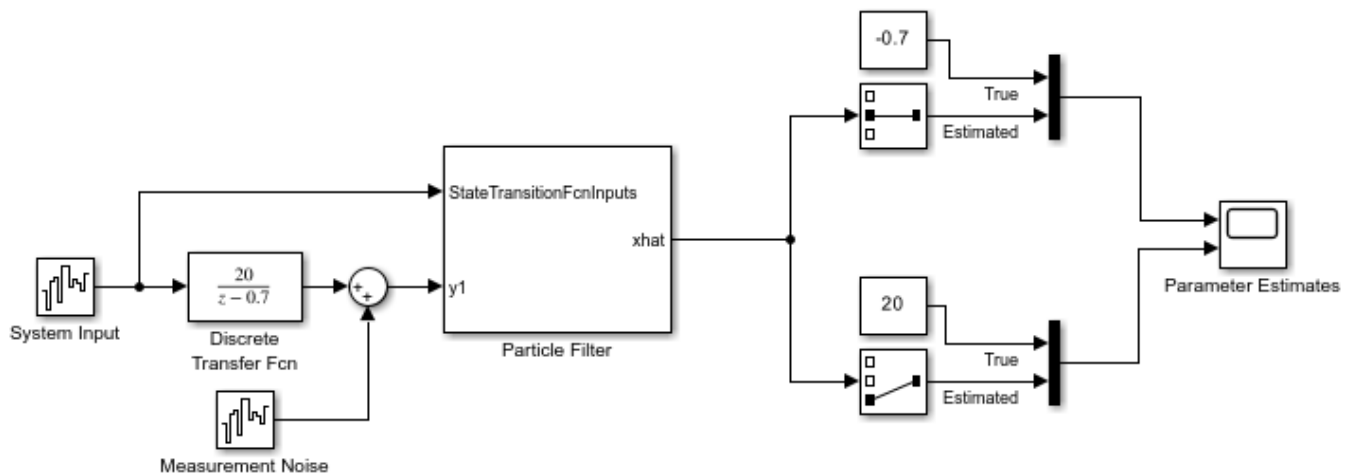

Copyright 2020-2021 The MathWorks Inc.

Simulate the system and compare the estimated and true parameters:

```
sim('pfBlockExample')
open_system('pfBlockExample/Parameter Estimates')
```
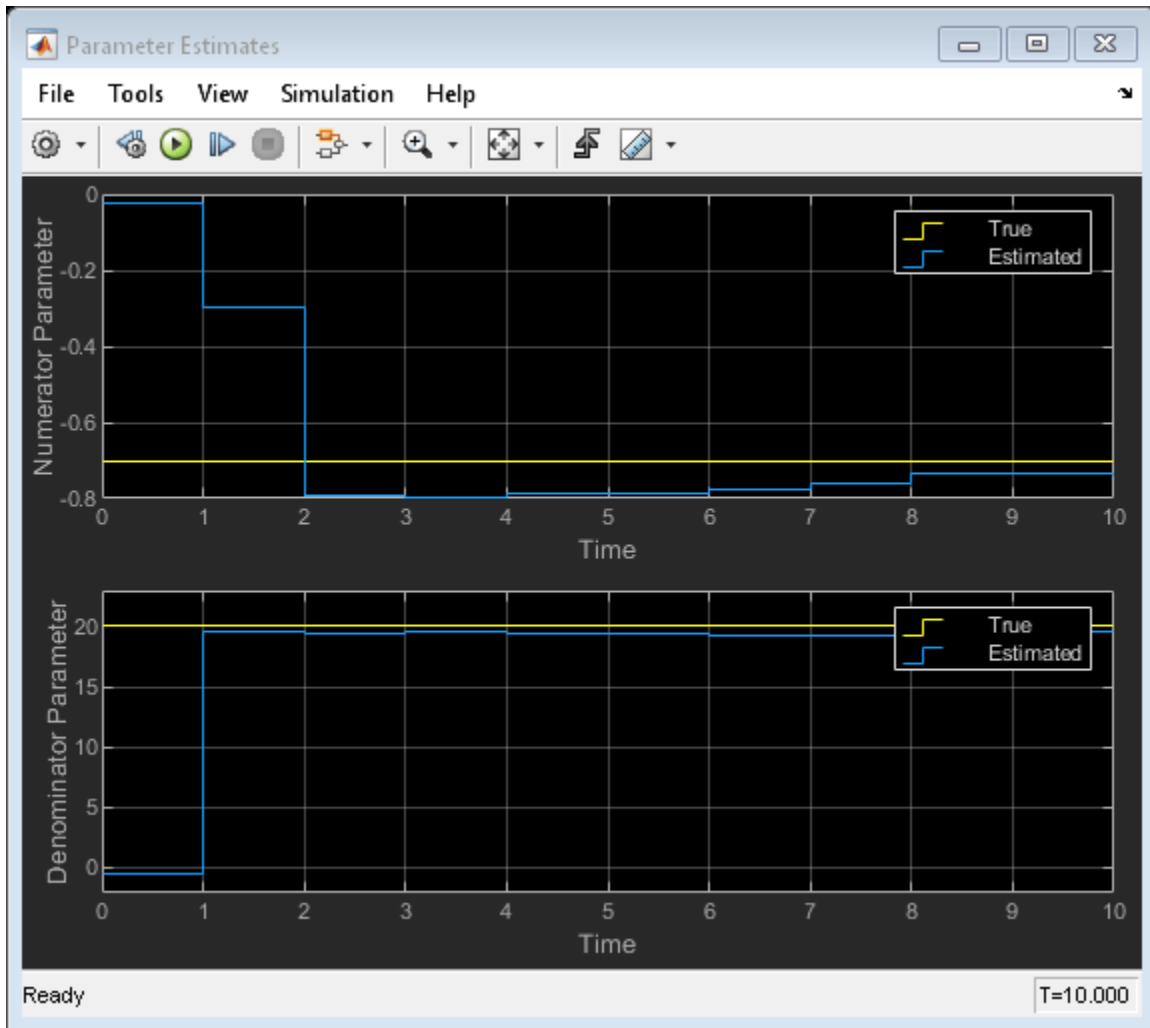
The plot shows the true numerator and denominator parameters, and their particle filter estimates. The estimates approximately converge to the true values after 10 time steps. Convergence is obtained even though the initial state guess was far from the true values.

Troubleshooting

A few potential implementation issues and troubleshooting ideas are listed here, in case the particle filter is not performing as expected for your application.

Troubleshooting of the particle filter is typically performed by looking into the set of particles and their weights, which can be obtained by choosing **Output all particles** and **Output all weights** in the **Block outputs, Multirate** tab of the block dialog.

The first sanity check is to ensure that the state transition and measurement likelihood functions capture the behavior of your system reasonably well. If you have a simulation model for your system (and hence access to the true state in simulation), you can try initializing the filter with the true state. Then you can validate if the state transition function calculates the time-propagation of the true state accurately, and if the measurement likelihood function is calculating a high likelihood for these particles.

Initial set of particles is important. Ensure that at least some of the particles have high likelihood at the beginning of your simulation. If the true state is outside the initial spread of state hypotheses, the estimates can be inaccurate or even diverge.

If the state estimation accuracy is fine initially, but deteriorating over time, the issue may be particle degeneracy or particle impoverishment [2]. Particle degeneracy occurs when the particles are distributed too widely, while particle impoverishment occurs because of particles clumping together after resampling. Particle degeneracy leads to particle impoverishment as a result of direct resampling. The addition of artificial process noise in the state-transition function used in this example is one practical approach. There is a large collection of literature on resolving these issues and based on your application more systematic approaches may be available. [1], [2] are two references that can be helpful.

Next Steps

1   Validate the state estimation: Once the filter is performing as expected in a simulation, typically the performance is further validated using extensive Monte Carlo simulations. For more information, see "Validate Online State Estimation in Simulink" on page 16-36. You can use the options under **Randomness** group in the Particle Filter block dialog to facilitate these simulations.

2   Generate code: The Particle Filter block supports C and C++ code generation using Simulink Coder™ software. The functions you provide to this block must comply with the restrictions of MATLAB Coder™ software (if you are using MATLAB functions to model your system) and Simulink Coder software (if you are using Simulink Function blocks to model your system).

Summary

This example has shown how to use the Particle Filter block in System Identification Toolbox. You estimated the parameters of a discrete-time transfer function recursively, where the parameters are updated at each time step as new information arrives.

```
close_system('pfBlockExample',0)
```

Remove the example file folder from the MATLAB path.

```
rmpath(fullfile(matlabroot,'examples','ident','main'));
```

References

[1] Simon, Dan. Optimal state estimation: Kalman, H infinity, and nonlinear approaches. John Wiley & Sons, 2006.

[2] Doucet, Arnaud, and Adam M. Johansen. "A tutorial on particle filtering and smoothing: Fifteen years later." Handbook of nonlinear filtering 12.656-704 (2009): 3.

See Also

Extended Kalman Filter | Unscented Kalman Filter | Particle Filter

More About

*   "Validate Online State Estimation in Simulink" on page 16-36
*   "Troubleshoot Online State Estimation" on page 16-42
*   "Estimate States of Nonlinear System with Multiple, Multirate Sensors" on page 16-115

State Estimation with Wrapped Measurements Using Extended Kalman Filter

This example shows how to use the extended Kalman filter algorithm for nonlinear state estimation for 3D tracking involving circularly wrapped angle measurements. For target tracking, sensors usually employ a spherical frame to report the position of an object in terms of azimuth, range, and elevation. The angular measurements from this set are reported within certain bounds. For instance, azimuth angles are reported within a range of $-180°$ to $180°$ or $0°$ to $360°$. However, tracking is usually done in a rectangular frame, which necessitates the use of a nonlinear filter that can handle the nonlinear transformations required to transform spherical measurements into rectangular states.

This example uses a constant velocity model to track the three-dimensional positions and velocities of a target. An Extended Kalman Filter is used as the nonlinear filter to track the states of this object. This filter uses azimuth (ψ), range, and elevation (ϕ) readings as measurements during this tracking exercise.
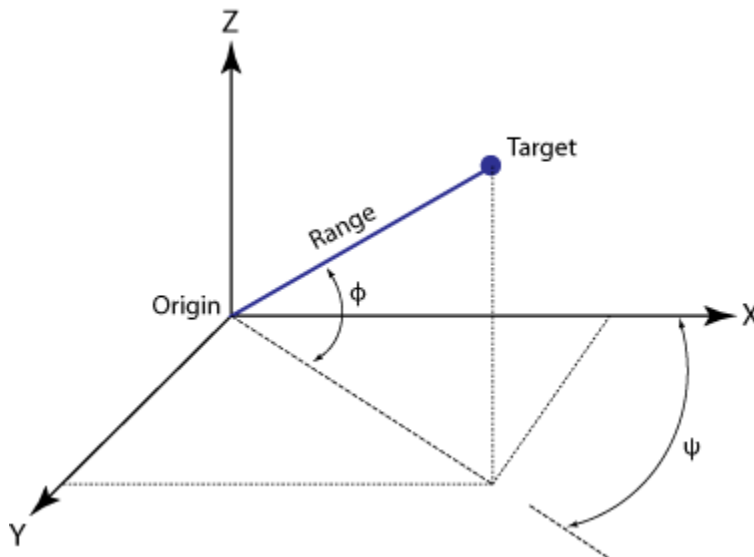
The constant velocity 3D plant model for this example consists of positions and velocities in the three axes as follows:

$$States = \begin{bmatrix} x \\ \dot{x} \\ y \\ \dot{y} \\ z \\ \dot{z} \end{bmatrix} = \begin{bmatrix} x \\ v_x \\ y \\ v_y \\ z \\ v_z \end{bmatrix}$$

The range, azimuth, and elevation angles are given by:

$$Range = \sqrt{x^2 + y^2 + z^2}$$

$$Elevation\ (\phi) = \sin^{-1}\left(\frac{z}{Range}\right)$$

$$Azimuth\ (\psi) = \tan^{-1}\left(\frac{y}{x}\right)$$

Computing the residual between the actual measurement and expected measurement is an essential step in the filtering process. Without the necessary precautions, the residual values may be large for an object located near the wrapping boundary. This causes the filter to diverge from the accurate state, leading to an inaccurate state estimate. This example demonstrates how to handle such situations by using measurement wrapping. A Simulink model is used to demonstrate the difference in tracked states when used with and without measurement wrapping.

EKF model with and without measurement wrapping

Copyright 2021 The MathWorks, Inc

State Estimation - Without Measurement Wrapping

First provide your initial state guess and specify the model name.

```
rng(0)
initialStateGuess = [-100 0 0 0 0 0]';
modelName = 'modelEKFWrappedMeasurements';
```

Generate the measurements based on the noise covariance as follows:

```
dt = 3;
tSpan = 0:dt:100;
yTrue = zeros(length(tSpan),3);
yMeas = zeros(size(yTrue));
noiseCovariance = diag([20 0.5 0.5]);
xCurrent = initialStateGuess;

for i = 1:numel(tSpan)
    xTrue = stateTransitionFcn(xCurrent);
    yTrue(i,:) = (measurementFcn(xTrue))';
```

```
    yMeas(i,:) = yTrue(i,:) + (chol(noiseCovariance)*randn(3,1))';
    xCurrent = xTrue;
end
```

Simulate the model.

```
out = sim(modelName);
xEstNoWrap = zeros(numel(tSpan),length(initialStateGuess));
xEstWithWrap = zeros(numel(tSpan),length(initialStateGuess));
```

Run the filter and vizualize the state convergence on a 3D plot.

```
figure()
plot3(initialStateGuess(1), initialStateGuess(3), initialStateGuess(5),'ro','MarkerFaceColor','r
grid
hold on;

% Plot options
xlabel('x');
ylabel('y');
zlabel('z');
title('No measurement wrapping');

% Run filter
for i = 1:numel(tSpan)
    xEst = out.noWrapOut(i,:);
    plot3(xEst(1),xEst(3),xEst(5),'bo','MarkerFaceColor','b');
    hold on;
    xEstNoWrap(i,:) = xEst;
end
hold off
```

No measurement wrapping
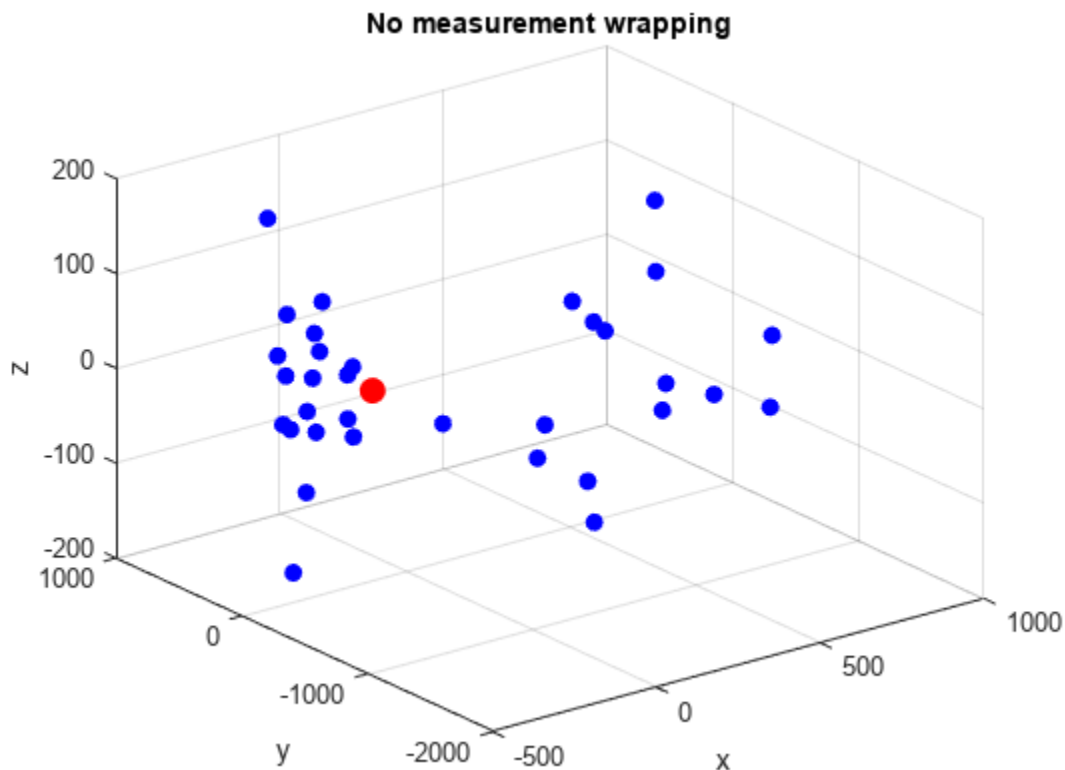
State Estimation - With Measurement Wrapping

Now, enable the measurement wrapping and run the filter. Visualize the state convergence on a 3D plot.

```
figure()
% Run filter
for i = 1:numel(tSpan)
    xEst = out.wrapOut(i,:);
    plot3(xEst(1),xEst(3),xEst(5),'bo','MarkerFaceColor','b');
    hold on;
    xEstWithWrap(i,:) = xEst;
end
plot3(initialStateGuess(1), initialStateGuess(3), initialStateGuess(5),'ro','MarkerFaceColor','r
grid
title('With measurement wrapping');
hold off
```

With measurement wrapping
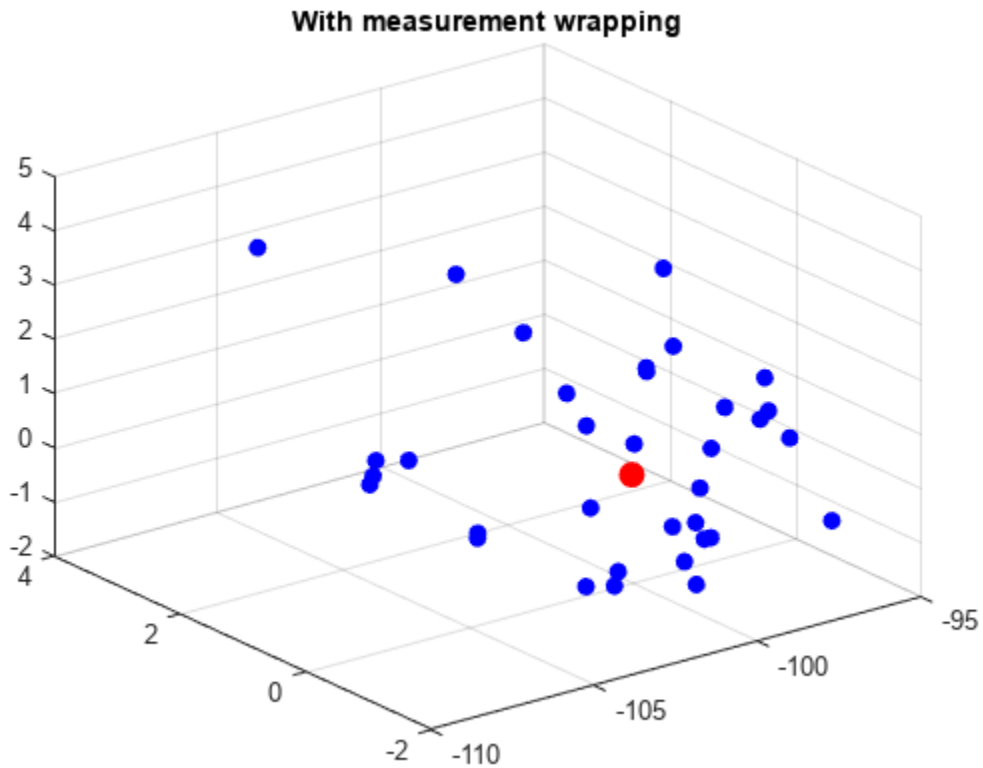
From the plot above, observe that the scale of the three axes is a lot smaller than the 3D plot for the case without measurement wrapping. This denotes that there is better convergence when measurement wrapping is enabled.

Validation

Compare the estimation errors for both cases on the same plot.

```
figure()
subplot(3,1,1)
plot(xEstNoWrap(:,1) - xTrue(1));
hold on
plot(xEstWithWrap(:,1) - xTrue(1));
ylim([-500 1200])
title('e_x');
hold off

subplot(3,1,2)
plot(xEstNoWrap(:,3) - xTrue(3));
hold on
plot(xEstWithWrap(:,3) - xTrue(3));
ylim([-2000 1000])
hold off
title('e_y');

subplot(3,1,3)
plot(xEstNoWrap(:,5) - xTrue(5));
```

```
hold on
plot(xEstWithWrap(:,5) - xTrue(5));
ylim([-200 200])
hold off
title('e_z');

sgtitle('Estimation Error Plot')
legend('No Wrapping', 'With Wrapping','location','best')
```


Estimation Error Plot
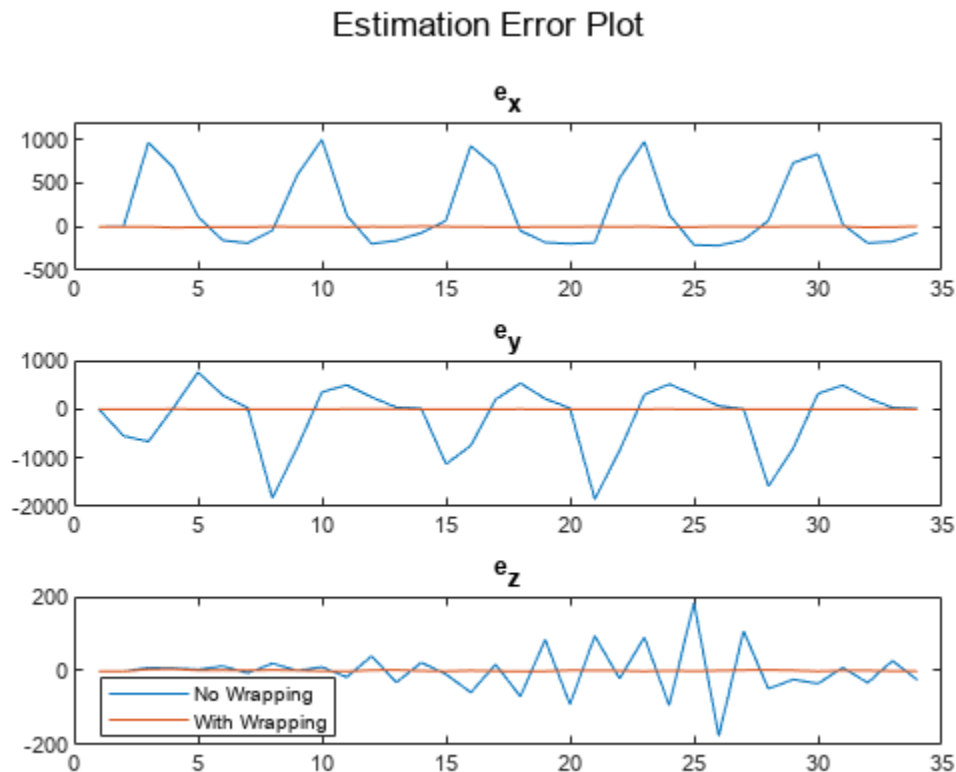
Observe that the errors are comparatively much larger in all the three axes for the case without measurement wrapping.

Summary

This example shows how to use an Extended Kalman Filter block for state estimation of a nonlinear system with and without measurement wrapping. The state estimation with wrapping enabled is observed to provide more accurate state estimates as determined by the 3D and error plots.

Supporting Functions

State Transition Function

The Extended Kalman Filter block requires a function that describes the evolution of states from one time step to the next. This is typically called the state transition function. For this example, the contents of the state transition function contained in `stateTransitionFcn.m` is as follows:

```
function x = stateTransitionFcn(x)
    dt = 3;
```

```
    A = [1 dt 0 0  0 0;
         0 1  0 0  0 0;
         0 0  1 dt 0 0;
         0 0  0 1  0 0;
         0 0  0 0  1 dt;
         0 0  0 0  0  1];
    x = A*x;
end
```

Measurement Function

The Extended Kalman Filter block also needs a function that describes how the model states are related to sensor measurements. This is the measurement function. For this example, the contents of the measurement function contained in `measurementFcn.m` is as follows:

```
function [y, bounds] = measurementFcn(x)
    xPos = x(1);
    yPos = x(3);
    zPos = x(5);

    % Range
    r = sqrt(xPos^2 + yPos^2 + zPos^2);
    % Azimuth
    psi = atan2d(yPos,xPos);
    % Elevation
    phi = asind(zPos/r);
    % Combined measurement
    y = [r ; psi; phi];
    bounds = [-inf inf;-180 180; -90 90];
end
```

See Also
Extended Kalman Filter

Related Examples

- "Validate Online State Estimation in Simulink" on page 16-36
- "Troubleshoot Online State Estimation" on page 16-42
- "Estimate States of Nonlinear System with Multiple, Multirate Sensors" on page 16-115

Model Analysis

Validating Models After Estimation

Ways to Validate Models

You can use the following approaches to validate models:

- Comparing simulated or predicted model output to measured output.

  See "Simulate and Predict Identified Model Output" on page 17-6.

  To simulate identified models in the Simulink environment, see "Simulate Identified Model in Simulink" on page 20-5.

- Analyzing autocorrelation and cross-correlation of the residuals with input.

  See "What Is Residual Analysis?" on page 17-43.

- Analyzing model response. For more information, see the following:

  - "Impulse and Step Response Plots" on page 17-51
  - "Frequency Response Plots for Model Validation" on page 17-57

  For information about the response of the noise model, see "Noise Spectrum Plots" on page 17-64.

- Plotting the poles and zeros of the linear parametric model.

  For more information, see "Pole and Zero Plots" on page 17-69.

- 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 "Compute Model Uncertainty" on page 17-94.

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 2-81.

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	"Simulate and Predict Identified Model Output" on page 17-6
Residual Analysis	All linear and nonlinear models	"What Is Residual Analysis?" on page 17-43
Transient Response	• All linear parametric models • Correlation analysis (nonparametric) models • For nonlinear models, only step response.	"Impulse and Step Response Plots" on page 17-51
Frequency Response	All linear models	"Frequency Response Plots for Model Validation" on page 17-57
Noise Spectrum	• All linear parametric models • Spectral analysis (nonparametric) models	"Noise Spectrum Plots" on page 17-64
Poles and Zeros	All linear parametric models	"Pole and Zero Plots" on page 17-69
Nonlinear ARX	Nonlinear ARX models only	Nonlinear ARX Plots on page 11-39
Hammerstein-Wiener	Hammerstein-Wiener models only	Hammerstein-Wiener Plots on page 12-22

See Also

Related Examples

- "Plot Models in the System Identification App" on page 17-5
- "Compare Simulated Output with Measured Validation Data" on page 17-23

More About

- "Validating Models After Estimation" on page 17-2

Plot Models in the System Identification App

To create one or more plots of your models, select the corresponding check box in the **Model Views** area of the System Identification app. An *active* model icon has a thick line in the icon, while an *inactive* model has a thin line. Only active models appear on the selected plots.

To include or exclude a model on a plot, click the corresponding icon in the System Identification app. Clicking the model icon updates any plots that are currently open.

For example, in the following figure, **Model output** is selected. In this case, the models n4s3 is not included on the plot because only arxqs is active.

Plots Include Only Active Models

To close a plot, clear the corresponding check box in the System Identification app.

Tip To get information about a specific plot, select a help topic from the **Help** menu in the plot window.

See Also

## Related Examples

- "Interpret the Model Output Plot" on page 17-15
- "Change Model Output Plot Settings" on page 17-16
- "Working with Plots" on page 21-8
- "Compare Simulated Output with Measured Validation Data" on page 17-23

Simulate and Predict Identified Model Output

You identify a model so that you can accurately compute a dynamic system response to an input. There are two ways of generating an identified model response:

- Simulation computes the model response using input data and initial conditions.
- Prediction computes the model response at some specified amount of time in the future using the current and past values of measured input and output values, as well as initial conditions.
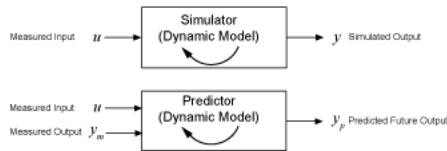
In system identification, the goal is to create a realistic dynamic system model that can then be used or handed off for an application goal. In this context, the main roles of simulation and prediction within the System Identification Toolbox are to provide tools for model identification, and also for choosing, tuning, and validating these models.

You can:

- Identify your model in a manner that minimizes either prediction (prediction focus) or simulation error (simulation focus)
- Visualize your model response in comparison with other models and with data measurements
- Validate your model by comparing its response with measured input/output data that was not used for the original model estimation

Your choice of simulation or prediction approach depends on what your application needs are, but also where you are in the system identification workflow:

- When you are identifying your models, a one-step prediction focus generally produces the best results. This advantage is because, by using both input and output measurements, one-step prediction accounts for the nature of the disturbances. Accounting for disturbances provides the most statistically optimal results.
- When you are validating your models, simulation usually provides the more perceptive approach for assessing how your model will perform under a wide range of conditions. Your application may drive prediction-based validation as well, however. 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.

You can work in either the time domain or the frequency domain, and remain consistent with the domain of the input/output data. For frequency-domain data, the simulation results are products of the Fourier transform of the input and frequency function of the model. Because frequency-response model identification ignores noise dynamics, simulation focus and one-step prediction focus yield the same identified model. For validation in the frequency domain, use simulation.

For examples, see:

- "Compare Predicted and Simulated Response of Identified Model to Measured Data" on page 17-9

- "Compare Models Identified with Prediction and with Simulation Focus" on page 17-12

What are Simulation and Prediction?

You can get a more detailed understanding of the differences between simulation and prediction by applying these techniques to a simple first-order system.

Simulation

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. In other words, given inputs $u(t_1, \ldots ,t_N)$, the simulation generates $y(t_1, \ldots ,t_N)$. The following diagram illustrates this flow.
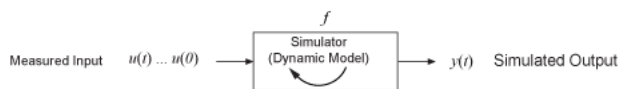

For a continuous-time system, simulation means solving a differential equation. For a discrete-time system, simulation means directly applying the model equations.

For example, consider a dynamic model described by a first-order difference equation that uses a sample time of 1 second:

$y(t+1) = ay(t) + bu(t),$

where y is the output and u is the input.

This system is equivalent to the following block diagram.
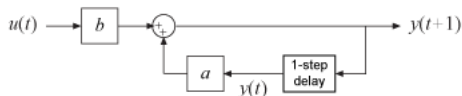
Suppose that your model identification provides you with estimated parameter values of $a = 0.9$ and $b = 1.5$. Then the equation becomes:

$y(t+1) = 0.9y(t) + 1.5u(t).$

Now suppose that 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

Prediction means projecting the model response k steps ahead into the future using the current and past values of measured input and output values. k is called the prediction horizon, and corresponds

to predicting output at time kT_s, where T_s is the sample time. In other words, given measured inputs $u_m(t_1, \ldots, t_{N+k})$ and measured outputs $y_m(t_1, \ldots, t_N)$, the prediction generates $y_p(t_{N+k})$.

For example, suppose that you use sensors to measure the input signal $u_m(t)$ and output signal $y_m(t)$ of the physical system described in the previous first-order equation. The equation becomes:

$$y_p(t+1) = ay_m(t) + bu_m(t),$$

where y is the output and u is the input.

The predictor version of the previous simulation block diagram is:
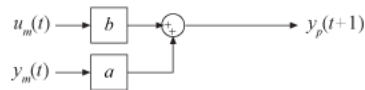
At the 10th sampling instant ($t = 10$), the measured output $y_m(10)$ is 16 mm and the corresponding input $u_m(10)$ is 12 N. Now, you want to predict the value of the output at the future time $t = 11$. Using the previous equation, the predicted output y_p is:

$$y_p(11) = 0.9y_m(10) + 1.5u_m(10)$$

Hence, the predicted value of future output $y(11)$ at time $t = 10$ is:

$$y_p(11) = 0.9*16 + 1.5*12 = 32.4$$

In general, to predict the model response k steps into the future ($k \geq 1$) from the current time t, you must know the inputs up to time $t+k$ and outputs up to time t:

$$y_p(t+k) = f(u_m(t+k), u_m(t+k-1), \ldots, u_m(t), u_m(t-1), \ldots, u_m(0),$$
$$y_m(t), y_m(t-1), y_m(t-2), \ldots, y_m(0))$$

$u_m(0)$ and $y_m(0)$ are the initial states. $f()$ represents the predictor, which is a dynamic model whose form depends on the model structure.
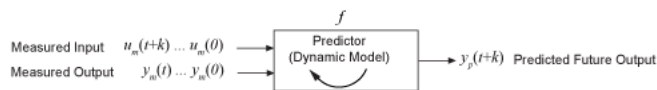
The one-step-ahead predictor from the previous example, y_p of the model $y(t) + ay(t-1) = bu(t)$ is:

$$y_p(t+1) = -ay_p(t) + bu_m(t+1)$$

In this simple one-step predictor case, the newest prediction is based only on measurements. For multiple-step predictors, the dynamic model propagates states internally, using the previous predicted states in addition to the inputs. Each predicted output therefore arises from a combination of the measured input and outputs and the previous predicted outputs.

You can set k to any positive integer value up to the number of measured data samples. If you set k to ∞, then no previous outputs are used in the prediction computation, and prediction returns the same result as simulation. If you set k to an integer greater than the number of data samples, `predict` sets k to `Inf` and issues a warning. If your intent is to perform a prediction in a time range beyond the last instant of measured data, use `forecast`.

For an example showing prediction and simulation in MATLAB, see "Compare Predicted and Simulated Response of Identified Model to Measured Data" on page 17-9.

Limitations on Prediction

Not all models support a predictive approach. For the previous dynamic model,

$$H(z) = \frac{1}{1 + az^{-1}}$$, the structure

supports use of past data. This support does not exist 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 these cases, prediction and simulation coincide. Even models that generally do use past information can still have OE structure in special cases. State-space models (`idss`) have OE structure when `K=0`. Polynomial models (`idpoly`) also have OE structure, when `a=c=d=1`. In these special cases, prediction and simulation are equivalent, and the disturbance model is fixed to `1`.

Compare Predicted and Simulated Response of Identified Model to Measured Data

This example shows how to visualize both the predicted model response and the simulated model response of an identified linear model.

Identify a third-order state-space model using the input/output measurements in `z1`.

```
load iddata1 z1;
sys = ssest(z1,3);
```

`sys` is a continuous-time identified state-space (`idss`) model. Here, `sys` is identified using the default 1-step prediction focus, which minimizes the 1-step prediction error. This focus generally provides the best overall model.

Now use `compare` to plot the predicted response. For this example, set the prediction horizon `kstep` to 10 steps, and use `compare` to plot the predicted response against the original measurement data. This setting of `kstep` specifies that each response point is 10 steps in the future with respect to the measurement data used to predict that point.

```
kstep = 10;
figure
compare(z1,sys,kstep)
```

10-Step Predicted Response Comparison
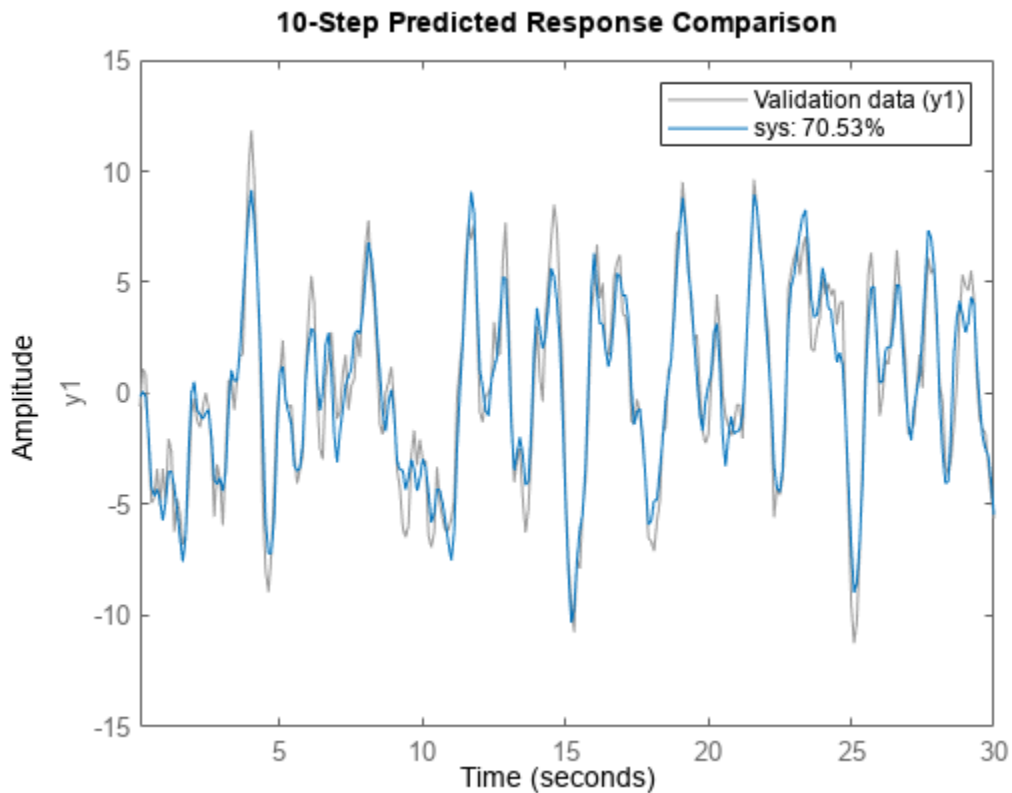
In this plot, each data point represents the predicted output associated with output measurement data that was taken at least 10 steps earlier. For instance, the point at t=15 seconds is based on measurements taken at or prior to t=5 seconds.

The plot illustrates the differences between the model response and the original data. The percentage in the legend is the NRMSE fitness value. It represents how closely the predicted model output matches the data.

To improve your results, you can reduce the prediction horizon.

```
kstep = 5;
figure
compare(z1,sys,kstep)
```

5-Step Predicted Response Comparison
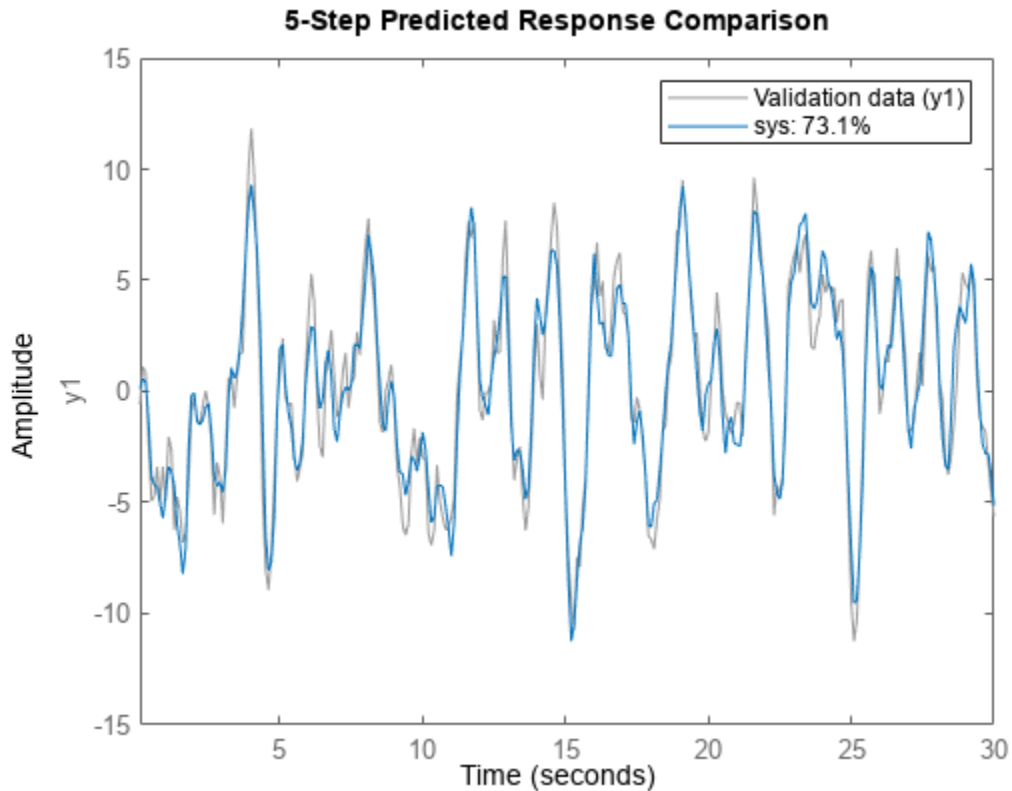
The NRMSE fitness value has improved from the fitness value obtained in the 10-step case. In an actual application, there are various factors that influence how small the prediction horizon can be. These include time constants and application look-ahead requirements.

You can view the simulated response for comparison, rather than the predicted response, by using the `kstep` default for `compare`, which is `Inf`. With simulation, the response computation uses only the input data, not the measured output data.

```
figure
compare(z1,sys)
```

Simulated Response Comparison
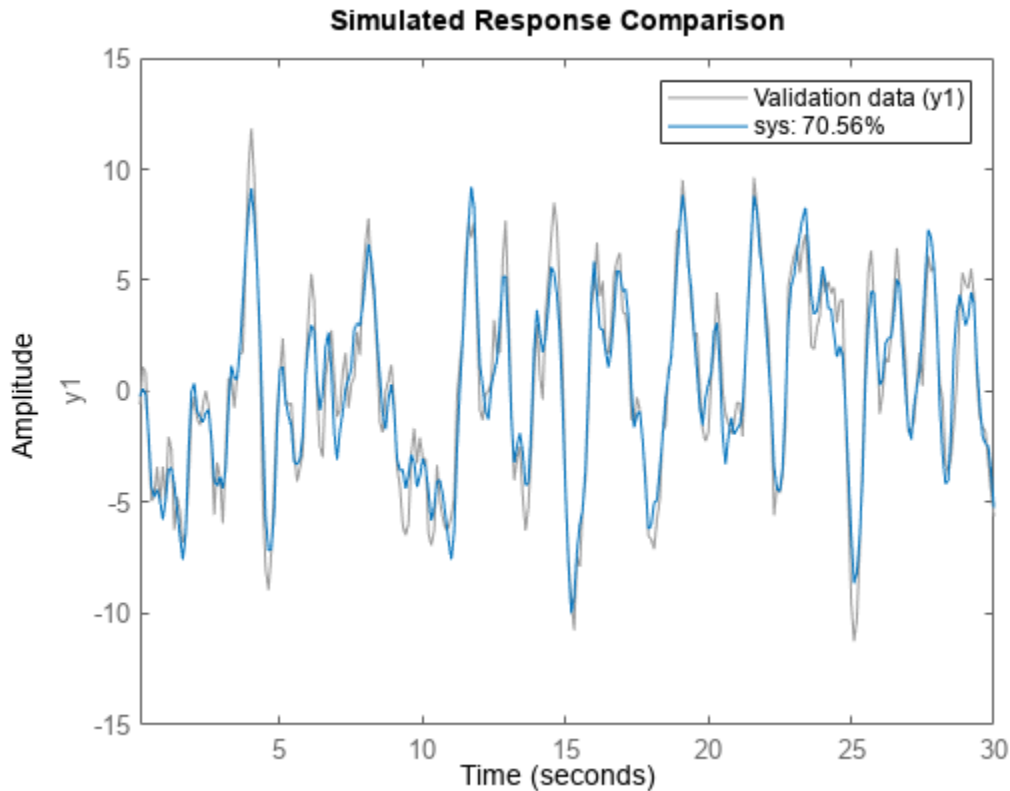
The simulated and the 10-step predicted responses yield similar overall fit percentages.

To change display options in the plot, right-click the plot to access the context menu. For example, to plot the error between the predicted output and measured output, select **Error Plot** from the context menu.

Compare Models Identified with Prediction and with Simulation Focus

This example shows how to identify models with prediction focus and with simulation focus. Compare the responses of prediction-focus and simulation-focus models against the original estimation data, and against validation data that was not used for estimation.

When you identify a model, the algorithm uses the `'Focus'` option to determine whether to minimize prediction error or simulation error. The default is `'prediction'`. You can change this by changing the `'Focus'` option to `'simulation'`.

Load the measurement data `z1`, and divide it into two halves `z1e` and `z1v`. One half is used for the model identification, and the other half for the model validation.

```
load iddata1 z1;
%z1e = z1(1:150); %To avoid ordqz stall
%z1v = z1(151:300);
z1e = z1(1:155);
z1v = z1(156:300);
```

Identify a third-order state-space model `sys_pf` using the input/output measurements in `z1e`. Use the default option 'Focus' option for this model, which is 'prediction'.

```
sys_pf = ssest(z1e,3);
```

`sys_pf` is a continuous-time identified state-space (`idss`) model.

Using the same set of measurement data, identify a second state-space model `sys_sf` which sets `'Focus'` to `'simulation'`.

```
opt = ssestOptions('Focus','simulation');
sys_sf = ssest(z1e,3,opt);
```

Use the `compare` function to simulate the response to both identified models.

```
figure
compare(z1e,sys_pf,sys_sf)
```
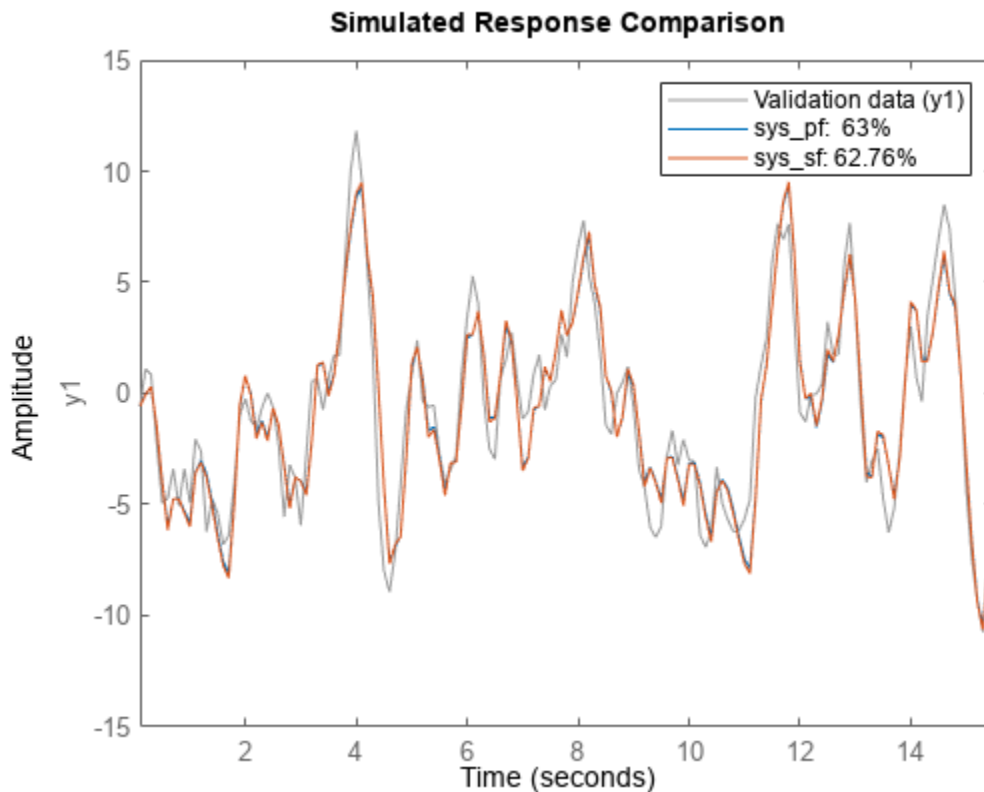

The model identified with predictive focus has a higher NRMSE fit value than the model identified with simulation focus.

Now perform a comparison against the validation data. This comparison shows how well the model performs with conditions that were not part of its identification.

```
figure
compare(z1v,sys_pf,sys_sf)
```
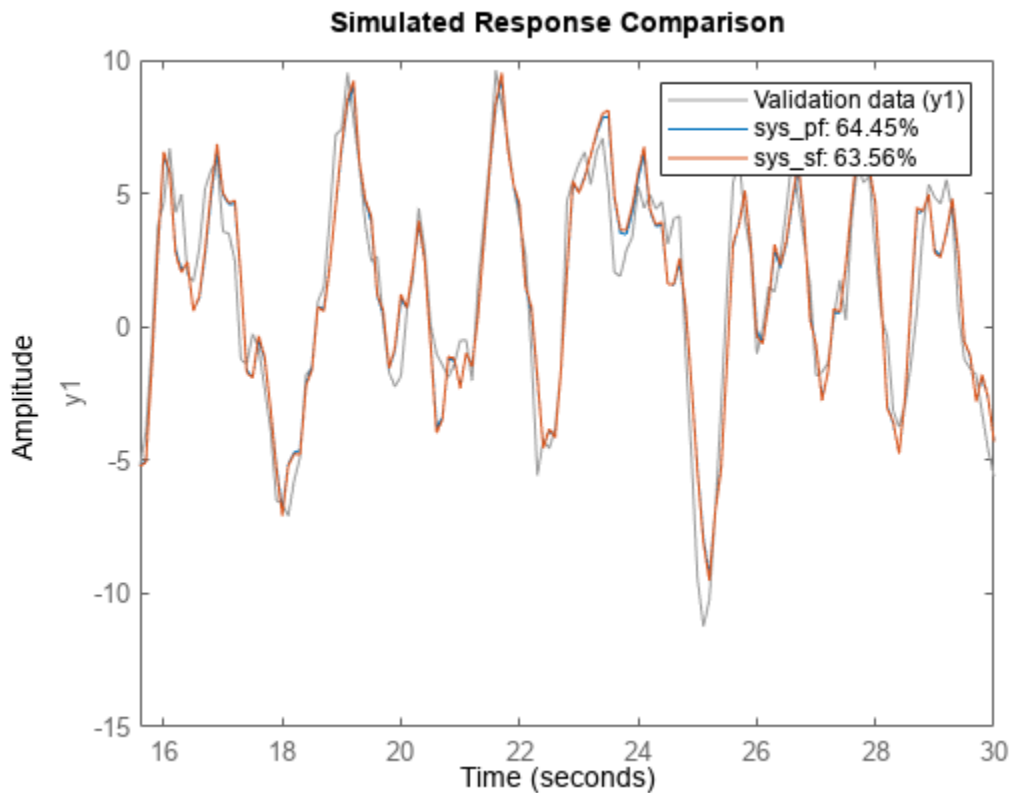
The fit values for both models improve. However, the model with prediction focus remains higher than the model with simulation focus.

See Also

`forecast` | `predict` | `sim` | `compare`

Related Examples

- "Simulation and Prediction in the App" on page 17-15
- "Simulation and Prediction at the Command Line" on page 17-19
- "Simulate Identified Model in Simulink" on page 20-5
- "Estimate Initial Conditions for Simulating Identified Models" on page 20-26

Simulation and Prediction in the App

How to Plot Simulated and Predicted Model Output

To create a model output plot for parametric linear and nonlinear models in the System Identification app, select the **Model output** check box in the **Model Views** area. By default, this operation estimates the initial states from the data and plots the output of selected models for comparison.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification app. Active models display a thick line inside the Model Board icon.

To learn how to interpret the model output plot, see "Interpret the Model Output Plot" on page 17-15.

To change plot settings, see "Change Model Output Plot Settings" on page 17-16.

For general information about creating and working with plots, see "Working with Plots" on page 21-8.

Interpret the Model Output Plot

The following figure shows a sample Model Output plot, created in the System Identification app.
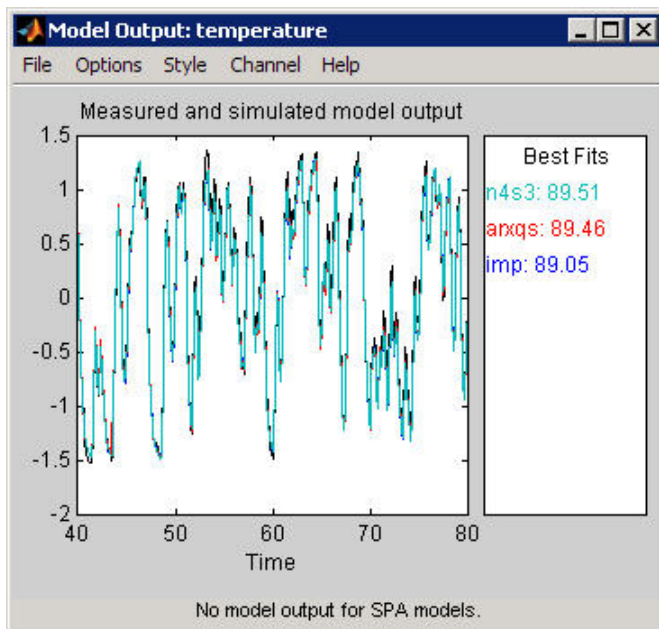
The model output plot shows different information depending on the domain of the input-output validation data, as follows:

- For time-domain validation data, the plot shows simulated or predicted model output.
- For frequency-domain data, the plot shows the amplitude of the model response to the frequency-domain input signal. The model response is equal to the product of the Fourier transform of the input and the model 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:

$$\text{Best Fit} = \left(1 - \frac{|y - \widehat{y}|}{|y - \overline{y}|}\right) \times 100$$

In this equation, y is the measured output, $\widehat{y}$ is the simulated or predicted model output, and $\overline{y}$ 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 ($\widehat{y} = \overline{y}$).

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 $|y - \widehat{y}|$. **Best Fit** can be negative when you minimized 1-step-ahead prediction during the estimation, but validate using the simulated output $\widehat{y}$.
- The validation data set was not preprocessed in the same way as the estimation data set.

Change Model Output Plot Settings

The following table summarizes the Model Output plot settings.

Model Output Plot Settings

Action	Command
Display confidence intervals. **Note** Confidence intervals are only available for simulated model output of linear models. Confidence internal are not available for nonlinear ARX and Hammerstein-Wiener models. See "Definition: Confidence Interval" on page 17-17.	• To display the dashed lines on either side of the nominal model curve, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Change between simulated output or predicted output. **Note** Prediction is only available for time-domain validation data.	• Select **Options > Simulated output** or **Options > k step ahead predicted output**. • To change the prediction horizon, select **Options > Set prediction horizon**, and select the number of samples. • To enter your own prediction horizon, select **Options > Set prediction horizon > Other**. Enter the value in terms of the number of samples.
Display the actual output values (**Signal plot**), or the difference between model output and measured output (**Error plot**).	Select **Options > Signal plot** or **Options > Error plot**.
(Time-domain validation data only) Set the time range for model output and the time interval for which the **Best Fit** value is computed.	Select **Options > Customized time span for fit** and enter the minimum and maximum time values. For example: [1 20]
(Multiple-output system only) Select a different output.	Select the output by name in the **Channel** menu.

Definition: Confidence Interval

The *confidence interval* corresponds to the range of output values with a specific probability of being the actual output of the system. The toolbox uses the estimated uncertainty in the model parameters to calculate confidence intervals and assumes the estimates have a Gaussian distribution.

For example, for a 95% confidence interval, the region around the nominal curve represents the range of values that have a 95% probability of being the true system response. You can specify the confidence interval as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution. For example, a probability of 0.99 (99%) corresponds to 2.58 standard deviations.

Note The calculation of the confidence interval assumes that the model sufficiently describes the system dynamics and the model residuals pass independence tests.

In the app, you can display a confidence interval on the plot to gain insight into the quality of a linear model. To learn how to show or hide confidence interval, see "Change Model Output Plot Settings" on page 17-16.

See Also

Related Examples

- "Simulation and Prediction at the Command Line" on page 17-19

More About

- "Simulate and Predict Identified Model Output" on page 17-6
- "Simulate Identified Model in Simulink" on page 20-5

Simulation and Prediction at the Command Line

Simulation and Prediction Commands

Note If you estimated a linear model from detrended data and want to simulate or predict the output at the original operation conditions, use `retrend` to add trend data back into the simulated or predicted output.

Command	Description	Example
`compare`	Determine how closely the simulated model response matches the measured output signal. Plots simulated or predicted output of one or more models on top of the measured output. You should use an independent validation data set as input to the model.	To plot five-step-ahead predicted output of the model `mod` against the validation data `data`, use the following command: `compare(data,mod,5)` --- **Note** Omitting the third argument assumes an infinite horizon and results in the comparison of the simulated response to the input data.
`sim`	Simulate and plot the model output only.	To simulate the response of the model `model` using input data `data`, use the following command: `sim(model,data)`

`predict`	Predict and plot the model output only.	To perform one-step-ahead prediction of the response for the model `model` and input data `data`, use the following command:
		
		`predict(model,data,1)`
		
		Use the following syntax to compute k-step-ahead prediction of the output signal using model `m`:
		
		`yhat = predict(m,[y u],k)`
		
		`predict` computes the prediction results only over the time range of `data`. It does not forecast results beyond the available data range.

Command	Description	Example
`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 Conditions 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. The `sim`, `forecast`, and `predict` commands provide options and default settings for handling initial conditions.

Simulation: Default initial conditions are zero for all model types except the `idnlgrey` model, whose default initial conditions are the internal model initial states (model property `x0`). You can specify other initial conditions using the `InitialCondition` simulation option. For more information on simulation options, 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. You can also use `compare` to return the estimated initial conditions for follow-on simulation and comparison with the same data set. These initial conditions can be in the form of an initial state vector (state-space models) or an `initialCondition` object (transfer function or polynomial models.)

If you are using `sim` to validate the quality of the identified model, you need to use the input signal from the validation data set and *also account for initial condition effects*. The simulated and the measured responses differ in the first few samples if the validation data set output contains initial condition effects that are not captured when you simulate the model. To minimize this difference, estimate the initial state values or the `initialCondition` model from the data using either `findstates` (state-space models) or `compare` (all LTI models) and specify these initial states using the `InitialCondition` simulation option (see `simOptions`). For example, 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)
```

For an example of obtaining and using `initialCondition` models, see "Apply Initial Conditions When Simulating Identified Linear Models" on page 20-45.

Prediction: Default initial conditions depend on the type of model. You can specify other initial conditions using the `InitialCondition` option (see `predictOptions`). For example, 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,IC] = predict(m,z,1,opt);
```

This command returns the estimated initial conditions as the output argument IC. For information about other ways to specify initial states, see the `predictOptions` reference page.

Simulate a Continuous-Time State-Space Model

This example shows how to simulate a continuous-time state-space model using a random binary input u and a sample time of `0.1 s`.

Consider the following state-space model:

$$\dot{x} = \begin{bmatrix} -1 & 1 \\ -0.5 & 0 \end{bmatrix} x + \begin{bmatrix} 1 \\ 0.5 \end{bmatrix} u + \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} e$$
$$y = [1 \ 0]x + e$$

where e is Gaussian white noise with variance 7.

Create a continuous-time state-space model.

```
A = [-1 1; -0.5 0];
B = [1;0.5];
C = [1 0];
D = 0;
K = [0.5;0.5];
% Ts = 0 indicates continuous time
model_ss = idss(A,B,C,D,K,'Ts',0,'NoiseVariance',7);
```

Create a random binary input.

```
u = idinput(400,'rbs',[0 0.3]);
```

Create an `iddata` object with empty output to represent just the input signal.

```
data = iddata([],u);
data.ts = 0.1;
```

Simulate the output using the model

```
opt = simOptions('AddNoise',true);
y = sim(model_ss,data,opt);
```

Simulate Model Output with Noise

This example shows how you can create input data and a model, and then use the data and the model to simulate output data.

In this example, you create the following ARMAX model with Gaussian noise e:

$$y(t) - 1.5y(t-1) + 0.7y(t-2) =$$
$$u(t-1) + 0.5u(t-2) + e(t) - e(t-1) + 0.2e(t-1)$$

Then, you simulate output data with random binary input u.

Create an ARMAX model.

```
m_armax = idpoly([1 -1.5 0.7],[0 1 0.5],[1 -1 0.2]);
```

Create a random binary input.

```
u = idinput(400,'rbs',[0 0.3]);
```

Simulate the output data.

```
opt = simOptions('AddNoise',true);
y = sim(m_armax,u,opt);
```

The `'AddNoise'` option specifies to include in the simulation the Gaussian noise `e` present in the model. Set this option to `false` (default behavior) to simulate the noise-free response to the input `u`, which is equivalent to setting `e` to zero.

See Also
`sim` | `predict` | `forecast` | `compare`

Related Examples
- "Compare Simulated Output with Measured Validation Data" on page 17-23
- "Forecast Multivariate Time Series" on page 17-25
- "Simulation and Prediction in the App" on page 17-15

More About
- "Simulate and Predict Identified Model Output" on page 17-6
- "Simulate Identified Model in Simulink" on page 20-5

Compare Simulated Output with Measured Validation Data

This example shows how to validate an estimated model by comparing the simulated model output with measured data that was not used for the original estimation.

Load the data, and divide it into sections for estimation and for validation.

```
load iddata1;
ze = z1(1:150);
zv = z1(151:300);
```

Estimate an ARMAX model, using the estimation data set `ze`.

```
m = armax(ze,[2 3 1 0]);
```

You can see how well the model performs against the original estimation data by using `compare`.

```
figure
compare(ze,m);
```
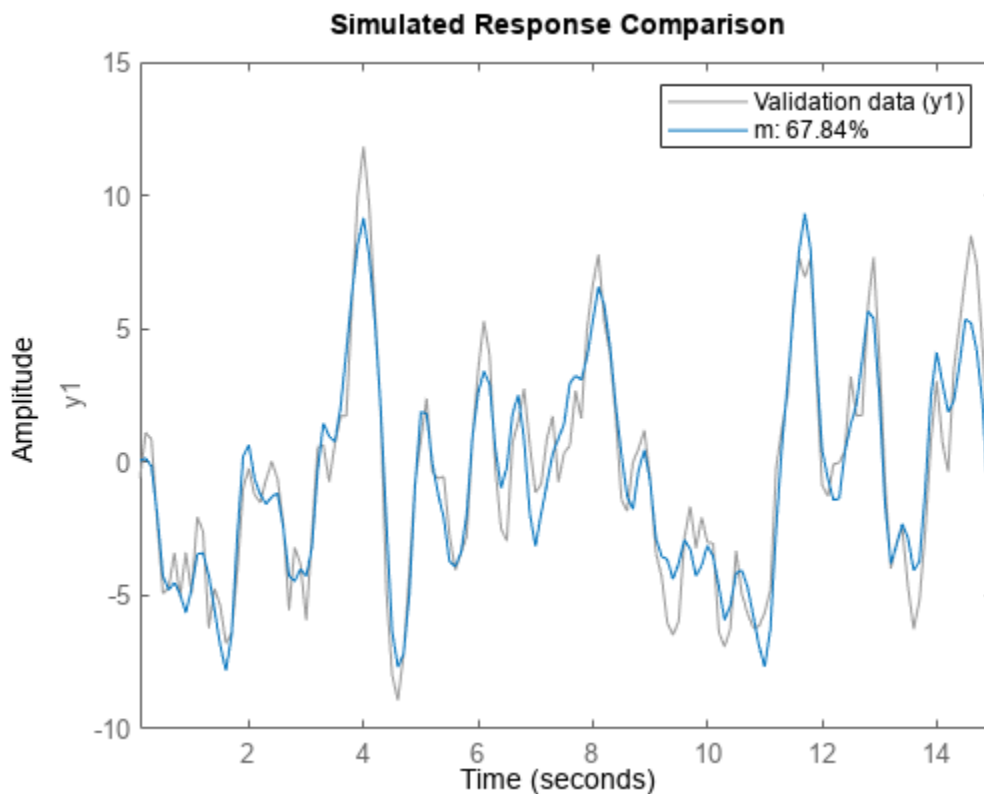

The legend shows the NRMSE fit percentage value.

Now, compare simulated model output with the measured data in the validation data set `zv`.

```
figure
compare(zv,m);
```
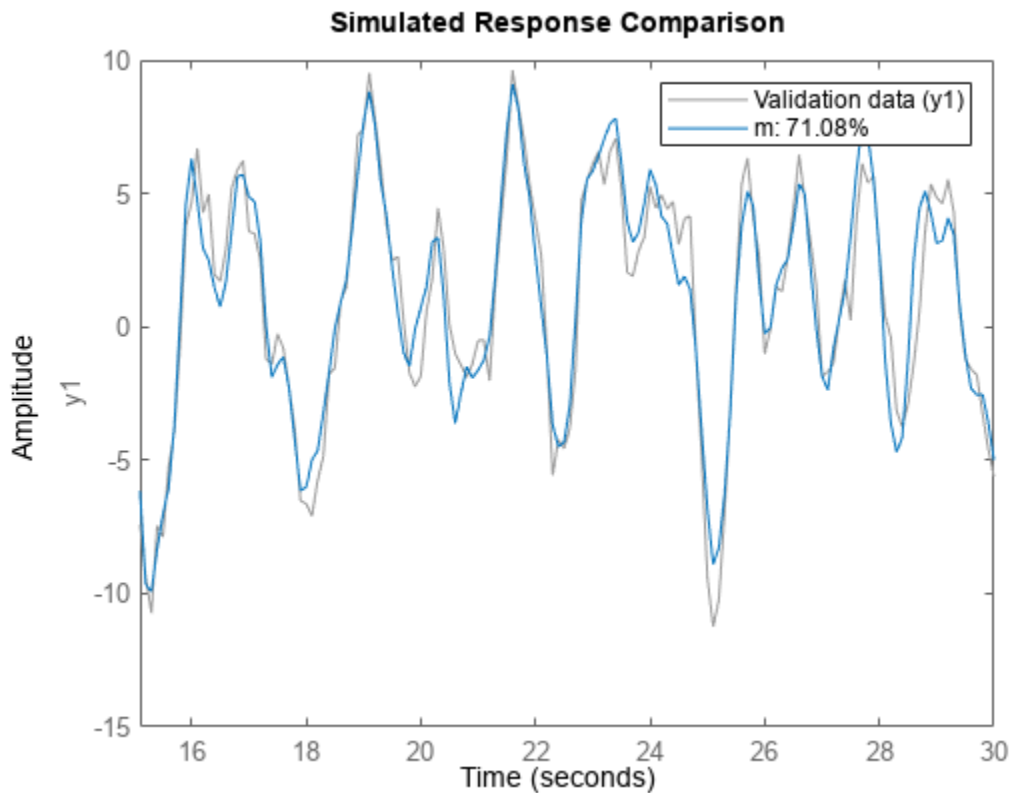
The model performs similarly against the validation data as it did against the original estimation data. It has a slightly higher fit value. This consistency is an indication of a valid model.

See Also

Related Examples
- "Simulation and Prediction at the Command Line" on page 17-19
- "Forecast Multivariate Time Series" on page 17-25

More About
- "Simulate and Predict Identified Model Output" on page 17-6

Forecast Multivariate Time Series

This example shows how to perform multivariate time series forecasting of data measured from predator and prey populations in a prey crowding scenario. The predator-prey population-change dynamics are modeled using linear and nonlinear time series models. Forecasting performance of these models is compared.

Data Description

The data is a bivariate time series consisting of 1-predator 1-prey populations (in thousands) collected 10 times a year for 20 years. For more information about the data, see "Three Ecological Population Systems: MATLAB and C MEX-File Modeling of Time-Series" on page 13-148.

Load the time series data.

```
load PredPreyCrowdingData
z = iddata(y,[],0.1,'TimeUnit','years','Tstart',0);
```

`z` is an `iddata` object containing two output signals, `y1` and `y2`, which refer to the predator and prey populations, respectively. The `OutputData` property of `z` contains the population data as a 201-by-2 matrix, such that `z.OutputData(:,1)` is the predator population and `z.OutputData(:,2)` is the prey population.

Plot the data.

```
plot(z)
title('Predator-Prey Population Data')
ylabel('Population (thousands)')
```
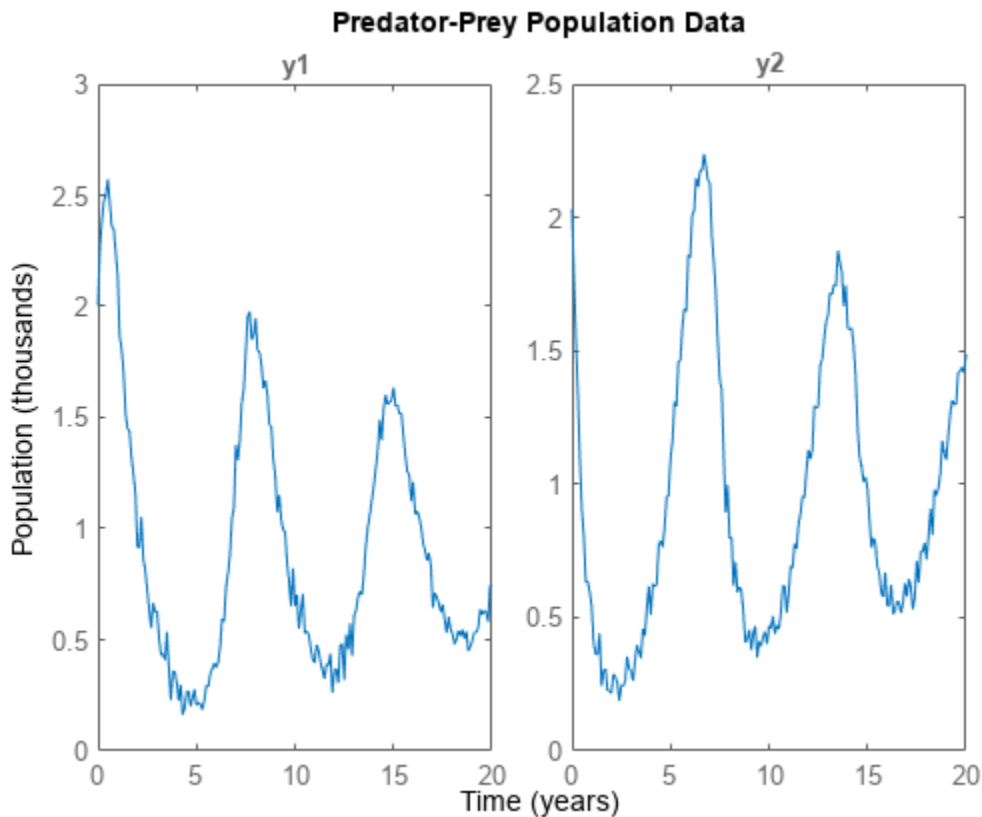
The data exhibits a decline in predator population due to crowding.

Use the first half as estimation data for identifying time series models.

```
ze = z(1:120);
```

Use the remaining data to search for model orders, and to validate the forecasting results.

```
zv = z(121:end);
```

Estimate a Linear Model

Model the time series as a linear, autoregressive process. Linear models can be created in polynomial form or state-space form using commands such as `ar` (for scalar time series only), `arx`, `armax`, `n4sid` and `ssest`. Since the linear models do not capture the data offsets (non-zero conditional mean), first detrend the data.

```
[zed, Tze] = detrend(ze, 0);
[zvd, Tzv] = detrend(zv, 0);
```

Identification requires specification of model orders. For polynomial models, you can find suitable orders using the `arxstruc` command. Since `arxstruc` works only on single-output models, perform the model order search separately for each output.

```
na_list = (1:10)';
V1 = arxstruc(zed(:,1,:),zvd(:,1,:),na_list);
na1 = selstruc(V1,0);
```

```
V2 = arxstruc(zed(:,2,:),zvd(:,2,:),na_list);
na2 = selstruc(V2,0);
```

The `arxstruc` command suggests autoregressive models of orders 7 and 8, respectively.

Use these model orders to estimate a multi-variance ARMA model where the cross terms have been chosen arbitrarily.

```
na = [na1 na1-1; na2-1 na2];
nc = [na1; na2];
sysARMA = armax(zed,[na nc])
```

```
sysARMA =
Discrete-time ARMA model:
  Model for output "y1": A(z)y_1(t) = - A_i(z)y_i(t) + C(z)e_1(t)

    A(z) = 1 - 0.885 z^-1 - 0.1493 z^-2 + 0.8089 z^-3 - 0.2661 z^-4
                                   - 0.9487 z^-5 + 0.8719 z^-6 - 0.2896 z^-7


    A_2(z) = 0.3433 z^-1 - 0.2802 z^-2 - 0.04949 z^-3 + 0.1018 z^-4
                                       - 0.02683 z^-5 - 0.2416 z^-6


    C(z) = 1 - 0.4534 z^-1 - 0.4127 z^-2 + 0.7874 z^-3 + 0.298 z^-4
                                   - 0.8684 z^-5 + 0.6106 z^-6 + 0.3616 z^-7

  Model for output "y2": A(z)y_2(t) = - A_i(z)y_i(t) + C(z)e_2(t)

    A(z) = 1 - 0.5826 z^-1 - 0.4688 z^-2 - 0.5949 z^-3 - 0.0547 z^-4
                         + 0.5062 z^-5 + 0.4024 z^-6 - 0.01544 z^-7 - 0.1766 z^-8


    A_1(z) = 0.2386 z^-1 + 0.1564 z^-2 - 0.2249 z^-3 - 0.2638 z^-4 - 0.1019 z^-5
                                       - 0.07821 z^-6 + 0.2982 z^-7


    C(z) = 1 - 0.1717 z^-1 - 0.09877 z^-2 - 0.5289 z^-3 - 0.24 z^-4
                         + 0.06555 z^-5 + 0.2217 z^-6 - 0.05765 z^-7 - 0.1824 z^-8

Sample time: 0.1 years

Parameterization:
   Polynomial orders:    na=[7 6;7 8]    nc=[7;8]
   Number of free coefficients: 43
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using ARMAX on time domain data "zed".
Fit to estimation data: [89.85;90.97]% (prediction focus)
FPE: 3.814e-05, MSE: 0.007533
```

Compute a 10-step-ahead (1 year) predicted output to validate the model over the time span of the estimation data. Since the data was detrended for estimation, you need to specify those offsets for meaningful predictions.

```
predOpt = predictOptions('OutputOffset',Tze.OutputOffset');
yhat1 = predict(sysARMA,ze,10, predOpt);
```

The `predict` command predicts the response over the time span of measured data and is a tool for validating the quality of an estimated model. The response at time `t` is computed using measured values at times t = 0, ..., t-10.

Plot the predicted response and the measured data.

```
plot(ze,yhat1)
title('10-step predicted response compared to measured data')
```
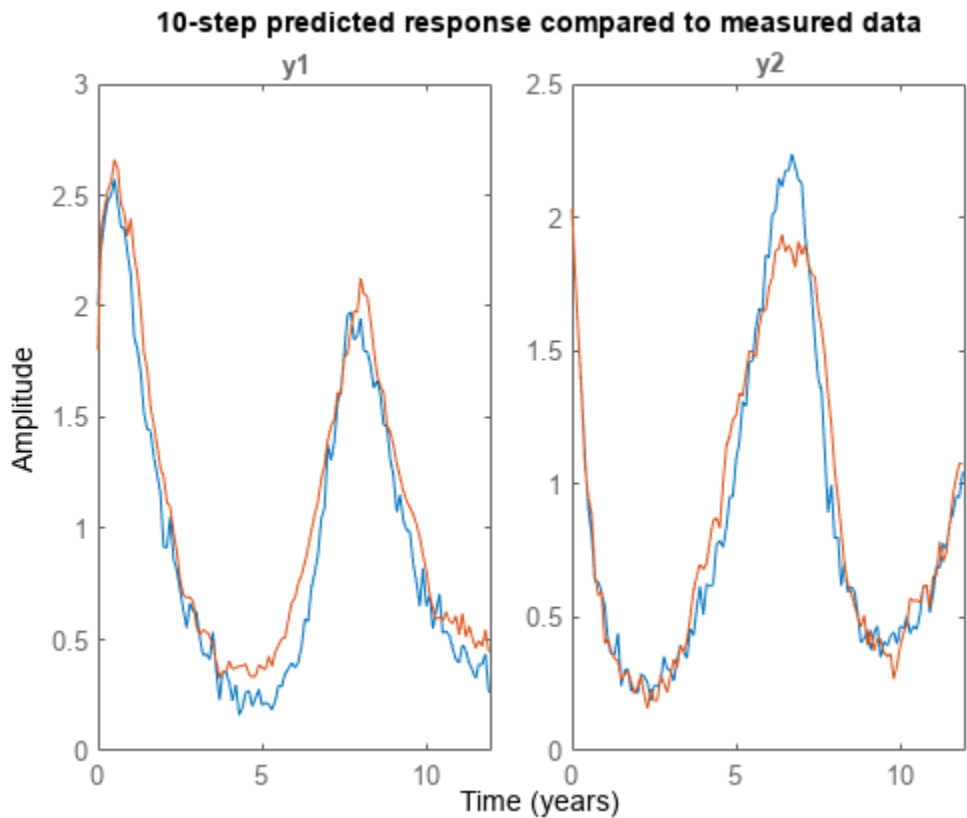

Note, the generation of predicted response and plotting it with the measured data, can be automated using the `compare` command.

```
compareOpt = compareOptions('OutputOffset',Tze.OutputOffset');
compare(ze,sysARMA,10,compareOpt)
```
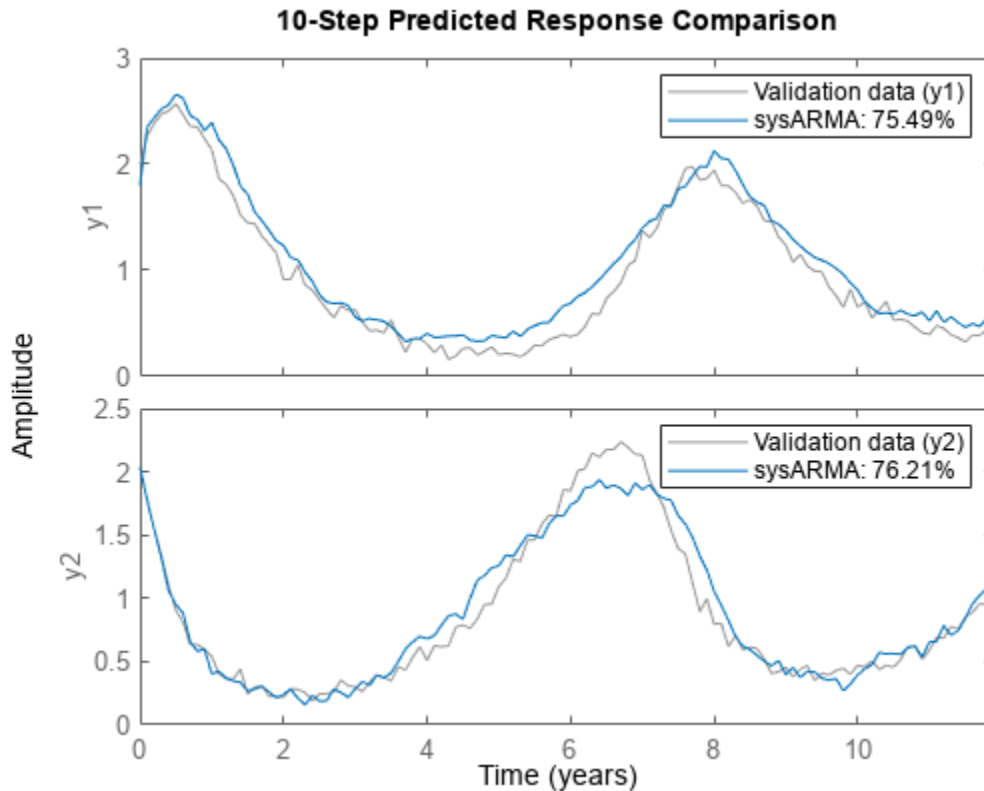
The plot generated using `compare` also shows the normalized root mean square (NRMSE) measure of goodness of fit in percent form.

After validating the data, forecast the output of the model `sysARMA` 100 steps (10 years) beyond the estimation data, and calculate output standard deviations.

```
forecastOpt = forecastOptions('OutputOffset',Tze.OutputOffset');
[yf1,x01,sysf1,ysd1] = forecast(sysARMA, ze, 100, forecastOpt);
```

`yf1` is the forecasted response, returned as an `iddata` object. `yf1.OutputData` contains the forecasted values.

`sysf1` is a system similar to `sysARMA` but is in state-space form. Simulation of `sysf1` using the `sim` command, with initial conditions, `x01`, reproduces the forecasted response, `yf1`.

`ysd1` is the matrix of standard deviations. It measures the uncertainty is forecasting owing to the effect of additive disturbances in the data (as measured by `sysARMA.NoiseVariance`), parameter uncertainty (as reported by `getcov(sysARMA)`) and uncertainties associated with the process of mapping past data to the initial conditions required for forecasting (see `data2state`).

Plot the measured, predicted, and forecasted output for model `sysARMA`.

```
t = yf1.SamplingInstants;
te = ze.SamplingInstants;
t0 = z.SamplingInstants;
subplot(1,2,1);
plot(t0,z.y(:,1),...
```

```
        te,yhat1.y(:,1),...
        t,yf1.y(:,1),'m',...
        t,yf1.y(:,1)+ysd1(:,1),'k--', ...
        t,yf1.y(:,1)-ysd1(:,1), 'k--')
    xlabel('Time (year)');
    ylabel('Predator population, in thousands');
    subplot(1,2,2);
    plot(t0,z.y(:,2),...
        te,yhat1.y(:,2),...
        t,yf1.y(:,2),'m',...
        t,yf1.y(:,2)+ysd1(:,2),'k--', ...
        t,yf1.y(:,2)-ysd1(:,2),'k--')
    % Make the figure larger.
    fig = gcf;
    p = fig.Position;
    fig.Position = [p(1),p(2)-p(4)*0.2,p(3)*1.4,p(4)*1.2];
    xlabel('Time (year)');
    ylabel('Prey population, in thousands');
    legend({'Measured','Predicted','Forecasted','Forecast Uncertainty (1 sd)'},...
        'Location','best')
```
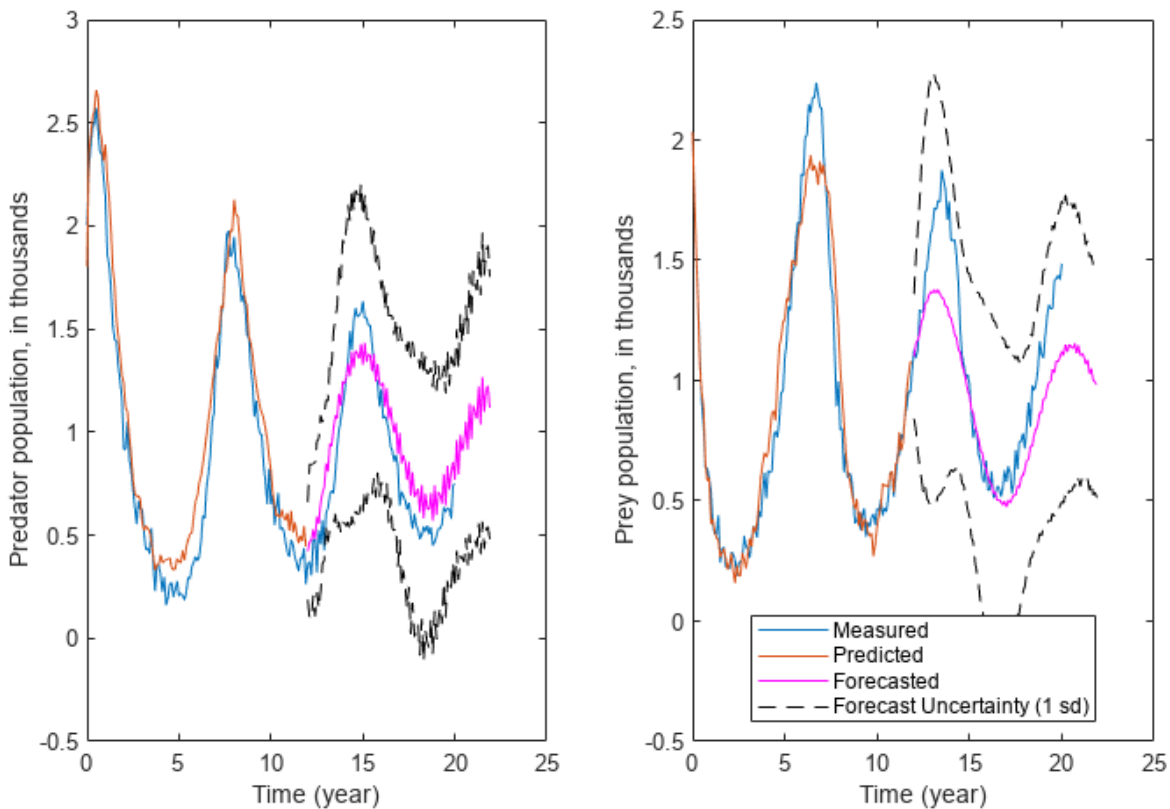

The plots show that forecasting using a linear ARMA model (with added handling of offsets) worked somewhat and the results showed high uncertainty compared to the actual populations over the 12-20 years time span. This indicates that the population change dynamics might be nonlinear.

Estimate Nonlinear Black-Box Models

In this section we extend the black-box identification approach of the previous section by adding nonlinear elements to the model. This is carried out using the Nonlinear ARX modeling framework. Nonlinear ARX models are conceptually similar to the linear ones (such as `sysARMA` above) in that they compute the predicted output using a two-step process:

1 Map the measured signals to a finite-dimensional regressor space.

2 Map the regressors to the predicted output using a static function which can be linear or nonlinear.
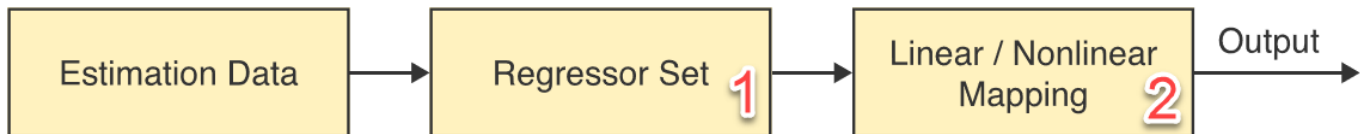
Nonlinearities can be added in two places - either in the regressor set (step 1), and/or in the static mapping function (step 2). In this section we explore both of these options. First, we create a linear-in-regressor form of the Nonlinear ARX model where in all the nonlinearities are limited to the regressor definition only; the static mapping function that transforms the regressors into the predicted output is linear (affine, to be exact). Then, we create a model that uses Gaussian Process function as its static mapping function, but keeps its regressors linear.

Polynomial AR Model

Create a nonlinear ARX model with 2 outputs and no inputs.

```
L = linearRegressor(ze.OutputName,{1,1});
sysNLARX = idnlarx(ze.OutputName, ze.InputName, L, []);
sysNLARX.Ts = 0.1;
sysNLARX.TimeUnit = 'years';
```

`sysNLARX` is a first order nonlinear ARX model that uses no nonlinear function; it predicts the model response using a weighted sum of its first-order regressors.

```
getreg(sysNLARX)
```

```
ans = 2x1 cell
    {'y1(t-1)'}
    {'y2(t-1)'}
```

To introduce a nonlinearity function, add polynomial regressors to the model.

Create regressors of order 2 and include cross terms (products of standard regressors listed above). Add those regressors to the model.

```
P = polynomialRegressor(ze.OutputName,{1,1},2,false,true);
sysNLARX.Regressors(end+1) = P;
getreg(sysNLARX)
```

```
ans = 5x1 cell
    {'y1(t-1)'        }
    {'y2(t-1)'        }
    {'y1(t-1)^2'      }
    {'y2(t-1)^2'      }
```

```
    {'y1(t-1)*y2(t-1)'}
```

Estimate the coefficients (the regressor weightings and the offset) of the model using estimation data, ze.

```
sysNLARX = nlarx(ze,sysNLARX)

sysNLARX =

Nonlinear multivariate time series model with 2 outputs
  Outputs: y1, y2

Regressors:
  1. Linear regressors in variables y1, y2
  2. Order 2 regressors in variables y1, y2

Output functions:
  Output 1:  Linear with offset
  Output 2:  Linear with offset

Sample time: 0.1 years

Status:
Termination condition: Near (local) minimum, (norm(g) < tol)..
Number of iterations: 0, Number of function evaluations: 1

Estimated using NLARX on time domain data "ze".
Fit to estimation data: [88.34;88.91]% (prediction focus)
FPE: 3.265e-05, MSE: 0.01048
More information in model's "Report" property.
```

Compute a 10-step-ahead predicted output to validate the model.

```
yhat2 = predict(sysNLARX,ze,10);
```

Forecast the output of the model 100 steps beyond the estimation data.

```
yf2 = forecast(sysNLARX,ze,100);
```

The standard deviations of the forecasted response are not computed for nonlinear ARX models. This data is unavailable because the parameter covariance information is not computed during estimation of these models.

Plot the measured, predicted, and forecasted outputs.

```
t = yf2.SamplingInstants;
subplot(1,2,1);
plot(t0,z.y(:,1),...
    te,yhat2.y(:,1),...
    t,yf2.y(:,1),'m')
xlabel('Time (year)');
ylabel('Predator population (thousands)');
title('Prediction Results for ARX Model with Polynomial Regressors')
subplot(1,2,2);
plot(t0,z.y(:,2),...
    te,yhat2.y(:,2),...
    t,yf2.y(:,2),'m')
```

```
legend('Measured','Predicted','Forecasted')
xlabel('Time (year)');
ylabel('Prey population (thousands)');
```
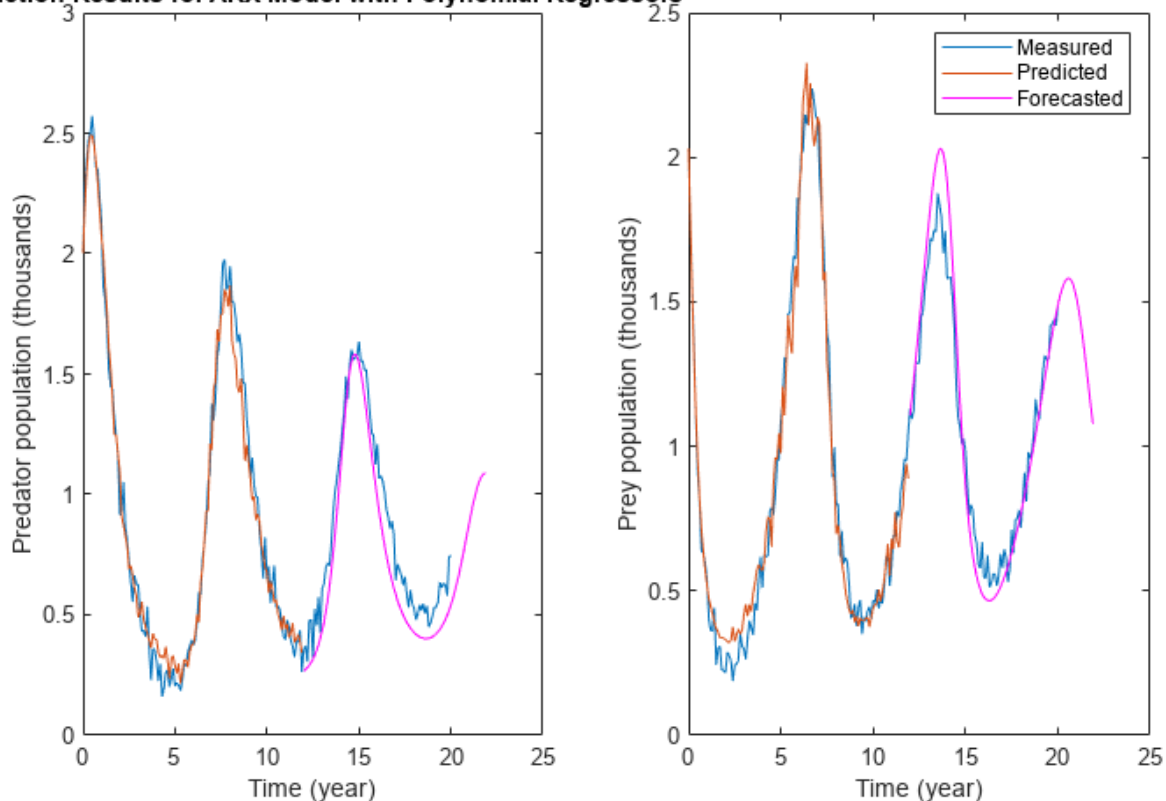

The plots show that forecasting using a nonlinear ARX model gave better forecasting results than using a linear model. Nonlinear black-box modeling did not require prior knowledge about the equations governing the data.

Note that to reduce the number of regressors, you can pick an optimal subset of (transformed) variables using principal component analysis (see `pca`) or feature selection (see `sequentialfs`) in the Statistics and Machine Learning Toolbox™.

Estimate a Gaussian Process Model

The model The model `sysNLARX` uses a linear mapping function; the nonlinearity is contained in the regressors only. A more flexible approach involves picking a non-trivial mapping function, such as a Gaussian Process function (GP). Create a nonlinear ARX model that uses only the linear regressors but adds a Gaussian process function for mapping the regressors to the output.

```
% Copy sysNLARX so we don't have to create the model structure again
sysGP = sysNLARX;
% Remove polynomial regressor set
sysGP.Regressors(2) = [];
% Replace the linear mapping function by a Gaussian Process (GP) function.
% The GP uses a squared exponential kernel by default.
```

```
GP = idGaussianProcess;
sysGP.OutputFcn = GP;
disp(sysGP)

  2x0 idnlarx array with properties:

         Regressors: [1x1 linearRegressor]
          OutputFcn: [2x1 idGaussianProcess]
      RegressorUsage: [2x4 table]
       Normalization: [1x1 struct]
        TimeVariable: 't'
        NoiseVariance: [2x2 double]
           InputName: {0x1 cell}
           InputUnit: {0x1 cell}
          InputGroup: [1x1 struct]
          OutputName: {2x1 cell}
          OutputUnit: {2x1 cell}
         OutputGroup: [1x1 struct]
               Notes: [0x1 string]
            UserData: []
                Name: ''
                  Ts: 0.1000
            TimeUnit: 'years'
              Report: [1x1 idresults.nlarxhw]
```

You now have a template model sysGP that is based on linear regressors and a GP mapping function. Its parameters are the two coefficients of its squared exponential kernel. Use `nlarx` to estimate these parameters.

```
sysGP = nlarx(ze, sysGP)

sysGP =

Nonlinear multivariate time series model with 2 outputs
  Outputs: y1, y2

Regressors:
  Linear regressors in variables y1, y2

Output functions:
  Output 1:  Gaussian process function using a SquaredExponential kernel
  Output 2:  Gaussian process function using a SquaredExponential kernel

Sample time: 0.1 years

Status:
Estimated using NLARX on time domain data "ze".
Fit to estimation data: [88.07;88.84]% (prediction focus)
FPE: 3.369e-05, MSE: 0.01083
More information in model's "Report" property.
```

```
getpvec(sysGP)

ans = 10×1

    -0.6671
     0.0196
    -0.0000
    -0.0571
```

```
-3.6447
-0.0221
-0.5752
 0.0000
 0.1725
-3.1068
```

As before, compute the predicted and forecasted responses and plot results.

Compute a 10-step-ahead predicted output to validate the model.

```
yhat3 = predict(sysGP,ze,10);
```

Forecast the output of the model 100 steps beyond the estimation data.

```
yf3 = forecast(sysGP,ze,100);
```

Plot the measured, predicted, and forecasted outputs.

```
t = yf3.SamplingInstants;
subplot(1,2,1);
plot(t0,z.y(:,1),...
   te,yhat3.y(:,1),...
   t,yf3.y(:,1),'m')
xlabel('Time (year)');
ylabel('Predator population (thousands)');
title('Prediction Results for Gaussian Process Model')
subplot(1,2,2);
plot(t0,z.y(:,2),...
   te,yhat3.y(:,2),...
   t,yf3.y(:,2),'m')
legend('Measured','Predicted','Forecasted')
xlabel('Time (year)');
ylabel('Prey population (thousands)');
```

Prediction Results for Gaussian Process Model
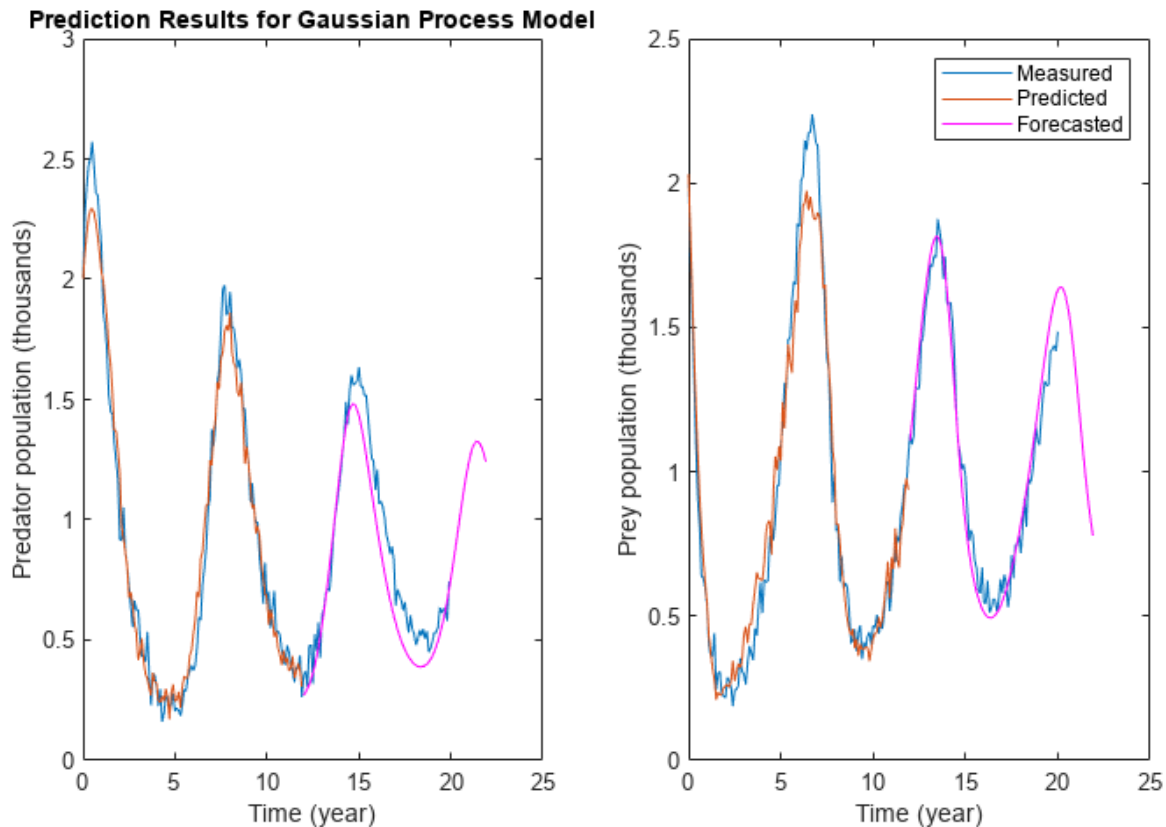
The results show improved accuracy over the linear model. An advantage of using GP based models is that they employ very few parameters, as opposed to other forms such as Wavelet Networks (`idWaveletNetwork`) or Sigmoid Networks (`idSigmoidNetwork`). This makes them easy to train with low uncertainty in their parameter estimates.

Estimate a Nonlinear Grey-Box Model

If you have prior knowledge of the system dynamics, you can fit the estimation data using a nonlinear grey-box model. Knowledge about the nature of the dynamics can, in general, help improve the model quality and thus the forecasting accuracy. For the predator-prey dynamics, the changes in the predator (`y1`) and prey (`y2`) population can be represented as:

$$\dot{y}_1(t) = p_1 * y_1(t) + p_2 * y_2(t) * y_1(t)$$

$$\dot{y}_2(t) = p_3 * y_2(t) - p_4 * y_1(t) * y_2(t) - p_5 * y_2(t)^2$$

For more information about the equations, see "Three Ecological Population Systems: MATLAB and C MEX-File Modeling of Time-Series" on page 13-148.

Construct a nonlinear grey-box model based on these equations.

Specify a file describing the model structure for the predator-prey system. The file specifies the state derivatives and model outputs as a function of time, states, inputs, and model parameters. The two

outputs (predator and prey populations) are chosen as states to derive a nonlinear state-space description of the dynamics.

```
FileName = 'predprey2_m';
```

Specify the model orders (number of outputs, inputs, and states)

```
Order = [2 0 2];
```

Specify the initial values for the parameters p_1, p_2, p_3, p_4, and p_5, and indicate that all parameters are to be estimated. Note that the requirement to specify initial guesses for parameters did not exist when estimating the black box models sysARMA and sysNLARX.

```
Parameters = struct('Name',{'Survival factor, predators' 'Death factor, predators' ...
    'Survival factor, preys' 'Death factor, preys' ...
    'Crowding factor, preys'}, ...
    'Unit',{'1/year' '1/year' '1/year' '1/year' '1/year'}, ...
    'Value',{-1.1 0.9 1.1 0.9 0.2}, ...
    'Minimum',{-Inf -Inf -Inf -Inf -Inf}, ...
    'Maximum',{Inf Inf Inf Inf Inf}, ...
    'Fixed',{false false false false false});
```

Similarly, specify the initial states of the model, and indicate that both initial states are to be estimated.

```
InitialStates = struct('Name',{'Predator population' 'Prey population'}, ...
    'Unit',{'Size (thousands)' 'Size (thousands)'}, ...
    'Value',{1.8 1.8}, ...
    'Minimum', {0 0}, ...
    'Maximum',{Inf Inf}, ...
    'Fixed',{false false});
```

Specify the model as a continuous-time system.

```
Ts = 0;
```

Create a nonlinear grey-box model with specified structure, parameters, and states.

```
sysGrey = idnlgrey(FileName,Order,Parameters,InitialStates,Ts,'TimeUnit','years');
```

Estimate the model parameters.

```
sysGrey = nlgreyest(ze,sysGrey);
present(sysGrey)
```

```
sysGreyGrey =

Continuous-time nonlinear grey-box model defined by 'predprey2_m' (MATLAB file):

    dx/dt = F(t, x(t), p1, ..., p5)
     y(t) = H(t, x(t), p1, ..., p5) + e(t)

 with 0 input(s), 2 state(s), 2 output(s), and 5 free parameter(s) (out of 5).

 States:                                    Initial value
    x(1)  Predator population(t) [Size (thou..]    xinit@exp1   2.01325   (estimated) in [0, Inf]
    x(2)  Prey population(t) [Size (thou..]        xinit@exp1   1.99687   (estimated) in [0, Inf]
```

```
  Outputs:
     y(1)  y1(t)
     y(2)  y2(t)
   Parameters:                               Value  Standard Deviation
     p1   Survival factor, predators [1/year]   -0.995895      0.0125269   (estimated) in [-Inf
     p2   Death factor, predators [1/year]       1.00441       0.0129368   (estimated) in [-Inf
     p3   Survival factor, preys [1/year]        1.01234       0.0135413   (estimated) in [-Inf
     p4   Death factor, preys [1/year]           1.01909       0.0121026   (estimated) in [-Inf
     p5   Crowding factor, preys [1/year]        0.103244      0.0039285   (estimated) in [-Inf

  Status:
  Termination condition: Change in cost was less than the specified tolerance..
  Number of iterations: 6, Number of function evaluations: 7

  Estimated using Solver: ode45; Search: lsqnonlin on time domain data "ze".
  Fit to estimation data: [91.21;92.07]%
  FPE: 8.613e-06, MSE: 0.005713
  More information in model's "Report" property.

  Model Properties
```

Compute a 10-step-ahead predicted output to validate the model.

```
yhat4 = predict(sysGrey,ze,10);
```

Forecast the output of the model 100 steps beyond the estimation data, and calculate output standard deviations.

```
[yf4,x04,sysf4,ysd4] = forecast(sysGrey,ze,100);
```

Plot the measured, predicted, and forecasted outputs.

```
t = yf4.SamplingInstants;
subplot(1,2,1);
plot(t0,z.y(:,1),...
   te,yhat4.y(:,1),...
   t,yf4.y(:,1),'m',...
   t,yf4.y(:,1)+ysd4(:,1),'k--', ...
   t,yf4.y(:,1)-ysd4(:,1),'k--')
xlabel('Time (year)');
ylabel('Predator population (thousands)');
title('Prediction Results for Grey-Box Model')
subplot(1,2,2);
plot(t0,z.y(:,2),...
   te,yhat4.y(:,2),...
   t,yf4.y(:,2),'m',...
   t,yf4.y(:,2)+ysd4(:,2),'k--', ...
   t,yf4.y(:,2)-ysd4(:,2),'k--')

legend('Measured','Predicted','Forecasted','Forecast uncertainty (1 sd)')
xlabel('Time (years)');
ylabel('Prey population (thousands)');
```
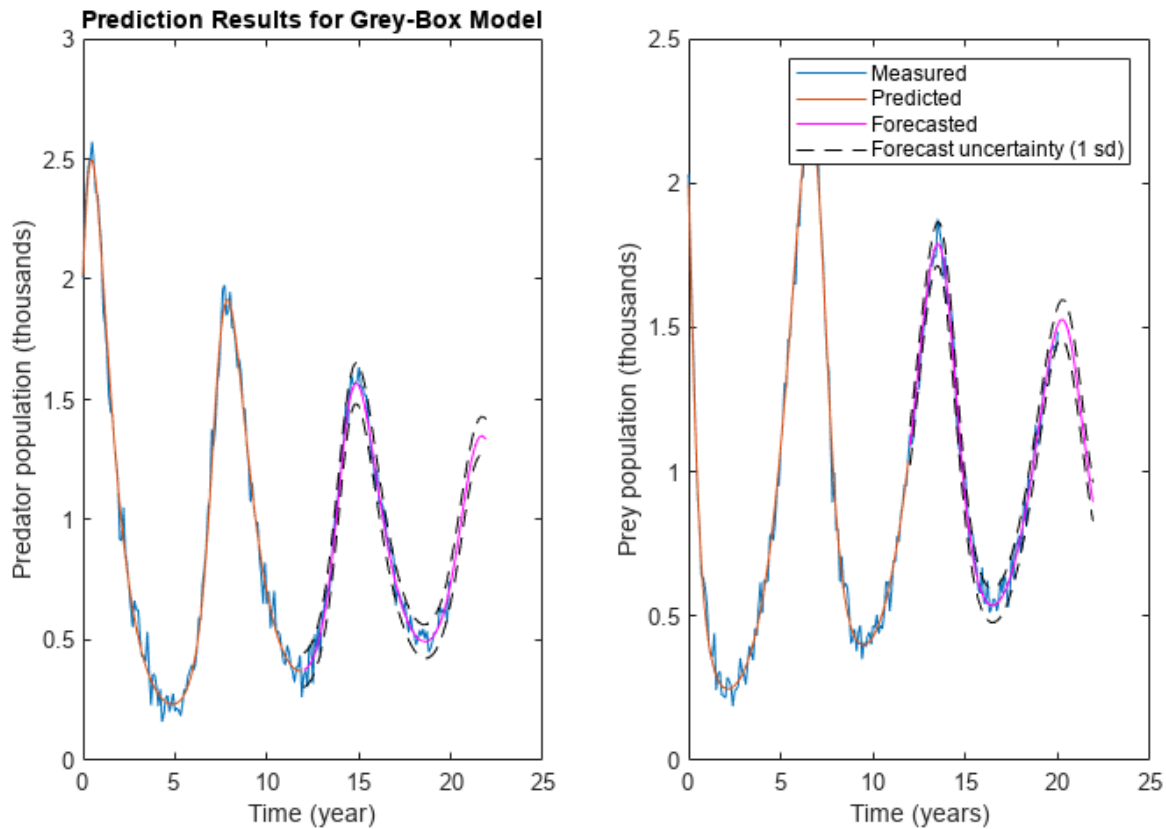
The plots show that forecasting using a nonlinear grey-box model gave good forecasting results and low forecasting output uncertainty.

Compare Forecasting Performance

Compare the forecasted response obtained from the identified models, `sysARMA`, `sysNLARX`, and `sysGrey`. The first two are discrete-time models and `sysGrey` is a continuous-time model.

```
clf
plot(z,yf1,yf2,yf3,yf4)
legend({'Measured','ARMA','Polynomial AR','GP','Grey-Box'})
title('Forecasted Responses')
xlim([7 23])
```

Forecasted Responses
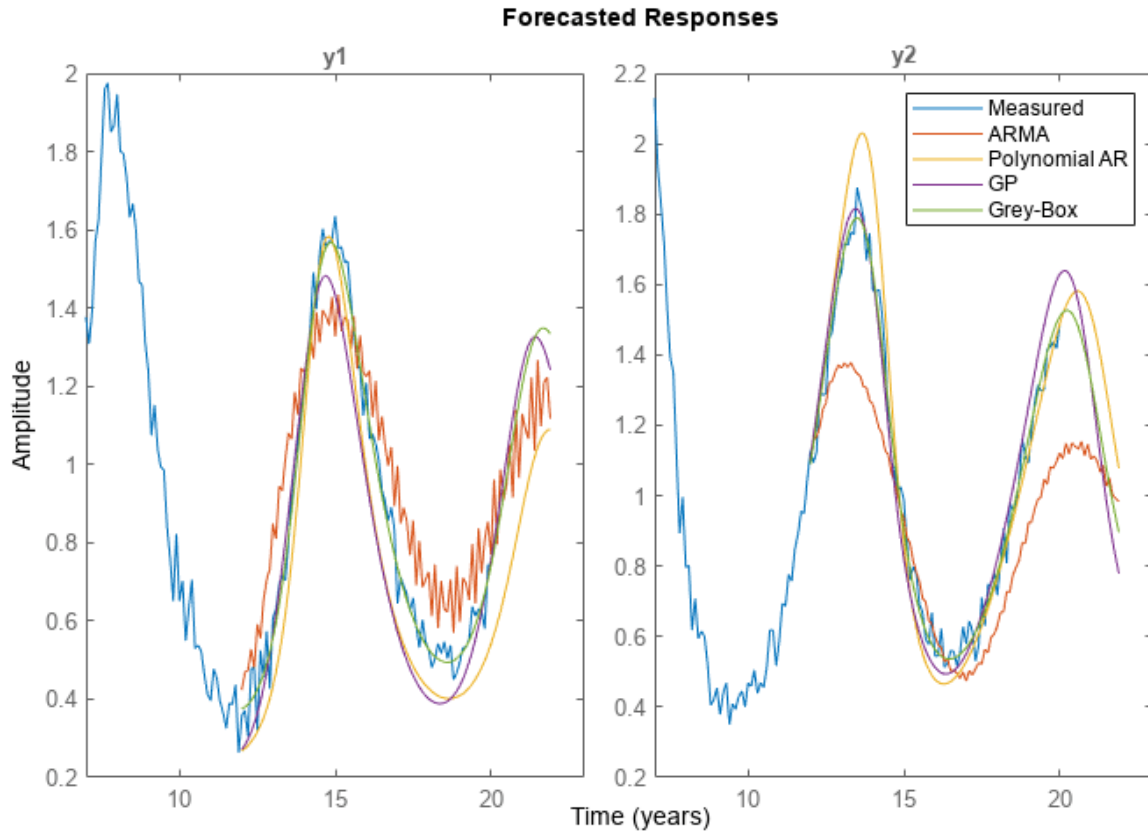
Forecasting with a nonlinear ARX model gave better results than forecasting with a linear model. Inclusion of the knowledge of the dynamics in the nonlinear grey-box model further improved the reliability of the model and therefore the forecasting accuracy.

Note that the equations used in grey-box modeling are closely related to the polynomial regressors used by the Nonlinear ARX model. If you approximate the derivatives in the governing equations by first-order differences, you will get equations similar to:

$$y_1(t) = s_1 * y_1(t-1) + s_2 * y_1(t-1) * y_2(t-1)$$

$$y_2(t) = s_3 * y_2(t-1) - s_4 * y_1(t-1) * y_2(t-1) - s_5 * y_2(t-1)^2$$

Where, $s_1, \ldots, s_5$ are some parameters related to the original parameters $p_1, \ldots, p_5$ and to the sample time used for differencing. These equations suggest that only 2 regressors are needed for the first output (y1(t-1) and *y1(t-1)*y2(t-1)) and 3 for the second output when constructing the Nonlinear ARX model. Even in absence of such prior knowledge, linear-in-regressor model structures employing polynomial regressors remain a popular choice in practice.

Forecast the values using the nonlinear grey-box model over 200 years.

```
[yf5,~,~,ysd5] = forecast(sysGrey, ze, 2000);
```

Plot the latter part of the data (showing 1 sd uncertainty)

```
t = yf5.SamplingInstants;
N = 700:2000;
subplot(1,2,1);
plot(t(N), yf5.y(N,1), 'm',...
    t(N), yf5.y(N,1)+ysd5(N,1), 'k--', ...
    t(N), yf5.y(N,1)-ysd5(N,1), 'k--')
xlabel('Time (year)');
ylabel('Predator population (thousands)');
title('Forecasted Population after 200 Years')
ax = gca;
ax.YLim = [0.8 1];
ax.XLim = [82 212];
subplot(1,2,2);
plot(t(N),yf5.y(N,2),'m',...
    t(N),yf5.y(N,2)+ysd5(N,2),'k--', ...
    t(N),yf5.y(N,2)-ysd5(N,2),'k--')
legend('Forecasted population','Forecast uncertainty (1 sd)')
xlabel('Time (years)');
ylabel('Prey population (thousands)');
ax = gca;
ax.YLim = [0.9 1.1];
ax.XLim = [82 212];
```
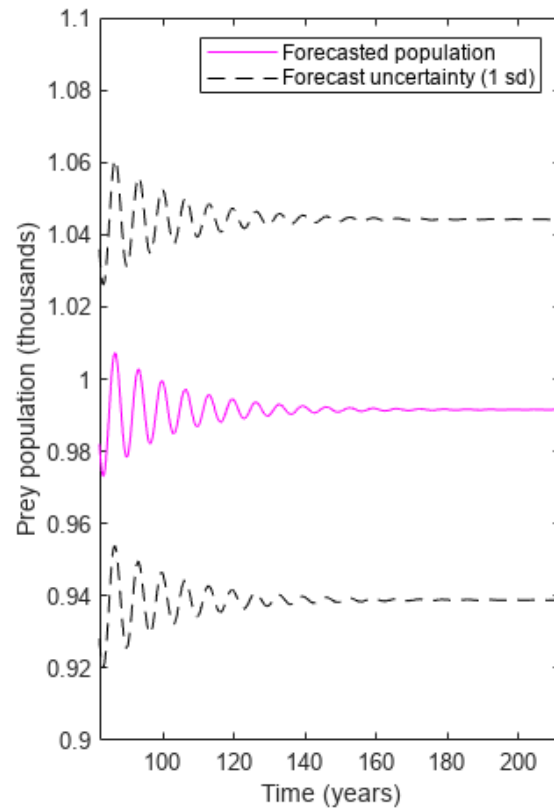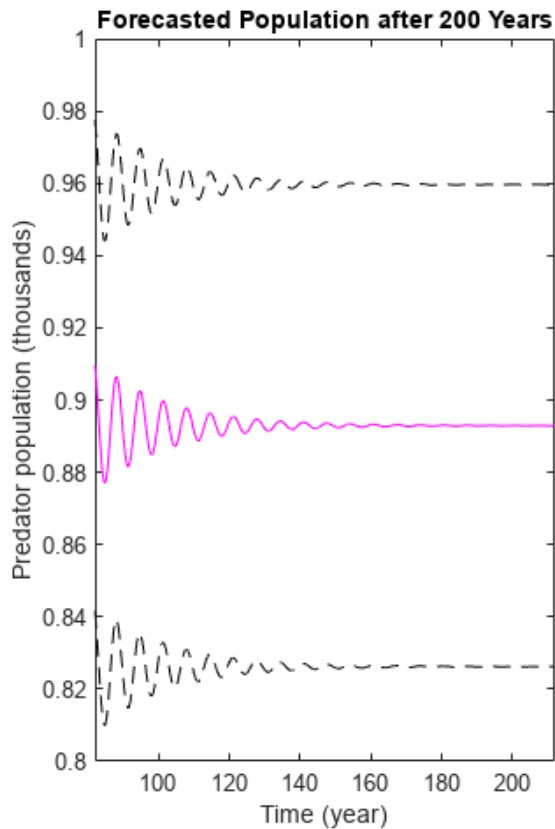
The plot show that the predatory population is forecasted to reach a steady-state of approximately 890 and the prey population is forecasted to reach 990.

See Also

Related Examples

- "Introduction to Forecasting of Dynamic System Response" on page 14-34
- "Simulation and Prediction at the Command Line" on page 17-19
- "Identify Time Series Models at the Command Line" on page 14-12

What Is Residual Analysis?

Residuals are differences between the one-step-predicted output from the model and the measured output from the validation data set. Thus, residuals represent the portion of the validation data not explained by the model.

Residual analysis consists of two tests: the whiteness test and the independence test.

According to the *whiteness test* criteria, a good model has the residual autocorrelation function inside the confidence interval of the corresponding estimates, indicating that the residuals are uncorrelated.

According to the *independence test* criteria, a good model has residuals uncorrelated with past inputs. Evidence of correlation indicates that the model does not describe how part of the output relates to the corresponding input. For example, a peak outside the confidence interval for lag *k* means that the output *y(t)* that originates from the input *u(t-k)* is not properly described by the model.

Your model should pass both the whiteness and the independence tests, except in the following cases:

- For output-error (OE) models and when using instrumental-variable (IV) methods, make sure that your model shows independence of e and u, and pay less attention to the results of the whiteness of e.

  In this case, the modeling focus is on the dynamics *G* and not the disturbance properties *H*.

- Correlation between residuals and input for negative lags, is not necessarily an indication of an inaccurate model.

  When current residuals at time *t* affect future input values, there might be feedback in your system. In the case of feedback, concentrate on the positive lags in the cross-correlation plot during model validation.

Supported Model Types

You can validate parametric linear and nonlinear models by checking the behavior of the model residuals. For a description of residual analysis, see "What Residual Plots Show for Different Data Domains" on page 17-43.

Note Residual analysis plots are not available for frequency response (FRD) models. For time-series models, you can only generate model-output plots for parametric models using time-domain time-series (no input) measured data.

What Residual Plots Show for Different Data Domains

Residual analysis plots show different information depending on whether you use time-domain or frequency-domain input-output validation data.

For time-domain validation data, the plot shows the following two axes:

- Autocorrelation function of the residuals for each output

- Cross-correlation between the input and the residuals for each input-output pair

> **Note** For time-series models, the residual analysis plot does not provide any input-residual correlation plots.

For frequency-domain validation data, the plot shows the following two axes:

- Estimated power spectrum of the residuals for each output
- Transfer-function amplitude from the input to the residuals for each input-output pair

For linear models, you can estimate a model using time-domain data, and then validate the model using frequency domain data. For nonlinear models, the System Identification Toolbox product supports only time-domain data.

The following figure shows a sample Residual Analysis plot, created in the System Identification app.
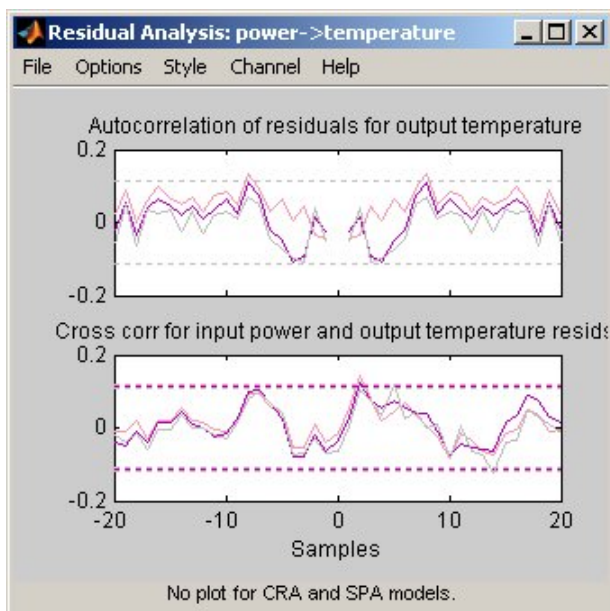
Displaying the Confidence Interval

The *confidence interval* corresponds to the range of residual values with a specific probability of being statistically insignificant for the system. The toolbox uses the estimated uncertainty in the model parameters to calculate confidence intervals and assumes the estimates have a Gaussian distribution.

For example, for a 95% confidence interval, the region around zero represents the range of residual values that have a 95% probability of being statistically insignificant. You can specify the confidence interval as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution. For example, a probability of 0.99 (99%) corresponds to 2.58 standard deviations.

You can display a confidence interval on the plot in the app to gain insight into the quality of the model. To learn how to show or hide confidence interval, see the description of the plot settings in "How to Plot Residuals in the App" on page 17-46.

Note If you are working in the System Identification app, you can specify a custom confidence interval. If you are using the `resid` command, the confidence interval is fixed at 99%.

See Also

Related Examples

- "How to Plot Residuals in the App" on page 17-46
- "Examine Model Residuals" on page 17-48

How to Plot Residuals in the App

To create a residual analysis plot for parametric linear and nonlinear models in the System Identification app, select the **Model resids** check box in the **Model Views** area. For general information about creating and working with plots, see "Working with Plots" on page 21-8.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification app. Active models display a thick line inside the Model Board icon.

The following table summarizes the Residual Analysis plot settings.

Residual Analysis Plot Settings

Action	Command
Display confidence intervals around zero. **Note** Confidence internal are not available for nonlinear ARX and Hammerstein-Wiener models.	• To display the dashed lines on either side of the nominal model curve, select **Options** > **Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level** and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Change the number of lags (data samples) for which to compute autocorrelation and cross-correlation functions. **Note** For frequency-domain validation data, increasing the number of lags increases the frequency resolution of the residual spectrum and the transfer function.	• Select **Options** > **Number of lags** and choose the value from the list. • To enter your own lag value, select **Options > Set confidence level > Other**. Enter the value as the number of data samples.
(Multiple-output system only) Select a different input-output pair.	Select the input-output by name in the **Channel** menu.

See Also

Related Examples

More About

How to Plot Residuals at the Command Line

The following table summarizes commands that generate residual-analysis plots for linear and nonlinear models. For detailed information about this command, see the corresponding reference page.

Note Apply `pe` and `resid` to one model at a time.

Command	Description	Example
pe	Computes and plots model prediction errors.	To plot the prediction errors for the model `model` using data `data`, type the following command: `pe(model,data)`
resid	Performs whiteness and independence tests on model residuals, or prediction errors. Uses validation data input as model input.	To plot residual correlations for the model `model` using data `data`, type the following command: `resid(data,model)`

See Also

Related Examples

- "How to Plot Residuals in the App" on page 17-46
- "Examine Model Residuals" on page 17-48

More About

- "What Is Residual Analysis?" on page 17-43

Examine Model Residuals

This example shows how you can use residual analysis to evaluate model quality.

Creating Residual Plots

1 To load the sample System Identification app session that contains estimated models, type the following command in the MATLAB Command Window:

```
systemIdentification('dryer2_linear_models')
```

2 To generate a residual analysis plot, select the **Model resids** check box in the System Identification app.

This opens an empty plot.

3 In the System Identification app window, click each parametric model icon to display it on the Residual Analysis plot. Do not click the nonparametric model `spad`, as residual analysis plots are not available for this model.
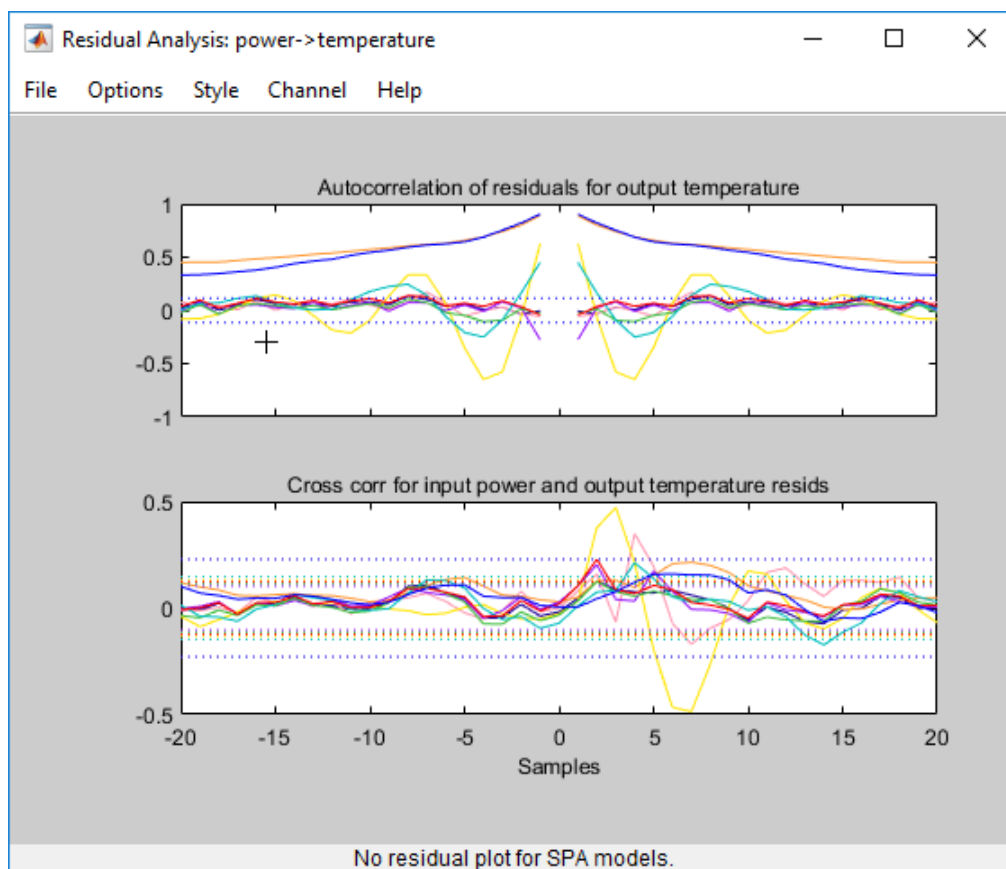
Description of the Residual Plot Axes

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. Six of the models —`arxqs`, `n4s3`, `arx223`, `tf1`,`ss1`, 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`, `tf1`, `ss1`, and `amx2222` in the System Identification app.

The Residual Analysis plot now includes only the two models that pass the residual tests: `arx692` and `amx3322`.
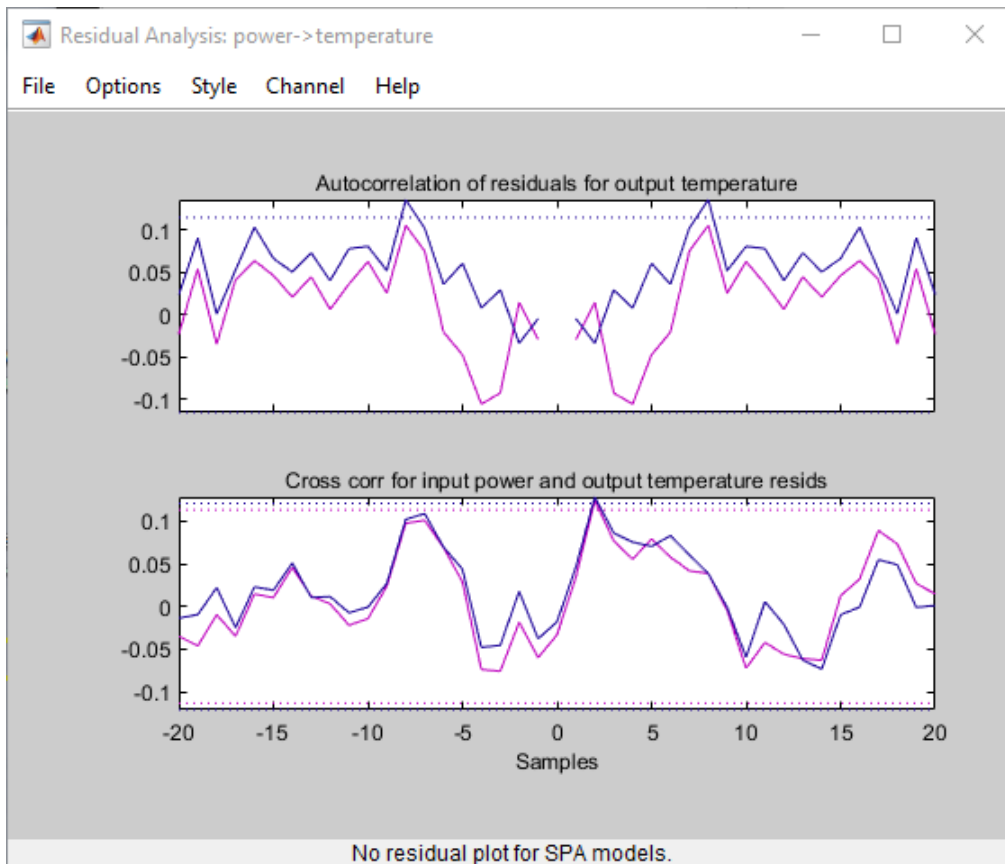
The plots for these models fall primarily within the confidence intervals, although the `amx3322` plot shows some excursions slightly outside of the boundaries. If these excursions are acceptable, it is reasonable to pick the `amx3322` model because it is a simpler low-order model. If the excursions are not acceptable, you can use the `arx392` model instead.

See Also

Related Examples

- "How to Plot Residuals in the App" on page 17-46
- "How to Plot Residuals at the Command Line" on page 17-47

More About

- "What Is Residual Analysis?" on page 17-43

Impulse and Step Response Plots

Supported Models

You can plot the simulated response of a model using impulse and step signals as the input for all linear parametric models and correlation analysis (nonparametric) models.

You can also create step-response plots for nonlinear models. These step and impulse response plots, also called *transient response* plots, provide insight into the characteristics of model dynamics, including peak response and settling time.

Note For frequency-response models, impulse- and step-response plots are not available. For nonlinear models, only step-response plots are available.

Examples

"Plot Impulse and Step Response Using the System Identification App" on page 17-53

"Plot Impulse and Step Response at the Command Line" on page 17-56

How Transient Response Helps to Validate Models

Transient response plots provide insight into the basic dynamic properties of the model, such as response times, static gains, and delays.

Transient response plots also help you validate how well a linear parametric model, such as a linear ARX model or a state-space model, captures the dynamics. For example, you can estimate an impulse or step response from the data using correlation analysis (nonparametric model), and then plot the correlation analysis result on top of the transient responses of the parametric models.

Because nonparametric and parametric models are derived using different algorithms, agreement between these models increases confidence in the parametric model results.

What Does a Transient Response Plot Show?

Transient response plots show the value of the impulse or step response on the vertical axis. The horizontal axis is in units of time you specified for the data used to estimate the model.

The impulse response of a dynamic model is the output signal that results when the input is an impulse. That is, $u(t)$ is zero for all values of t except at $t=0$, where $u(0)=1$. In the following difference equation, you can compute the impulse response by setting $y(-T)=y(-2T)=0$, $u(0)=1$, and $u(t>0)=0$.

$$y(t) - 1.5y(t - T) + 0.7y(t - 2T) =$$
$$0.9u(t) + 0.5u(t - T)$$

The step response is the output signal that results from a step input, where $u(t<0)=0$ and $u(t>0)=1$.

If your model includes a noise model, you can display the transient response of the noise model associated with each output channel. For more information about how to display the transient response of the noise model, see "Plot Impulse and Step Response Using the System Identification App" on page 17-53.

The following figure shows a sample Transient Response plot, created in the System Identification app.
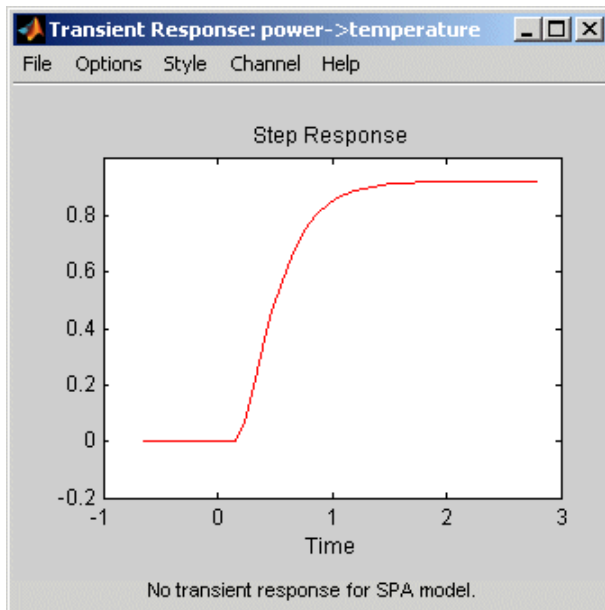
Displaying the Confidence Interval

In addition to the transient-response curve, you can display a confidence interval on the plot. To learn how to show or hide confidence interval, see the description of the plot settings in "Plot Impulse and Step Response Using the System Identification App" on page 17-53.

The *confidence interval* corresponds to the range of response values with a specific probability of being the actual response of the system. The toolbox uses the estimated uncertainty in the model parameters to calculate confidence intervals and assumes the estimates have a Gaussian distribution.

For example, for a 95% confidence interval, the region around the nominal curve represents the range where there is a 95% chance that it contains the true system response. You can specify the confidence interval as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution. For example, a probability of 0.99 (99%) corresponds to 2.58 standard deviations.

Note The calculation of the confidence interval assumes that the model sufficiently describes the system dynamics and the model residuals pass independence tests.

Plot Impulse and Step Response Using the System Identification App

To create a transient analysis plot in the System Identification app, select the **Transient resp** check box in the **Model Views** area. For general information about creating and working with plots, see "Working with Plots" on page 21-8.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification app. Active models display a thick line inside the Model Board icon.

The following table summarizes the Transient Response plot settings.

Transient Response Plot Settings

Action	Command
Display step response for linear or nonlinear model.	Select **Options > Step response**.
Display impulse response for linear model.	Select **Options > Impulse response**. **Note** Not available for nonlinear models.
Display the confidence interval. **Note** Only available for linear models.	• To display the dashed lines on either side of the nominal model curve, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Change time span over which the impulse or step response is calculated. For a scalar time span *T*, the resulting response is plotted from *-T/4* to *T*. **Note** To change the time span of models you estimated using correlation analysis models, select **Estimate > Correlation models** and reestimate the model using a new time span.	• Select **Options > Time span (time units)**, and choose a new time span in units of time you specified for the model. • To enter your own time span, select **Options > Time span (time units) > Other**, and enter the total response duration. • To use the time span based on model dynamics, type [] or `default`. The `default` time span is computed based on the model dynamics and might be different for different models. For nonlinear models, the default time span is 10.
Toggle between line plot or stem plot. **Tip** Use a stem plot for displaying impulse response.	Select **Style > Line plot** or **Style > Stem plot**.
(Multiple-output system only) Select an input-output pair to view the noise spectrum corresponding to those channels.	Select the output by name in the **Channel** menu. If the plotted models include a noise model, you can display the transient response properties associated with each output channel. The name of the channel has the format e@OutputName, where OutputName is the name of the output channel corresponding to the noise model.

Action	Command
(Step response for nonlinear models only) Set level of the input step. **Note** For multiple-input models, the input-step level applies only to the input channel you selected to display in the plot.	Select **Options** > **Step Size**, and then chose from two options: • **0->1** sets the lower level to 0 and the upper level to 1. • **Other** opens the Step Level dialog box, where you enter the values for the lower and upper level values.

More About

"Impulse and Step Response Plots" on page 17-51

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 17-51

Frequency Response Plots for Model Validation

Frequency response plots of a linear model provide insight into the characteristics of the model dynamics, including the frequency of the peak response and stability margins. You can use frequency response plots can help to validate how well a linear parametric model captures the dynamics. The System Identification Toolbox provides multiple plotting options.

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 $G(i\omega)$ is the transfer function evaluated on the imaginary axis $s = i\omega$.

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 $Ge^{i\omega T}$ is the transfer function $G(z)$ evaluated on the unit circle. The argument of the frequency function $Ge^{i\omega T}$ is scaled by the sample time T to make the frequency function periodic with the sampling frequency $2\pi/T$.

Use Frequency Response to Help Validate Models

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.

You can use frequency response plots can help to validate how well a linear parametric model, such as a linear ARX mode, a state-space model, or a frequency response model, captures the dynamics. For example, you can estimate a frequency response from the estimation 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. For an example of using spectral analysis to compare models, see "Identify Time Series Models at the Command Line" on page 14-12.

Options for Frequency-Response Plotting

The System Identification Toolbox provides three frequency-response plotting options.

- Linear input-output models and frequency-response data models

  - bode—Plot the magnitude and phase of the frequency response on a logarithmic frequency scale. 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.

    In the **System Identification** app, use the **Frequency resp** option in **Model Views** to create a Bode plot.

  - nyquist— Plot the imaginary versus real parts of the transfer function. If your model is in a form other than transfer function, nyquist first converts the model to a transfer function. You can obtain Nyquist plots only at the command line.

- All linear models, including time series (no input) models:

  - spectrum—Plot the output power spectrum of a time series model or the disturbance spectrum of a linear input-output model. The disturbance spectrum is also called the *noise spectrum*. This plot is the same as a Bode plot of the model response, but it shows the output power spectrum of the noise model instead. For more information, see "Noise Spectrum Plots" on page 17-64.

    In the **System Identification** app, use the **Noise spectrum** option in **Model Views** to create a power spectrum plot.

The figure shows the three types of frequency plot, all generated at the command line.
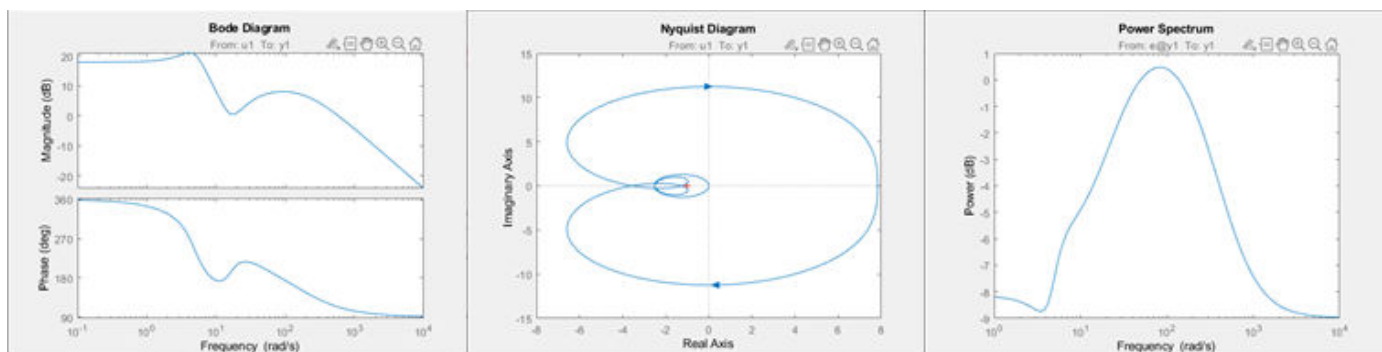
If you want to customize the appearance of the plot or turn on the confidence region programmatically, use bodeplot, nyquistplot, or spectrumplot.

For examples and detailed information about each command, see the corresponding reference pages. For information about obtaining frequency response in the **System Identification** app, see "Identify Linear Models Using System Identification App", "Plot Bode Plots Using the System Identification App" on page 17-60, and "Plot the Noise Spectrum Using the System Identification App" on page 17-66.

Display the Confidence Interval

In addition to the frequency-response curve, you can display the region that represents the confidence interval on the plot.

- To display the confidence interval for a plot generated at the command line, right-click in the plot and select `Characteristics > Confidence region`. By default, the plot displays the region for one standard deviation, or 95%. To change the interval, right-click on the plot and select `Properties`. Then, in the **Options** tab of the Property Editor, modify **Number of standard deviations to display**.

- To display the confidence interval in the app, select `Options > Show 97% confidence intervals`, or for a different size interval, select from `Options > Set Confidence level`.

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 instance, a probability of 0.99 (99%) corresponds to 2.58 standard deviations.

The figure shows an example of the confidence region for a bode plot generated at the command line.
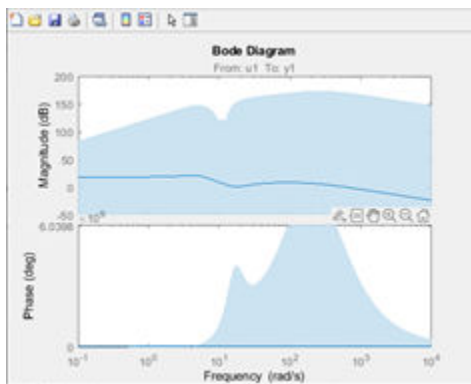
See Also
spectrumplot | spectrum | nyquistplot | nyquist | bodeplot | bode

More About
- "Plot Bode Plots Using the System Identification App" on page 17-60
- "Identify Linear Models Using System Identification App"
- "Plot the Noise Spectrum Using the System Identification App" on page 17-66
- "Identify Time Series Models at the Command Line" on page 14-12

Plot Bode Plots Using the System Identification App

To create a frequency-response plot for linear models in the System Identification app, select the **Frequency resp** check box in the **Model Views** area. For general information about creating and working with plots, see "Working with Plots" on page 21-8.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification app. Active models display a thick line inside the Model Board icon.

The following table summarizes the Frequency Function plot settings.

Frequency Function Plot Settings

Action	Command
Display the confidence interval.	• To display the dashed lines on either side of the nominal model curve, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Change the frequency values for computing the noise spectrum. The default frequency vector is 128 linearly distributed values, greater than zero and less than or equal to the Nyquist frequency.	Select **Options > Frequency range** and specify a new frequency vector in units of rad/s. Enter the frequency vector using any one of following methods: • MATLAB expression, such as `[1:100]*pi/100` or `logspace(-3,-1,200)`. Cannot contain variables in the MATLAB workspace. • Row vector of values, such as `[1:.1:100]` **Note** To restore the default frequency vector, enter `[]`.
Change frequency units between hertz and radians per second.	Select **Style > Frequency (Hz)** or **Style > Frequency (rad/s)**.
Change frequency scale between linear and logarithmic.	Select **Style > Linear frequency scale** or **Style > Log frequency scale**.
Change amplitude scale between linear and logarithmic.	Select **Style > Linear amplitude scale** or **Style > Log amplitude scale**.
(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 for Model Validation" on page 17-57

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 for Model Validation" on page 17-57

Noise Spectrum Plots

Supported Models

When you estimate the noise model of your linear system, you can plot the spectrum of the estimated noise model. Noise-spectrum plots are available for all linear parametric models and spectral analysis (nonparametric) models.

Note For nonlinear models and correlation analysis models, noise-spectrum plots are not available. For time-series models, you can only generate noise-spectrum plots for parametric and spectral-analysis models.

Examples

"Plot the Noise Spectrum Using the System Identification App" on page 17-66

"Plot the Noise Spectrum at the Command Line" on page 17-68

What Does a Noise Spectrum Plot Show?

The general equation of a linear dynamic system is given by:

$$y(t) = G(z)u(t) + v(t)$$

In this equation, G is an operator that takes the input to the output and captures the system dynamics, and v is the additive noise term. The toolbox treats the noise term as filtered white noise, as follows:

$$v(t) = H(z)e(t)$$

where $e(t)$ is a white-noise source with variance λ.

The toolbox computes both H and λ during the estimation of the noise model and stores these quantities as model properties. The $H(z)$ operator represents the noise model.

Whereas the frequency-response plot shows the response of G, the noise-spectrum plot shows the frequency-response of the noise model H.

For input-output models, the noise spectrum is given by the following equation:

$$\Phi_v(\omega) = \lambda \left| H\left(e^{i\omega}\right) \right|^2$$

For time-series models (no input), the vertical axis of the noise-spectrum plot is the same as the dynamic model spectrum. These axes are the same because there is no input for time series and $y = He$.

Note You can avoid estimating the noise model by selecting the Output-Error model structure or by setting the `DisturbanceModel` property value to `'None'` for a state space model. If you choose to not estimate a noise model for your system, then H and the noise spectrum amplitude are equal to 1 at all frequencies.

Displaying the Confidence Interval

In addition to the noise-spectrum curve, you can display a confidence interval on the plot. To learn how to show or hide confidence interval, see the description of the plot settings in "Plot the Noise Spectrum Using the System Identification App" on page 17-66.

The *confidence interval* corresponds to the range of power-spectrum values with a specific probability of being the actual noise spectrum of the system. The toolbox uses the estimated uncertainty in the model parameters to calculate confidence intervals and assumes the estimates have a Gaussian distribution.

For example, for a 95% confidence interval, the region around the nominal curve represents the range where there is a 95% chance that the true response belongs.. You can specify the confidence interval as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution. For example, a probability of 0.99 (99%) corresponds to 2.58 standard deviations.

Note The calculation of the confidence interval assumes that the model sufficiently describes the system dynamics and the model residuals pass independence tests.

Plot the Noise Spectrum Using the System Identification App

To create a noise spectrum plot for parametric linear models in the app, select the **Noise spectrum** check box in the **Model Views** area. For general information about creating and working with plots, see "Working with Plots" on page 21-8.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification app. Active models display a thick line inside the Model Board icon.

The following figure shows a sample Noise Spectrum plot.
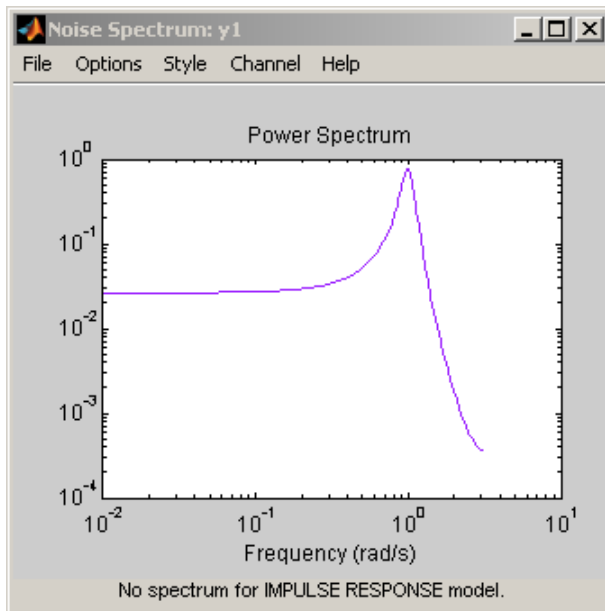
The following table summarizes the Noise Spectrum plot settings.

Noise Spectrum Plot Settings

Action	Command
Display the confidence interval.	• To display the dashed lines on either side of the nominal model curve, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Change the frequency values for computing the noise spectrum. The default frequency vector is 128 linearly distributed values, greater than zero and less than or equal to the Nyquist frequency.	Select **Options > Frequency range** and specify a new frequency vector in units of radians per second. Enter the frequency vector using any one of following methods: • MATLAB expression, such as `[1:100]*pi/100` or `logspace(-3,-1,200)`. Cannot contain variables in the MATLAB workspace. • Row vector of values, such as `[1:.1:100]` **Tip** To restore the default frequency vector, enter `[]`.
Change frequency units between hertz and radians per second.	Select **Style > Frequency (Hz)** or **Style > Frequency (rad/s)**.
Change frequency scale between linear and logarithmic.	Select **Style > Linear frequency scale** or **Style > Log frequency scale**.
Change amplitude scale between linear and logarithmic.	Select **Style > Linear amplitude scale** or **Style > Log amplitude scale**.
(Multiple-output system only) Select an input-output pair to view the noise spectrum corresponding to those channels. **Note** You cannot view cross spectra between different outputs.	Select the output by name in the **Channel** menu.

More About

"Noise Spectrum Plots" on page 17-64

Plot the Noise Spectrum at the Command Line

To plot the disturbance spectrum of an input-output model or the output spectrum of a time series model, use `spectrum`. To customize such plots, or to turn on the confidence region view programmatically for such plots, use `spectrumplot` instead.

To determine if your estimated noise model is good enough, you can compare the output spectrum of the estimated noise-model *H* to the estimated output spectrum of *v(t)*. To compute *v(t)*, which represents the actual noise term in the system, use the following commands:

```
ysimulated = sim(m,data);
v = ymeasured-ysimulated;
```

`ymeasured` is `data.y`. `v` is the noise term *v(t)*, as described in "What Does a Noise Spectrum Plot Show?" on page 17-64 and corresponds to the difference between the simulated response `ysimulated` and the actual response `ymeasured`.

To compute the frequency-response model of the actual noise, use `spa`:

```
V = spa(v);
```

The toolbox uses the following equation to compute the noise spectrum of the actual noise:

$$\Phi_v(\omega) = \sum_{\tau = -\infty}^{\infty} R_v(\tau)e^{-i\omega\tau}$$

The covariance function R_v is given in terms of *E*, which denotes the mathematical expectation, as follows:

$$R_v(\tau) = Ev(t)v(t - \tau)$$

To compare the parametric noise-model *H* to the (nonparametric) frequency-response estimate of the actual noise *v(t)*, use `spectrum`:

```
spectrum(V,m)
```

If the parametric and the nonparametric estimates of the noise spectra are different, then you might need a higher-order noise model.

More About

"Noise Spectrum Plots" on page 17-64

Pole and Zero Plots

Supported Models

You can create pole-zero plots of linear identified models. To study the poles and zeros of the noise component of an input-output model or a time series model, use `noise2meas` to first extract the noise model as an independent input-output model, whose inputs are the noise channels of the original model.

For examples of creating pole-zero plots, see "Model Poles and Zeros Using the System Identification App" on page 17-72 and "Plot Poles and Zeros at the Command Line" on page 17-73.

What Does a Pole-Zero Plot Show?

The following figure shows a sample pole-zero plot of the model with confidence intervals. x indicate poles and o indicate zeros.
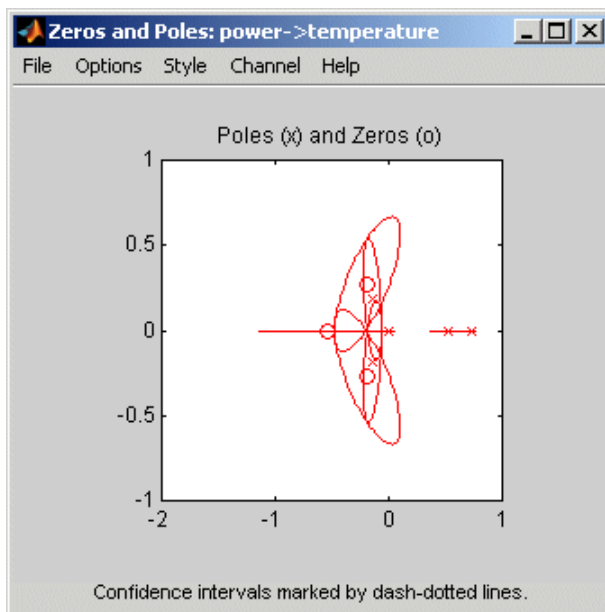
The general equation of a linear dynamic system is given by:

$$y(t) = G(z)u(t) + v(t)$$

In this equation, G is an operator that takes the input to the output and captures the system dynamics, and v is the additive noise term.

The *poles* of a linear system are the roots of the denominator of the transfer function G. The poles have a direct influence on the dynamic properties of the system. The *zeros* are the roots of the numerator of G. If you estimated a noise model H in addition to the dynamic model G, you can also view the poles and zeros of the noise model.

Zeros and the poles are equivalent ways of describing the coefficients of a linear difference equation, such as the ARX model. Poles are associated with the output side of the difference equation, and zeros are associated with the input side of the equation. The number of poles is equal to the number

of sampling intervals between the most-delayed and least-delayed output. The number of zeros is equal to the number of sampling intervals between the most-delayed and least-delayed input. For example, there two poles and one zero in the following ARX model:

$$y(t) - 1.5y(t - T) + 0.7y(t - 2T) =$$
$$0.9u(t) + 0.5u(t - T)$$

Displaying the Confidence Interval

You can display a confidence interval for each pole and zero on the plot. To learn how to show or hide confidence interval, see "Model Poles and Zeros Using the System Identification App" on page 17-72 and "Plot Poles and Zeros at the Command Line" on page 17-73.

The *confidence interval* corresponds to the range of pole or zero values with a specific probability of being the actual pole or zero of the system. The toolbox uses the estimated uncertainty in the model parameters to calculate confidence intervals and assumes the estimates have a Gaussian distribution.

For example, for a 95% confidence interval, the region around the nominal pole or zero value represents the range of values that have a 95% probability of being the true system pole or zero value. You can specify the confidence interval as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution. For example, a probability of 0.99 (99%) corresponds to 2.58 standard deviations.

You can use pole-zero plots to evaluate whether it might be useful to reduce model order. When confidence intervals for a pole-zero pair overlap, this overlap indicates a possible pole-zero cancellation. For more information, see "Reducing Model Order Using Pole-Zero Plots" on page 17-71.

See Also

More About

*   "Model Poles and Zeros Using the System Identification App" on page 17-72
*   "Plot Poles and Zeros at the Command Line" on page 17-73

Reducing Model Order Using Pole-Zero Plots

You can use pole-zero plots of linear identified models to evaluate whether it might be useful to reduce model order. When confidence intervals for a pole-zero pair overlap, this overlap indicates a possible pole-zero cancellation.

For example, you can use the following syntax to plot a 1-standard deviation confidence interval around model poles and zeros.

```
showConfidence(iopzplot(model))
```

If poles and zeros overlap, try estimating a lower order model.

Always validate model output and residuals to see if the quality of the fit changes after reducing model order. If the plot indicates pole-zero cancellations, but reducing model order degrades the fit, then the extra poles probably describe noise. In this case, you can choose a different model structure that decouples system dynamics and noise. For example, try ARMAX, Output-Error, or Box-Jenkins polynomial model structures with an *A* or *F* polynomial of an order equal to that of the number of uncanceled poles. For more information about estimating linear polynomial models, see "Input-Output Polynomial Models".

See Also
```
iopzplot
```

More About
- "Pole and Zero Plots" on page 17-69
- "Plot Poles and Zeros at the Command Line" on page 17-73
- "Model Poles and Zeros Using the System Identification App" on page 17-72

Model Poles and Zeros Using the System Identification App

To create a pole-zero plot for parametric linear models in the System Identification app, select the **Zeros and poles** check box in the **Model Views** area. For general information about creating and working with plots, see "Working with Plots" on page 21-8.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification app. Active models display a thick line inside the Model Board icon.

The following table summarizes the Zeros and Poles plot settings.

Zeros and Poles Plot Settings

Action	Command
Display the confidence interval.	• To display the dashed lines on either side of the nominal pole and zero values, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Show real and imaginary axes.	Select **Style > Re/Im-axes**. Select this option again to hide the axes.
Show the unit circle.	Select **Style > Unit circle**. Select this option again to hide the unit circle. The unit circle is useful as a reference curve for discrete-time models.
(Multiple-output system only) Select an input-output pair to view the poles and zeros corresponding to those channels.	Select the output by name in the **Channel** menu.

See Also

More About

• "Pole and Zero Plots" on page 17-69
• "Plot Poles and Zeros at the Command Line" on page 17-73

Plot Poles and Zeros at the Command Line

You can create a pole-zero plot for linear identified models using the `iopzmap` and `iopzplot` commands.

To display confidence intervals for a specified number of standard deviations, use the following syntax:

```
h = iopzplot(model);
showConfidence(h,sd)
```

where `sd` is the number of standard deviations of a Gaussian distribution. For example, a confidence value of 99% for the nominal model curve corresponds to 2.58 standard deviations.

Command	Description	Example
`iopzmap,iopzplot`	Plots zeros and poles of the model on the S-plane or Z-plane for continuous-time or discrete-time model, respectively.	To plot the poles and zeros of the model `sys`, use the following command: `iopzmap(sys)`

See Also

More About

- "Pole and Zero Plots" on page 17-69
- "Model Poles and Zeros Using the System Identification App" on page 17-72

Analyzing MIMO Models

Overview of Analyzing MIMO Models

If you plot a MIMO system, or an LTI array containing multiple identified linear models, you can use special features of the right-click menu to group the response plots by input/output (I/O) pairs, or select individual plots for display. For example, generate a random 3-input, 3-output MIMO system and then randomly sample it 10 times. Plot the step response for all the models.

```
sys_mimo=rsample(idss(rss(3,3,3)),10);
step(sys_mimo);
```

sys_mimo is an array of ten 3-input, 3-output systems.

A set of 9 plots appears, one from each input to each output, for the ten model samples.
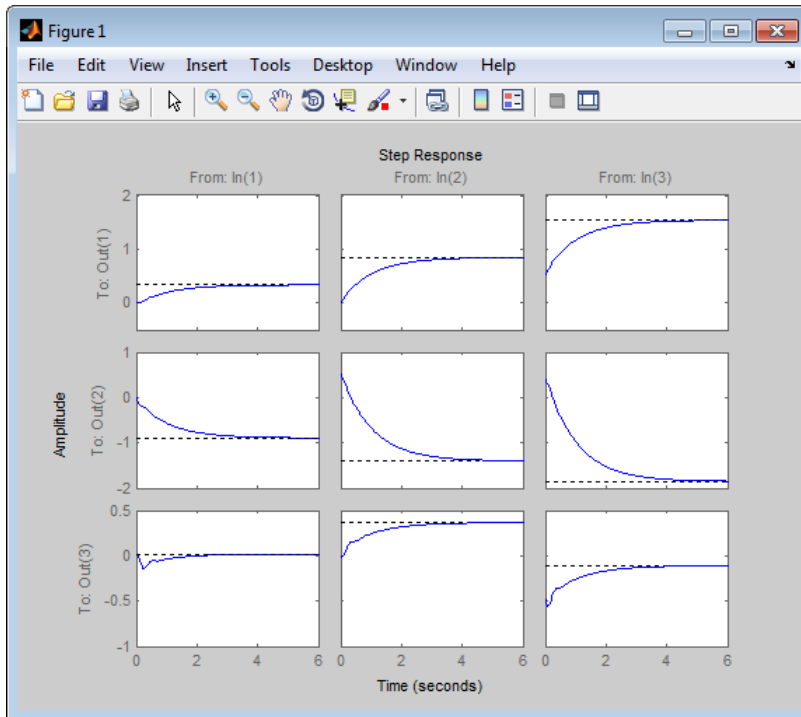
Array Selector

If you plot an identified linear model array, **Array Selector** appears as an option in the right-click menu. Selecting this option opens the **Model Selector for LTI Arrays**, shown below.
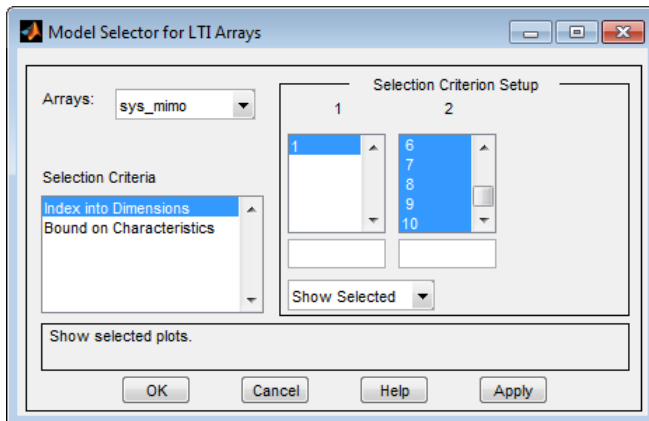
You can use this window to include or exclude models within the LTI array using various criteria.

Arrays

Select the LTI array for model selection using the **Arrays** list.

Selection Criteria

There are two selection criteria. The default, **Index into Dimensions**, allows you to include or exclude specified indices of the LTI Array. Select systems from the **Selection Criterion Setup** section of the dialog box. Then, Specify whether to show or hide the systems using the pull-down menu below the Setup lists.

The second criterion is **Bound on Characteristics**. Selecting this options causes the Model Selector to reconfigure. The reconfigured window is shown below
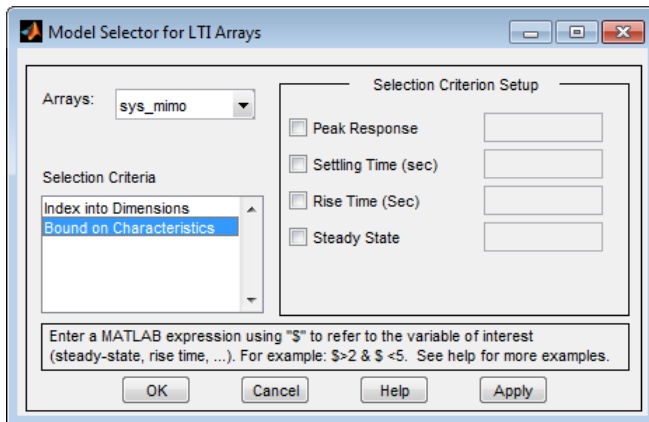
Use this option to select systems for inclusion or exclusion in your response plot based on their time response characteristics. The panel directly above the buttons describes how to set the inclusion or exclusion criteria based on which selection criteria you select from the reconfigured **Selection Criteria Setup** panel.

I/O Grouping for MIMO Models

You can group the plots by inputs, by outputs, or both by selecting **I/O Grouping** from the right-click menu, and then selecting **Inputs**, **Outputs**, or **All**.

17-75

For example, if you select **Outputs**, the step plot reconfigures into 3 plots, grouping all the outputs together on each plot. Each plot now displays the responses from one of the inputs to all of the MIMO system's outputs, for all of the models in the array.
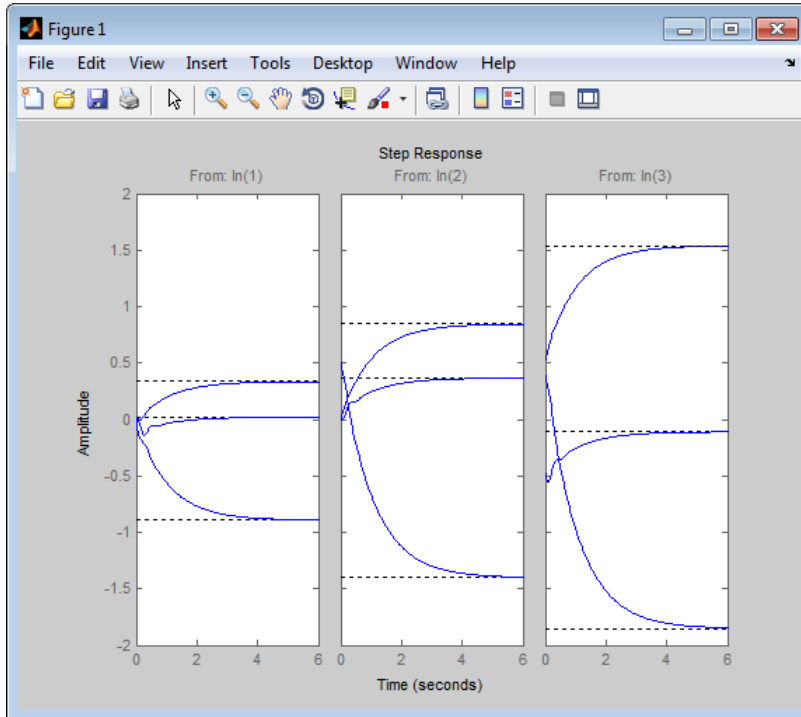
Selecting **None** returns to the default configuration, where all I/O pairs are displayed individually.

Selecting I/O Pairs

Another way to organize MIMO system information is to choose **I/O Selector** from the right-click menu, which opens the **I/O Selector** window.
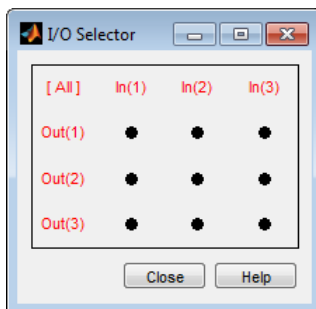
This window automatically configures to the number of I/O pairs in your MIMO system. You can select:

•   Any individual plot (only one at a time) by clicking on a button

•   Any row or column by clicking on Y(*) or U(*)

- All of the plots by clicking [all]

Using these options, you can inspect individual I/O pairs, or look at particular I/O channels in detail.

See Also
Linear System Analyzer

More About
- "Model Arrays" (Control System Toolbox)

Customize Response Plots Using the Response Plots Property Editor

Opening the Property Editor

After you create a response plot, there are two ways to open the Property Editor:

- Double-click in the plot region.
- Right-click the plot, and select **Properties** from the context menu.

Before looking at the Property Editor, open a step response plot using these commands.

```
sys_dc = idtf([1 4],[1 20 5]);
step(sys_dc)
```

This creates a step plot. Right-click the plot, and select **Properties** from the context menu. When you open the Property Editor, squares appear around the step response plot.
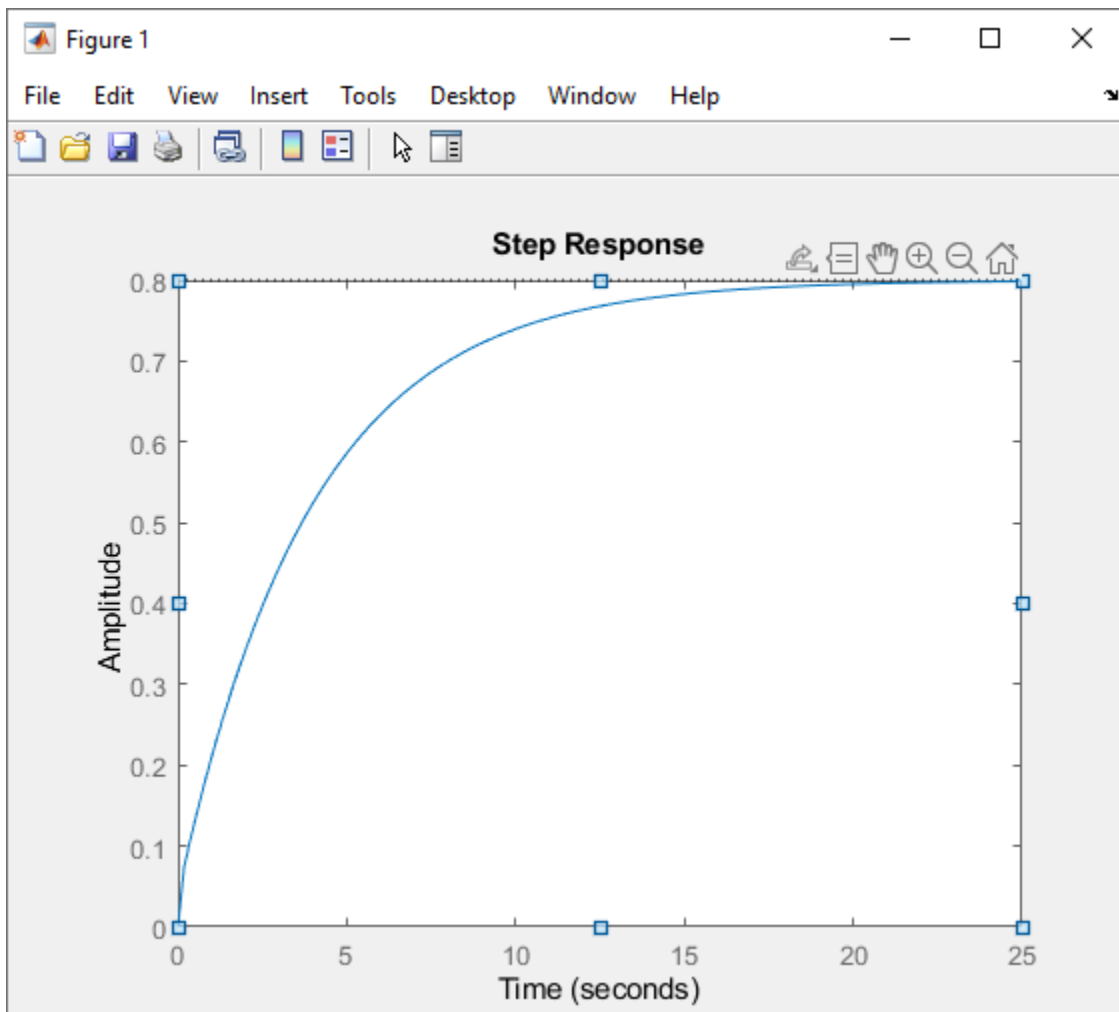
Overview of Response Plots Property Editor

The appearance of the Property Editor dialog box depends on the type of response plot. This figure shows the Property Editor dialog box for a step response.
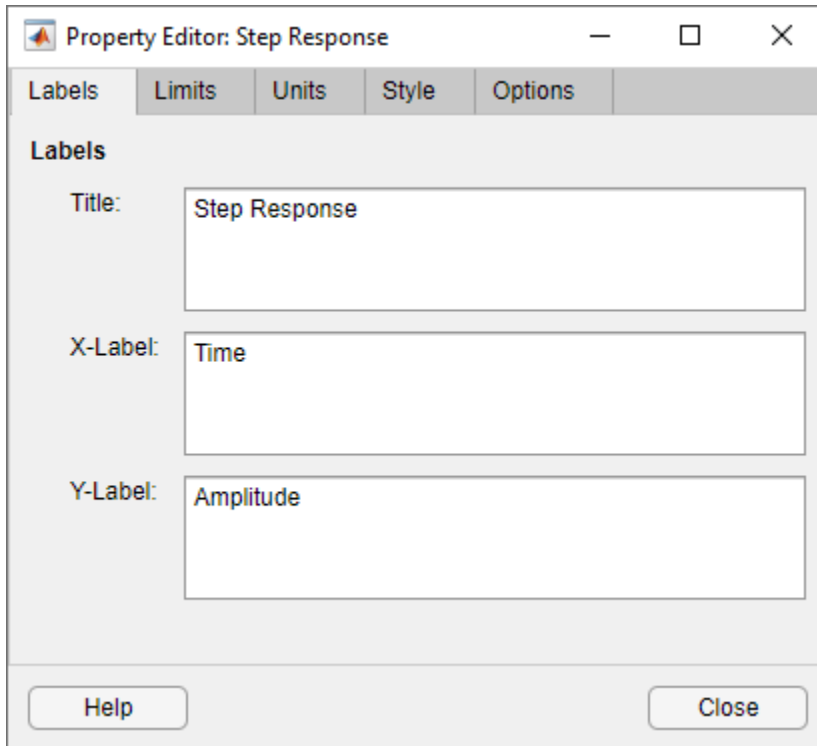
The Property Editor for Step Response

In general, you can change the following properties of response plots. Only the **Labels** and **Limits** panes are available when using the Property Editor with Simulink Design Optimization™ software.

- Titles and X- and Y-labels in the **Labels** pane.

- Numerical ranges of the X and Y axes in the **Limits** pane.

- Units where applicable (e.g., rad/s to Hertz) in the **Units** pane.

  If you cannot customize units, the Property Editor displays that no units are available for the selected plot.

- Styles in the **Styles** pane.

  You can show a grid, adjust font properties, such as font size, bold, and italics, and change the axes foreground color

- Change options where applicable in the **Options** pane.

  These include peak response, settling time, phase and gain margins, etc. Plot options change with each plot response type. The Property Editor displays only the options that make sense for the selected response plot. For example, phase and gain margins are not available for step responses.

As you make changes in the Property Editor, they display immediately in the response plot. Conversely, if you make changes in a plot using right-click menus, the Property Editor for that plot automatically updates. The Property Editor and its associated plot are dynamically linked.

Labels Pane

To specify new text for plot titles and axis labels, type the new names in the field next to the label you want to change. The label changes immediately as you type, so you can see how the new text looks as you are typing.
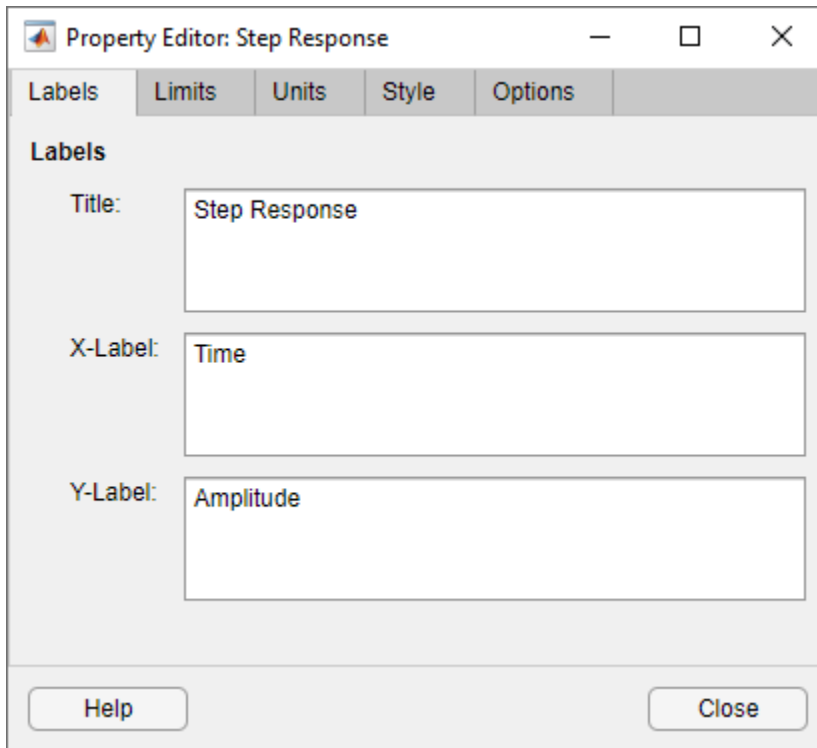
Limits Pane

Default values for the axes limits make sure that the maximum and minimum x and y values are displayed. If you want to override the default settings, change the values in the Limits fields. The **Auto-Scale** box automatically clears if you click a different field. The new limits appear immediately in the response plot.

To re-establish the default values, select the **Auto-Scale** box again.
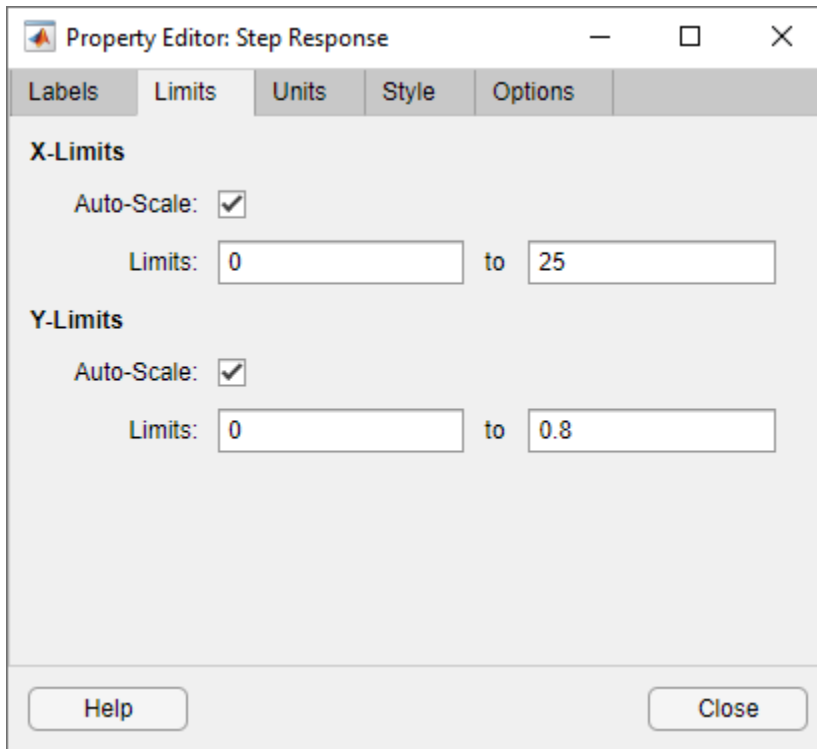
Units Pane

You can use the **Units** pane to change units in your response plot. The contents of this pane depend on the response plot associated with the editor. Use the menus to toggle between units.
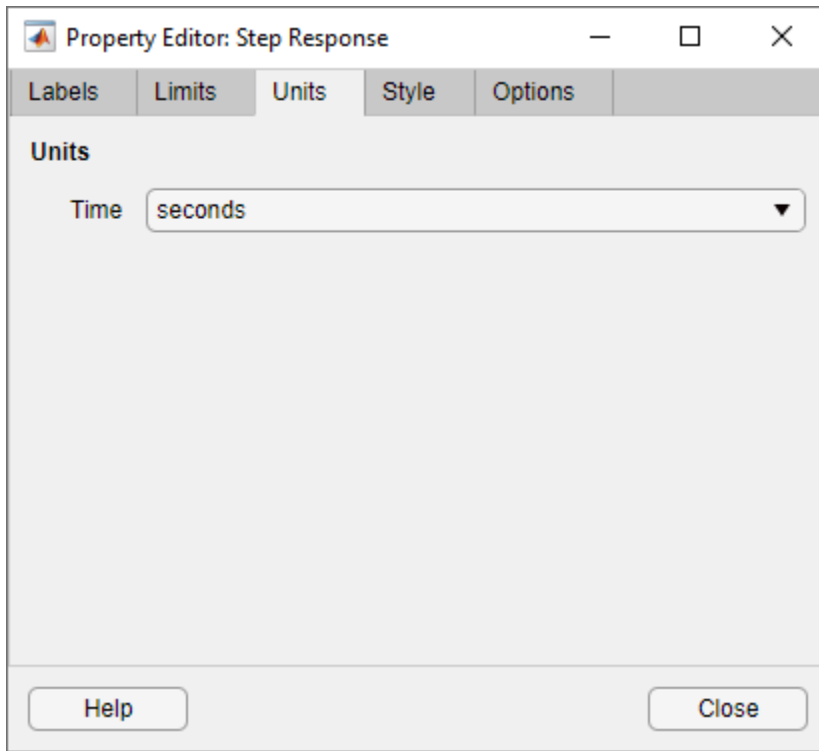
Optional Unit Conversions for Response Plots

Response Plot	Unit Conversions
Bode and Bode Magnitude	• Frequency By default, shows `rad/TimeUnit` where `TimeUnit` is the system time units specified in the `TimeUnit` property of the input system. **Frequency Units Options** • `'Hz'` • `'rad/s'` • `'rpm'` • `'kHz'` • `'MHz'` • `'GHz'` • `'rad/nanosecond'` • `'rad/microsecond'` • `'rad/millisecond'` • `'rad/minute'` • `'rad/hour'` • `'rad/day'` • `'rad/week'` • `'rad/month'` • `'rad/year'` • `'cycles/nanosecond'` • `'cycles/microsecond'` • `'cycles/millisecond'` • `'cycles/hour'` • `'cycles/day'` • `'cycles/week'` • `'cycles/month'` • `'cycles/year'` • Frequency scale is logarithmic or linear. • Magnitude in decibels (dB) or the absolute value • Phase in degrees or radians

Response Plot	Unit Conversions
Impulse	• Time By default, shows the system time units specified in the `TimeUnit` property of the input system. **Time Units Options** • `'nanoseconds'` • `'microseconds'` • `'milliseconds'` • `'seconds'` • `'minutes'` • `'hours'` • `'days'` • `'weeks'` • `'months'` • `'years'`

Response Plot	Unit Conversions
Nichols Chart	• Frequency By default, shows rad/TimeUnit where TimeUnit is the system time units specified in the TimeUnit property of the input system. **Frequency Units Options** • `'Hz'` • `'rad/s'` • `'rpm'` • `'kHz'` • `'MHz'` • `'GHz'` • `'rad/nanosecond'` • `'rad/microsecond'` • `'rad/millisecond'` • `'rad/minute'` • `'rad/hour'` • `'rad/day'` • `'rad/week'` • `'rad/month'` • `'rad/year'` • `'cycles/nanosecond'` • `'cycles/microsecond'` • `'cycles/millisecond'` • `'cycles/hour'` • `'cycles/day'` • `'cycles/week'` • `'cycles/month'` • `'cycles/year'` • Phase in degrees or radians

Response Plot	Unit Conversions
Nyquist Diagram	• Frequency By default, shows rad/TimeUnit where TimeUnit is the system time units specified in the TimeUnit property of the input system. **Frequency Units Options** • 'Hz' • 'rad/s' • 'rpm' • 'kHz' • 'MHz' • 'GHz' • 'rad/nanosecond' • 'rad/microsecond' • 'rad/millisecond' • 'rad/minute' • 'rad/hour' • 'rad/day' • 'rad/week' • 'rad/month' • 'rad/year' • 'cycles/nanosecond' • 'cycles/microsecond' • 'cycles/millisecond' • 'cycles/hour' • 'cycles/day' • 'cycles/week' • 'cycles/month' • 'cycles/year'

Response Plot	Unit Conversions
Pole/Zero Map	• Time By default, shows the system time units specified in the `TimeUnit` property of the input system. **Time Units Options** • `'nanoseconds'` • `'microseconds'` • `'milliseconds'` • `'seconds'` • `'minutes'` • `'hours'` • `'days'` • `'weeks'` • `'months'` • `'years'` • Frequency By default, shows `rad/TimeUnit` where `TimeUnit` is the system time units specified in the `TimeUnit` property of the input system. **Frequency Units Options** • `'Hz'` • `'rad/s'` • `'rpm'` • `'kHz'` • `'MHz'` • `'GHz'` • `'rad/nanosecond'` • `'rad/microsecond'` • `'rad/millisecond'` • `'rad/minute'` • `'rad/hour'` • `'rad/day'` • `'rad/week'` • `'rad/month'` • `'rad/year'` • `'cycles/nanosecond'` • `'cycles/microsecond'`

Response Plot	Unit Conversions
	'cycles/millisecond' 'cycles/hour' 'cycles/day' 'cycles/week' 'cycles/month' 'cycles/year'
Singular Values	- Frequency By default, shows rad/TimeUnit where TimeUnit is the system time units specified in the TimeUnit property of the input system. **Frequency Units Options** 'Hz' 'rad/s' 'rpm' 'kHz' 'MHz' 'GHz' 'rad/nanosecond' 'rad/microsecond' 'rad/millisecond' 'rad/minute' 'rad/hour' 'rad/day' 'rad/week' 'rad/month' 'rad/year' 'cycles/nanosecond' 'cycles/microsecond' 'cycles/millisecond' 'cycles/hour' 'cycles/day' 'cycles/week' 'cycles/month' 'cycles/year' - Frequency scale is logarithmic or linear. - Magnitude in decibels or the absolute value using logarithmic or linear scale

Response Plot	Unit Conversions
Step	• Time By default, shows the system time units specified in the `TimeUnit` property of the input system. **Time Units Options** • `'nanoseconds'` • `'microseconds'` • `'milliseconds'` • `'seconds'` • `'minutes'` • `'hours'` • `'days'` • `'weeks'` • `'months'` • `'years'`

Style Pane

Use the Style pane to toggle grid visibility and set font preferences and axes foreground colors for response plots.
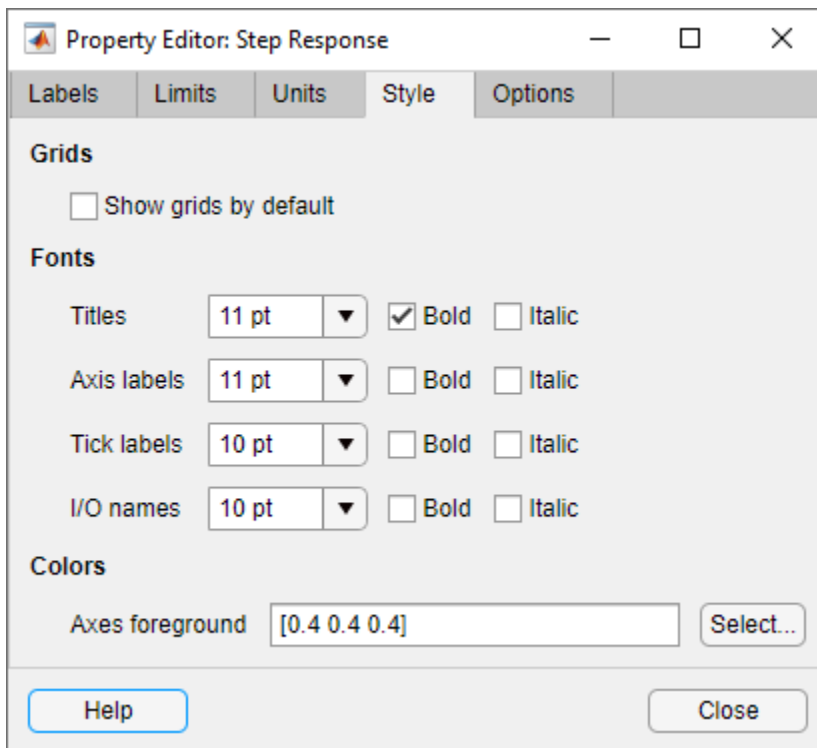
You have the following choices:

- **Grid** — Activate grids by default in new plots.
- **Fonts** — Set the font size, weight (bold), and angle (italic) for fonts used in response plot titles, X/Y-labels, tick labels, and I/O names. Select font sizes from the menus or type any font-size values in the fields.
- **Colors** — Specify the color vector to use for the axes foreground, which includes the X-Y axes, grid lines, and tick labels. Use a three-element vector to represent red, green, and blue (RGB) values. Vector element values can range from 0 to 1.

  If you do not want to specify RGB values numerically, click the **Select** button to open the Select Color dialog box.

Options Pane

The **Options** pane enables you to customize response characteristics for plots. Each response plot has its own set of characteristics and optional settings. When you change the value in a field, press **Enter** on your keyboard to update the response plot.
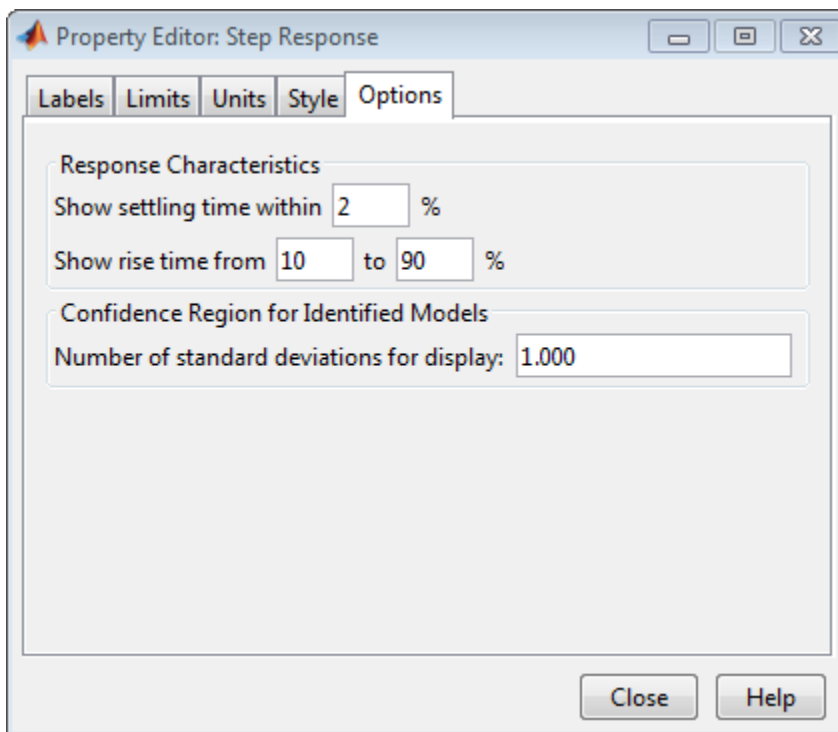
Response Characteristic Options for Response Plots

Plot	Customizable Feature
Bode Diagram and Bode Magnitude	• **Magnitude Response** Select lower magnitude limit. • **Phase Response** By default, plots display exact phase. Check **Wrap phase** to wrap the phase into the interval [–180º,180º). To wrap accumulated phase at a different value, enter the value in the **Branch** field. For example, entering 0 causes the plot to wrap the phase into the interval [0º,360º). Check **Adjust phase offsets** to keep phase close to a particular value, within a range of 0º–360º, at a given frequency. • **Confidence Region for Identified Models** This option is available with System Identification Toolbox. Specify number of standard deviations for plotting the response confidence region. To see the confidence region, right-click the plot, and select **Characteristics > Confidence Region**.
Impulse	• **Response Characteristics** Show settling time within xx% (specify the percentage). • **Confidence Region for Identified Models** These options are available with System Identification Toolbox. **Display using zero mean interval**: For an identified model with impulse response y and standard deviation Δy, plot the uncertainty $\pm\Delta y$ as a function of time (default). If cleared, $y\pm\Delta y$ as a function of time is plotted. **Number of standard deviations for display**: Specify number of standard deviations for plotting the uncertainty. To see the confidence interval, right-click the plot, and select **Characteristics > Confidence Region**.

Plot	Customizable Feature
Nichols Chart	• **Magnitude Response** Select lower magnitude limit. • **Phase Response** By default, plots display exact phase. Check **Wrap phase** to wrap the phase into the interval [–180º,180º). To wrap accumulated phase at a different value, enter the value in the **Branch** field. For example, entering 0 causes the plot to wrap the phase into the interval [0º,360º). Check **Adjust phase offsets** to keep phase close to a particular value, within a range of 0º–360º, at a given frequency.
Nyquist Diagram	• **Confidence Region for Identified Models** These options are available with System Identification Toolbox. **Number of standard deviations for display**: Specify number of standard deviations for plotting the confidence ellipses. **Display spacing**: Specify the frequency spacing of confidence ellipses. The default is 5, which means that the confidence ellipses are shown at every fifth frequency sample. To see the confidence ellipses, right-click the plot, and select **Characteristics > Confidence Region**.
Pole/Zero Map	• **Confidence Region for Identified Models** This option is available with System Identification Toolbox. Specify number of standard deviations for displaying the confidence region characteristic. To see the confidence region, right-click the plot, and select **Characteristics > Confidence Region**.
Singular Values	None

Plot	Customizable Feature
Step	• **Response Characteristics** Show settling time within *xx*% (specify the percentage). Show rise time from *xx* to *yy*% (specify the percentages) • **Confidence Region for Identified Models** This option is available with System Identification Toolbox. Specify number of standard deviations for plotting the response confidence region. To see the confidence region, right-click the plot, and select **Characteristics > Confidence Region**.

Editing Subplots Using the Property Editor

If you create more than one plot in a single figure window, you can edit each plot individually. For example, the following code creates a figure with two plots, a step and an impulse response with two randomly selected systems:

```
subplot(2,1,1)
step(rss(2,1))
subplot(2,1,2)
impulse(rss(1,1))
```

After the figure window appears, double-click in the upper (step response) plot to activate the Property Editor. A set of small squares appear around the step response, indicating that it is the active plot for the editor. To switch to the lower (impulse response) plot, click once in the impulse response plot region. The set of squares switches to the impulse response, and the Property Editor updates as well.

See Also

More About

• "Toolbox Preferences Editor" on page 18-2

Compute Model Uncertainty

Why Analyze Model Uncertainty?

In addition to estimating model parameters, the toolbox algorithms also estimate variability of the model parameters that result from random disturbances in the output.

Understanding model variability helps you to understand how different your model parameters would be if you repeated the estimation using a different data set (with the same input sequence as the original data set) and the same model structure.

When validating your parametric models, check the uncertainty values. Large uncertainties in the parameters might be caused by high model orders, inadequate excitation, and poor signal-to-noise ratio in the data.

Note You can get model uncertainty data for linear parametric black-box models, and both linear and nonlinear grey-box models. Supported model objects include `idproc`, `idpoly`, `idss`, `idtf`, `idgrey`, `idfrd`, and `idnlgrey`.

What Is Model Covariance?

Uncertainty in the model is called *model covariance*.

When you estimate a model, the covariance matrix of the estimated parameters is stored with the model. Use `getcov` to fetch the covariance matrix. Use `getpvec` to fetch the list of parameters and their individual uncertainties that have been computed using the covariance matrix. The covariance matrix is used to compute all uncertainties in model output, Bode plots, residual plots, and pole-zero plots.

Computing the covariance matrix is based on the assumption that the model structure gives the correct description of the system dynamics. For models that include a disturbance model H, a correct uncertainty estimate assumes that the model produces white residuals. To determine whether you can trust the estimated model uncertainty values, perform residual analysis tests on your model. For more details about residual analysis, see the topics on the "Residual Analysis" page. If your model passes residual analysis tests, there is a good chance that the true system lies within the confidence interval and any parameter uncertainties results from random disturbances in the output.

For output-error models, such as transfer function models, state-space with K=0 and polynomial models of output-error form, with the noise model H fixed to 1, the covariance matrix computation does not assume white residuals. Instead, the covariance is estimated based on the estimated color of the residual correlations. This estimation of the noise color is also performed for state-space models with $K=0$, which is equivalent to an output-error model.

Types of Model Uncertainty Information

You can view the following uncertainty information from linear and nonlinear grey-box models:

- Uncertainties of estimated parameters.

  Type `present(model)` at the prompt, where `model` represents the name of a linear or nonlinear model.

- Confidence intervals on the linear model plots, including step-response, impulse-response, Bode, Nyquist, noise spectrum and pole-zero plots.

  Confidence intervals are computed based on the variability in the model parameters. For information about displaying confidence intervals, see "Definition of Confidence Interval for Specific Model Plots" on page 17-95.

- Covariance matrix of the estimated parameters in linear models and nonlinear grey-box models using `getcov`.

- Estimated standard deviations of polynomial coefficients, poles/zeros, or state-space matrices using `idssdata`, `tfdata`, `zpkdata`, and `polydata`.

- Simulated output values for linear models with standard deviations using `sim`.

  Call the `sim` command with output arguments, where the second output argument is the estimated standard deviation of each output value. For example, type `[ysim,ysimsd] = sim(model,data)`, where `ysim` is the simulated output, `ysimsd` contains the standard deviations on the simulated output, and `data` is the simulation data.

- Perform Monte-Carlo analysis using `rsample` to generate a random sampling of an identified model in a given confidence region. An array of identified systems of the same structure as the input system is returned. The parameters of the returned models are perturbed about their nominal values in a way that is consistent with the parameter covariance.

- Simulate the effect of parameter uncertainties on a model's response using `simsd`.

Definition of Confidence Interval for Specific Model Plots

You can display the confidence interval on the following plot types:

Plot Type	Confidence Interval Corresponds to the Range of ...	More Information on Displaying Confidence Interval
Simulated and Predicted Output	Output values with a specific probability of being the actual output of the system.	Model Output Plots on page 17-17
Residuals	Residual values with a specific probability of being statistically insignificant for the system.	Residuals Plots on page 17-44
Impulse and Step	Response values with a specific probability of being the actual response of the system.	Impulse and Step Plots on page 17-52
Frequency Response	Response values with a specific probability of being the actual response of the system.	Frequency Response Plots on page 17-59
Noise Spectrum	Power-spectrum values with a specific probability of being the actual noise spectrum of the system.	Noise Spectrum Plots on page 17-65
Poles and Zeros	Pole or zero values with a specific probability of being the actual pole or zero of the system.	Pole-Zero Plots on page 17-65

Troubleshooting Model Estimation

About Troubleshooting Models

During validation, models can exhibit undesirable characteristics or a poor fit to the validation data.

Use the tips in these sections to help improve your model performance. Some features, such as low signal-to-noise ratio, varying system properties, or nonstationary disturbances, can produce data for which a good model fit is not possible.

Model Order Is Too High or Too Low

A poor fit in the Model Output plot can be the result of an incorrect model order. System identification is largely a trial-and-error process when selecting model structure and model order. Ideally, you want the lowest-order model that adequately captures the system dynamics. High-order models are more expensive to compute and result in greater parameter uncertainty.

Start by estimating the model order as described in "Preliminary Step – Estimating Model Orders and Input Delays" on page 6-8. Use the suggested order as a starting point to estimate the lowest possible order with different model structures. After each estimation, monitor the Model Output and Residual Analysis plots, and then adjust your settings for the next estimation.

When a low-order model fits the validation data poorly, estimate a higher-order model to see if the fit improves. For example, if the Model Output plot shows that a fourth-order model gives poor results, estimate an eighth-order model. When a higher-order model improves the fit, you can conclude that higher-order linear models are potentially sufficient for your application.

Use an independent data set to validate your models. If you use the same data set for both estimation and validation, the fit always improves as you increase the model order and you risk overfitting. However, if you use an independent data set to validate your model, the fit eventually deteriorates if the model orders are too high.

Substantial Noise in the System

Substantial noise in your system can result in a poor model fit. The presence of such noise is indicated when:

- A state-space model produces a better fit than an ARX model. While a state-space structure has sufficient flexibility to model noise, an ARX structure is unable to independently model noise and system dynamics. The following ARX model equation shows that *A* couples the dynamics and the noise terms by appearing in the denominator of both:

$$y = \frac{B}{A}u + \frac{1}{A}e$$

- A residual analysis plot shows significant autocorrelation of residuals at nonzero lags. For more information about residual analysis, see the topics on the "Residual Analysis" page.

To model noise more carefully, use either an ARMAX or the Box-Jenkins model structure, both of which model the noise and dynamics terms using different polynomials.

Unstable Models

Unstable Linear Model

You can test whether a *linear model* is unstable is by examining the pole-zero plot of the model, which is described in "Pole and Zero Plots" on page 17-69. The stability threshold for pole values differs for discrete-time and continuous-time models, as follows:

- For stable continuous-time models, the real part of the pole is less than 0.
- For stable discrete-time models, the magnitude of the pole is less than 1.

Note Linear trends in estimation data can cause the identified linear models to be unstable. However, detrending the model does not guarantee stability.

If your model is unstable, but you believe that your system is stable, you can.

- Force stability during estimation — Set the `Focus` estimation option to a value that guarantees a stable model. This setting can result in reduced model quality.
- Allow for some instability — Set the stability threshold advanced estimation option to allow for a margin of error:
  - For continuous-time models, set the value of `Advanced.StabilityThreshold.s`. The model is considered stable if the pole on the far right is to the left of *s*.
  - For discrete-time models, set the value of `Advanced.StabilityThreshold.z`. The model is considered stable if all of the poles are inside a circle with a radius of *z* that is centered at the origin.

For more information about `Focus` and `Advanced.StabilityThreshold`, see the various commands for creating estimation option sets, such as `tfestOptions`, `ssestOptions`, and `procestOptions`.

Unstable Nonlinear Models

To test if a *nonlinear model* is unstable, plot the simulated model output on top of the validation data. If the simulated output diverges from measured output, the model is unstable. However, agreement between model output and measured output does not guarantee stability.

When an Unstable Model Is OK

In some cases, an unstable model is still useful. For example, if your system is unstable without a controller, you can use your model for control design. In this case, you can import the unstable model into Simulink or Control System Toolbox products.

Missing Input Variables

If modeling noise and trying different model structures and orders still results in a poor fit, try adding more inputs that can affect the output. Inputs do not need to be control signals. Any measurable signal can be considered an input, including measurable disturbances.

Include additional measured signals in your input data, and estimate the model again.

System Nonlinearities

If a linear model shows a poor fit to the validation data, consider whether nonlinear effects are present in the system.

You can model the nonlinearities by performing a simple transformation on the input signals to make the problem linear in the new variables. For example, in a heating process with electrical power as the driving stimulus, you can multiply voltage and current measurements to create a power input signal.

If your problem is sufficiently complex and you do not have physical insight into the system, try fitting nonlinear black-box models to your data, see "About Identified Nonlinear Models" on page 11-2.

Nonlinearity Estimator Produces a Poor Fit

For nonlinear ARX and Hammerstein-Wiener models, the Model Output plot does not show a good fit when the nonlinearity estimator has incorrect complexity.

Specify the complexity of piece-wise-linear, wavelet, sigmoid, and custom networks using the `NumberOfUnits` nonlinear estimator property. A higher number of units indicates a more complex nonlinearity estimator. When using neural networks, specify the complexity using the parameters of the network object. For more information, see the Deep Learning Toolbox documentation.

To select the appropriate nonlinearity estimator complexity, first validate the output of a low-complexity model. Next, increase the model complexity and validate the output again. The model fit degrades when the nonlinearity estimator becomes too complex. This degradation in performance is only visible if you use independent estimation and validation data sets

See Also

More About
- "Ways to Validate Models" on page 17-2
- "Preliminary Step – Estimating Model Orders and Input Delays" on page 6-8
- "Pole and Zero Plots" on page 17-69
- "What Is Residual Analysis?" on page 17-43
- "Next Steps After Getting an Accurate Model" on page 17-99
- "Estimate Initial Conditions for Simulating Identified Models" on page 20-26
- "Reproduce Command Line or System Identification App Simulation Results in Simulink" on page 20-15

Next Steps After Getting an Accurate Model

For linear parametric models, you can perform the following operations:

- Transform between continuous-time and discrete-time representation.

  See "Transforming Between Discrete-Time and Continuous-Time Representations" on page 4-15.

- Transform between linear model representations, such as between polynomial, state-space, and zero-pole representations.

  See "Transforming Between Linear Model Representations" on page 4-29.

- Extract numerical data from transfer functions, pole-zero models, and state-space matrices.

  See "Extracting Numerical Model Data" on page 4-12.

For nonlinear black-box models (`idnlarx` and `idnlhw` objects), you can compute a linear approximation of the nonlinear model. See "Linear Approximation of Nonlinear Black-Box Models" on page 11-50.

System Identification Toolbox models in the MATLAB workspace are immediately available to other MathWorks® products. However, if you used the System Identification app to estimate models, you must first export the models to the MATLAB workspace.

Tip To export a model from the app, drag the model icon to the **To Workspace** rectangle. Alternatively, right-click the model to open the Data/model Info dialog box. Click **Export**.

If you have the Control System Toolbox software installed, you can import your linear plant model for control-system design. For more information, see "Using Identified Models for Control Design Applications" on page 19-2.

Finally, if you have Simulink software installed, you can exchange data between the System Identification Toolbox software and the Simulink environment. For more information, see "Simulate Identified Model in Simulink" on page 20-5.

Setting Toolbox Preferences

Toolbox Preferences Editor

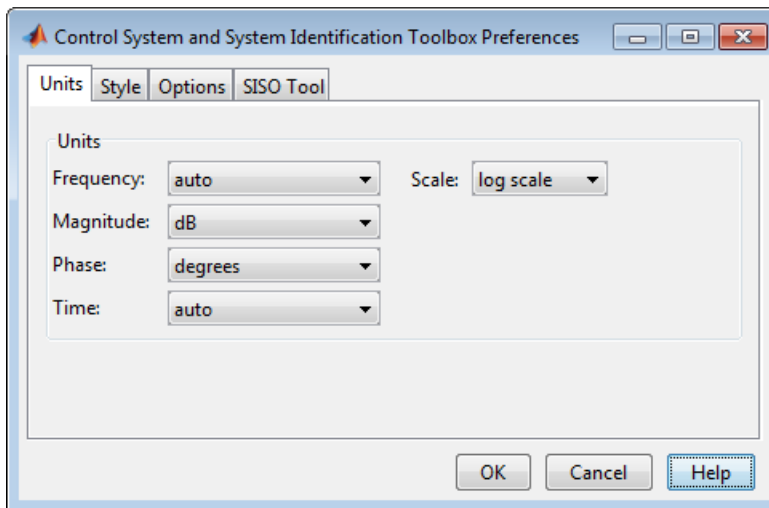### Overview of the Toolbox Preferences Editor

The Toolbox Preferences editor allows you to set plot preferences that will persist from session to session.
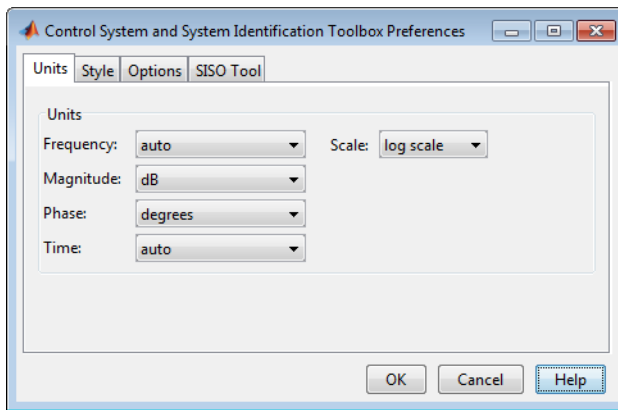
Opening the Toolbox Preferences Editor

To open the Toolbox Preferences editor, enter

```
identpref
```

at the MATLAB prompt.

Units Pane

Use the **Units** pane to set preferences for the following:

- **Frequency**

  The default `auto` option uses `rad/TimeUnit` as the frequency units relative to the system time units, where `TimeUnit` is the system time units specified in the `TimeUnit` property of the system on frequency-domain plots. For multiple systems with different time units, the units of the first system is used.

  For the frequency axis, you can select logarithmic or linear scales.

  Other Frequency Units Options

  - `'Hz'`
  - `'rad/s'`
  - `'rpm'`
  - `'kHz'`
  - `'MHz'`
  - `'GHz'`
  - `'rad/nanosecond'`
  - `'rad/microsecond'`
  - `'rad/millisecond'`
  - `'rad/minute'`
  - `'rad/hour'`
  - `'rad/day'`
  - `'rad/week'`
  - `'rad/month'`
  - `'rad/year'`
  - `'cycles/nanosecond'`
  - `'cycles/microsecond'`
  - `'cycles/millisecond'`
  - `'cycles/hour'`
  - `'cycles/day'`
  - `'cycles/week'`
  - `'cycles/month'`
  - `'cycles/year'`
- **Magnitude** — Decibels (dB) or absolute value (abs)
- **Phase** — Degrees or radians
- **Time**

  The default `auto` option uses the time units specified in the `TimeUnit` property of the system on the time- and frequency-domain plots. For multiple systems with different time units, the units of the first system is used.

  Other Time Units Options

  - `'nanoseconds'`

- `'microseconds'`
- `'milliseconds'`
- `'seconds'`
- `'minutes'`
- `'hours'`
- `'days'`
- `'weeks'`
- `'months'`
- `'years'`

Style Pane

Use the **Style** pane to toggle grid visibility and set font preferences and axes foreground colors for all plots you create. This figure shows the Style pane.
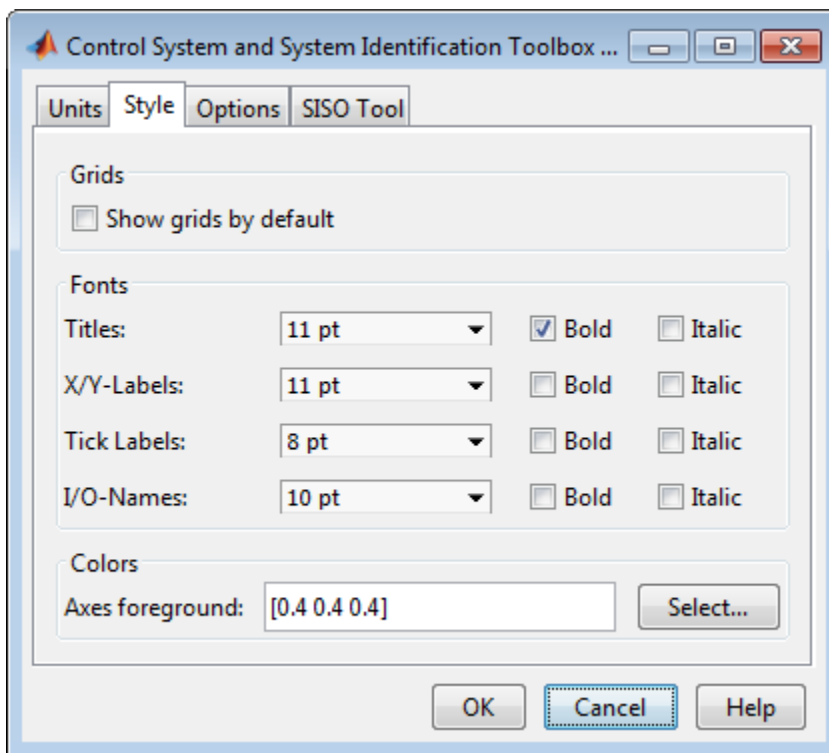
You have the following choices:

- **Grid** — Activate grids by default in new plots.
- **Fonts** — Set the font size, weight (bold), and angle (italic). Select font sizes from the menus or type any font-size values in the fields.
- **Colors** — Specify the color vector to use for the axes foreground, which includes the X-Y axes, grid lines, and tick labels. Use a three-element vector to represent red, green, and blue (RGB) values. Vector element values can range from 0 to 1.

If you do not want to specify RGB values numerically, click the **Select** button to open the **Select Colors** dialog box.

Options Pane

The Options pane has selections for time responses and frequency responses. This figure shows the Options pane with default settings.
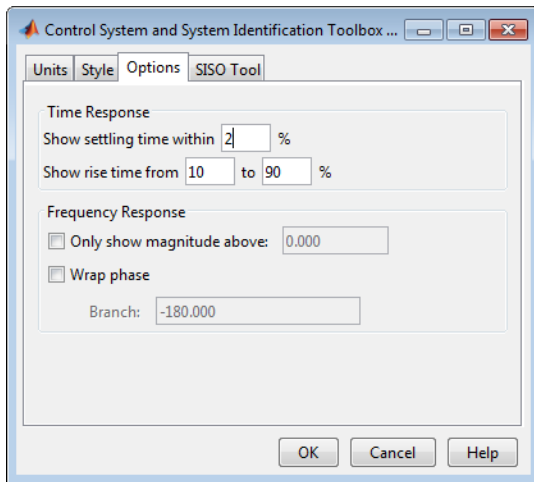
For time response plots, the following options are available:

- **Show settling time within** *xx*% — Set the threshold of the settling time calculation to any percentage from 0 to 100%. The default is 2%.
- **Specify rise time from** *xx*% to *yy*%— The standard definition of rise time is the time it takes the signal to go from 10% to 90% of the final value. Specify any percentages you like (from 0% to 100%), provided that the first value is smaller than the second.

For frequency response plots, the following options are available:

- **Only show magnitude above** — Specify a lower limit for magnitude values in response plots so that you can focus on a region of interest.
- **Wrap phase** — Wrap the phase into the interval [–180º,180º). To wrap accumulated phase at a different value, enter the value in the **Branch** field. For example, entering 0 causes the plot to wrap the phase into the interval [0º,360º).

SISO Tool Pane

The SISO Tool pane has settings for **Control System Designer** (requires a Control System Toolbox license). This figure shows the SISO Tool pane with default settings.
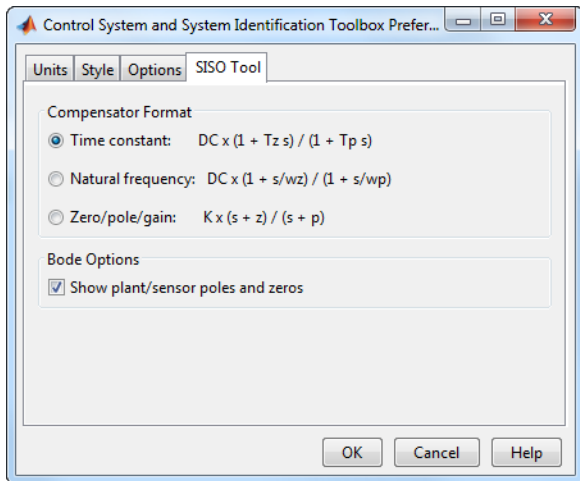
You can make the following selections:

- **Compensator Format** — Select the time constant, natural frequency, or zero/pole/gain format. The time constant format is a factorization of the compensator transfer function of the form

$$K \times \frac{\left(1 + T_{z_1}s\right)\left(1 + T_{z_2}s\right)}{\left(1 + T_{p_1}s\right)\left(1 + T_{p_2}s\right)}...$$

where K is compensator DC gain, Tz_1, Tz_2, ..., are the zero time constants, and Tp_1, Tp_2, ..., are the pole time constants.

The natural frequency format is

$$K \times \frac{\left(1 + s/\omega_{z_1}\right)\left(1 + s/\omega_{z_2}\right)}{\left(1 + s/\omega_{p_1}\right)\left(1 + s/\omega_{p_2}\right)}...$$

where K is compensator DC gain, ω_{z1}, and ω_{z2}, ... and ω_{p1}, ω_{p2}, ..., are the natural frequencies of the zeros and poles, respectively.

The zero/pole/gain format is

$$K \times \frac{(s + z_1)(s + z_2)}{(s + p_1)(s + p_2)}$$

where K is the overall compensator gain, and z_1, z_2, ... and p_1, p_2, ..., are the zero and pole locations, respectively.

- **Bode Options** — By default, the **Control System Designer** shows the plant and sensor poles and zeros as blue x's and o's, respectively. Clear this box to eliminate the plant's poles and zeros from the Bode plot. Note that the compensator poles and zeros (in red) will still appear.

Control Design Applications

Using Identified Models for Control Design Applications

How Control System Toolbox Software Works with Identified Models

System Identification Toolbox software integrates with Control System Toolbox software by providing a plant for control design.

Control System Toolbox software also provides the Linear System Analyzer to extend System Identification Toolbox functionality for linear model analysis.

Control System Toolbox software supports only linear models. If you identified a nonlinear plant model using System Identification Toolbox software, you must linearize it before you can work with this model in the Control System Toolbox software. For more information, see the `linapp`, `idnlarx/linearize`, or `idnlhw/linearize` reference page.

Note You can only use the System Identification Toolbox software to linearize nonlinear ARX (`idnlarx`) and Hammerstein-Wiener (`idnlhw`) models. Linearization of nonlinear grey-box (`idnlgrey`) models is not supported.

Using balred to Reduce Model Order

In some cases, the order of your identified model might be higher than necessary to capture the dynamics. If you have the Control System Toolbox software, you can use `balred` to compute a state-spate model approximation with a reduced model order.

To learn how you can reduce model order using pole-zero plots, see "Reducing Model Order Using Pole-Zero Plots" on page 17-71.

Compensator Design Using Control System Toolbox Software

After you estimate a plant model using System Identification Toolbox software, you can use Control System Toolbox software to design a controller for this plant.

System Identification Toolbox models in the MATLAB workspace are immediately available to Control System Toolbox commands. However, if you used the System Identification app to estimate models, you must first export the models to the MATLAB workspace. To export a model from the app, drag the model icon to the **To Workspace** rectangle. Alternatively, right-click the icon to open the Data/model Info dialog box. Click **Export** to export the model.

Control System Toolbox software provides both the **Control System Designer** and commands for working at the command line. You can import linear models directly into **Control System Designer** using the following command:

```
controlSystemDesigner(model)
```

You can also identify a linear model from measured SISO data and tune a PID controller for the resulting model in the PID Tuner. You can interactively adjust the identified parameters to obtain an LTI model whose response fits your response data. The PID Tuner automatically tunes a PID controller for the identified model. You can then interactively adjust the performance of the tuned control system, and save the identified plant and tuned controller. To access the PID Tuner, enter

`pidTuner` at the MATLAB command line. For more information, see "PID Controller Tuning" (Control System Toolbox).

Converting Models to LTI Objects

You can convert linear identified models into numeric LTI models (`ss`, `tf`, `zpk`) of Control System Toolbox software.

The following table summarizes the commands for transforming linear state-space and polynomial models to an LTI object.

Commands for Converting Models to LTI Objects

Command	Description	Example
frd	Convert to frequency-response representation.	ss_sys = frd(model)
ss	Convert to state-space representation.	ss_sys = ss(model)
tf	Convert to transfer-function form.	tf_sys = tf(model)
zpk	Convert to zero-pole form.	zpk_sys = zpk(model)

The following code converts the noise component of a linear identified model, `sys`, to a numeric state-space model:

```
noise_model_ss = idss(sys,'noise');
```

To convert both the measured and noise components of a linear identified model, `sys`, to a numeric state-space model:

```
model_ss = idss(sys,'augmented');
```

For more information about subreferencing the dynamic or the noise model, see "Separation of Measured and Noise Components of Models" on page 4-33.

Viewing Model Response Using the Linear System Analyzer

What Is the Linear System Analyzer?

If you have the Control System Toolbox software, you can plot models in the Linear System Analyzer from either the System Identification app or the MATLAB Command Window.

The Linear System Analyzer is a graphical user interface for viewing and manipulating the response plots of linear models.

Note The Linear System Analyzer does not display model uncertainty.

For more information about working with plots in the Linear System Analyzer, see the "Linear System Analyzer Overview" (Control System Toolbox).

Displaying Identified Models in the Linear System Analyzer

When the MATLAB software is installed, the System Identification app contains the **To LTI Viewer** rectangle. To plot models in the Linear System Analyzer, do one of the following:

- Drag and drop the corresponding icon to the **To LTI Viewer** rectangle in the System Identification app.
- Right-click the icon to open the Data/model Info dialog box. Click **Show in LTI Viewer** to plot the model in the Linear System Analyzer.

Alternatively, use the following syntax when working at the command line to view a model in the Linear System Analyzer:

```
linearSystemAnalyzer(model)
```

Combining Model Objects

If you have the Control System Toolbox software, you can combine linear model objects, such as `idtf`, `idgrey`, `idpoly`, `idproc`, and `idss` model objects, similar to the way you combine LTI objects. The result of these operations is a numeric LTI model that belongs to the Control System Toolbox software. The only exceptions are the model stacking and model concatenation operations, which deliver results as identified models.

For example, you can perform the following operations on identified models:

- `G1+G2`
- `G1*G2`
- `append(G1,G2)`
- `feedback(G1,G2)`

See Also

Related Examples
- "Create and Plot Identified Models Using Control System Toolbox Software" on page 19-5

Create and Plot Identified Models Using Control System Toolbox Software

This example shows how to create and plot models using the System Identification Toolbox software and Control System Toolbox software. The example requires a Control System Toolbox license.

Construct a random numeric model using the Control System Toolbox software.

```
rng('default');
sys0 = drss(3,3,2);
```

`rng('default')` specifies the setting of the random number generator as its default setting.

`sys0` is a third-order numeric state-space model with three outputs and two inputs.

Convert `sys0` to an identified state-space model and set its output noise variance.

```
sys = idss(sys0);
sys.NoiseVariance = 0.1*eye(3);
```

Generate input data for simulating the output.

```
u = iddata([],idinput([800 2],'rbs'));
```

Simulate the model output with added noise.

```
opt = simOptions('AddNoise',true);
y = sim(sys,u,opt);
```

`opt` is an option set specifying simulation options. `y` is the simulated output for `sys0`.

Create an input-output (`iddata`) object.

```
data = [y u];
```

Estimate the state-space model from the generated data using `ssest` .

```
estimated_ss = ssest(data(1:400));
```

`estimated_ss` is an identified state-space model.

Convert the identified state-space model to a numeric transfer function.

```
sys_tf = tf(estimated_ss);
```

Plot the model output for identified state-space model.

```
compare(data(401:800),estimated_ss)
```

Simulated Response Comparison
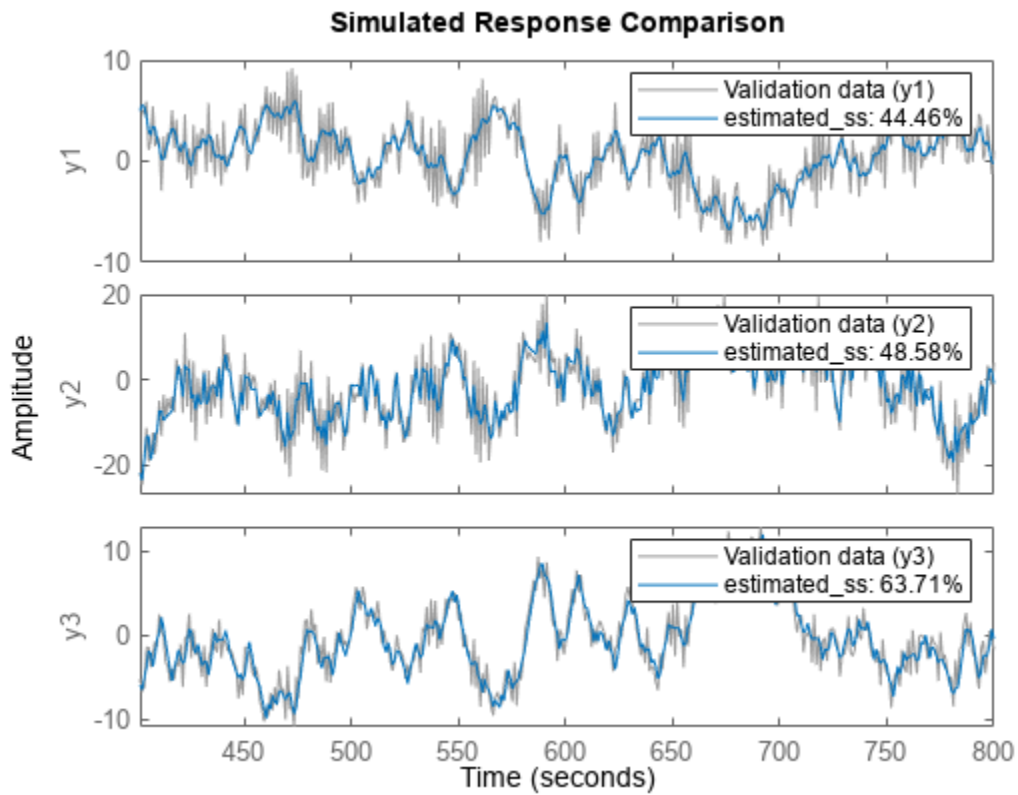
Plot the response of identified model using the Linear System Analyzer.

```
linearSystemAnalyzer(estimated_ss);
```
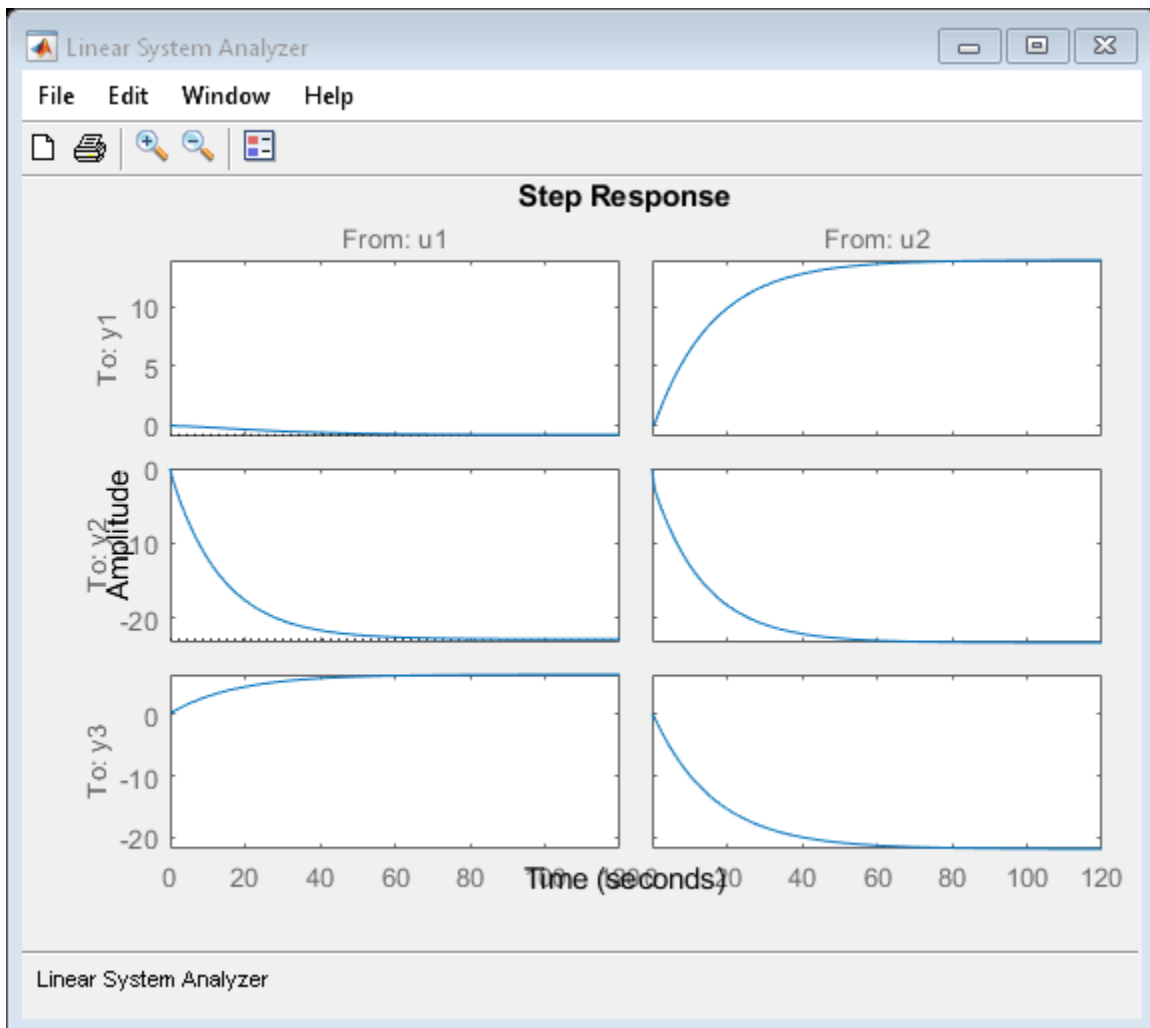
See Also

More About

- "Using Identified Models for Control Design Applications" on page 19-2

20

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, enter `slLibraryBrowser` at the MATLAB prompt. In the Library Browser, select **System Identification Toolbox**.

You can also open the System Identification Toolbox block library directly by typing the following command at the MATLAB prompt:

```
slident
```

To get help on a block, right-click the block in the Library Browser, and select **Help**.

See Also

More About

- "Simulate Identified Model in Simulink" on page 20-5

Preparing Data

The following table summarizes the blocks you use to transfer data between the MATLAB and Simulink environments.

After you add a block to the Simulink model, double-click the block to specify block parameters. For an example of bringing data into a Simulink model, see the tutorial on estimating process models in the *System Identification Toolbox Getting Started Guide*.

Block	Description
Iddata Sink	Export input and output signals to the MATLAB workspace as an `iddata` object.
Iddata Source	Import `iddata` object from the MATLAB workspace. Input and output ports of the block correspond to input and output signals of the data. These inputs and outputs provide signals to blocks that are connected to this data block.

For information about configuring each block, see the corresponding reference pages.

Identifying Linear Models

The following table summarizes the blocks you use to recursively estimate model parameters in a Simulink model during simulation and export the results to the MATLAB environment.

After you add a block to the model, double-click the block to specify block parameters.

Block	Description
Recursive Least Squares Estimator	Estimate model coefficients using recursive least squares (RLS) algorithm
Recursive Polynomial Model Estimator	Estimate input-output and time-series model coefficients
Kalman Filter	Estimate states of discrete-time or continuous-time linear system
Model Type Converter	Convert polynomial model coefficients to state-space model matrices

For information about configuring each block, see the corresponding reference pages.

Simulate Identified Model in Simulink

After estimating a model at the command line or in the **System Identification** app, you can import the model from the MATLAB workspace into Simulink using model blocks. You can then simulate the model output for the initial conditions and the model inputs that you specify.

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 model
- Replace a complex Simulink subsystem with a simpler data-based model

Summary of Simulation Blocks

The following model blocks are available in the System Identification Toolbox library in Simulink.

Block	Description
Idmodel	Simulate a linear identified model in Simulink software. The model can be a process (`idproc`), linear polynomial (`idpoly`), state-space (`idss`), grey-box (`idgrey`), or transfer function (`idtf`) model.
Nonlinear ARX Model	Simulate `idnlarx` model in Simulink.
Hammerstein-Wiener Model	Simulate `idnlhw` model in Simulink.
Nonlinear Grey-Box Model	Simulate nonlinear ODE (`idnlgrey` model object) in Simulink.

In any of these model blocks, you specify the model-variable name of the identified model you want to import. You also specify initial conditions for simulation (see "Specifying Initial Conditions for Simulation" on page 20-5.) You can specify time-domain input data by:

- Using a From Workspace block, if one sample of input data is a scalar or vector
- Using an Iddata Source block, if the input data is in an `iddata` object

For detailed information about how to configure the blocks, see the corresponding block reference pages.

Specifying Initial Conditions for Simulation

When your model is not at rest at the start of the simulation, you must specify the initial conditions. The model may not start at rest, for example, when you are:

- Comparing your model response with measured output data or a previous simulation
- Continuing from a previous simulation
- Initiating your simulation during the steady-state phase
- Initiating your simulation from a predetermined operating condition

If you do not specify initial conditions, you introduce an error source that impacts the apparent performance of your model. This error may be transient; in a model that is lightly damped or includes an integral component, the error may not completely die out.

Specifying initial conditions requires two steps:

1 Determine what the initial conditions should be (see "Estimate Initial Conditions for Simulating Identified Models" on page 20-26)

2 Specify these values as parameters in the model block.

If you have a linear model that is not `idss` or `idgrey`, you also have to convert the model into state-space form before the first step.

If you have already performed a simulation using `compare`, `sim`, or the **System Identification** app, and you want to check out your Simulink implementation by precisely reproducing your earlier results, see "Reproduce Command Line or System Identification App Simulation Results in Simulink" on page 20-15.

Specifying Initial States of Linear Models

To specify initial states for state-space linear models (`idss`, `idgrey`), use the **Initial states (state space only: idss, idgrey)** parameter in the Idmodel block. Initial states must be a vector of length equal to the order of the model.

Other types of linear model, such as transfer-function form models (`idtf`), do not use an explicit state representation. The model blocks therefore have no input for initial state specification; the software assumes initial conditions of zero. To specify initial conditions for one of these models, first convert your model `m` into state-space form `mss` at the command line.

```
mss = idss(m);
```

Then use `mss` in the **Identified model**parameter of the Idmodel block. You can now specify your initial states as described in the previous paragraph for state-space models.

Simulate Identified Linear Model in Simulink with Initial Conditions

This example shows how to set the initial states for simulating a linear model such that the simulation provides a best fit to measured input-output data.

You first estimate a model `M` using a multiple-experiment data set `Z`, which contains data from three experiments - `z1`, `z2`, and `z3`.

Load the multi-experiment data.

```
load(fullfile(matlabroot,'toolbox','ident','iddemos',...
'data', 'twobodiesdata'))
```

Create an `iddata` object to store the multi-experiment data. Set `'Tstart'` to 0 so that the data start time matches the Simulink start time 0f 0 s.

```
z1=iddata(y1,u1,0.005,'Tstart',0);
z2=iddata(y2,u2,0.005,'Tstart',0);
z3=iddata(y3,u3,0.005,'Tstart',0);
Z = merge(z1,z2,z3);
```

Estimate a 5th order state-space model.

```
[M,x0] = n4sid(Z,5);
```

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 simulated response of M.

Extract the initial states that maximize the fit to the corresponding output y2, and simulate the model in Simulink using the second experiment, z2.

```
X0est = x0(:,2);
```

Open a preconfigured Simulink model.

```
mdl = 'ex_idmodel_block';
open_system(mdl)
```
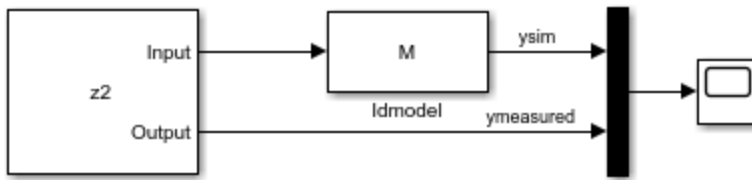

Copyright 2000-2021 The MathWorks Inc

The model uses the Iddata Source, Idmodel, and Scope blocks. The following block parameters have been preconfigured to specify the simulation data, estimated model, and initial conditions.

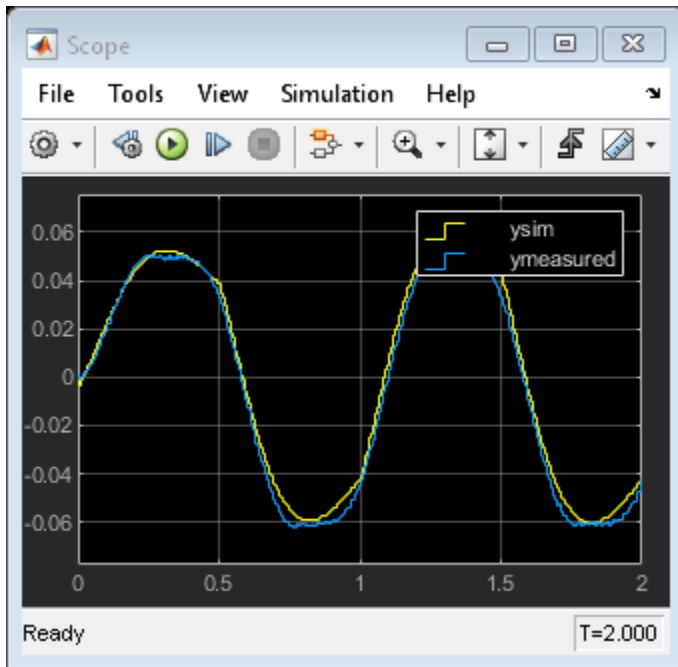**Block parameters of Iddata Source block:**

- **IDDATA Object** - z2

Block parameters of Idmodel block:

- **Identified Model** - M
- **Initial states** - X0est

Simulate the model for two seconds, and compare the simulated output ysim with the measured output ymeasured using the Scope block.

```
simOut = sim(mdl);
open_system([mdl '/Scope'])
```

Compare this result with the result you would have gotten without setting initial conditions. You can nullify X0 in the **Initial States** field of the Simulink model block by replacing the X0est variable with 0. You can also nullify X0est itself in command line, as shown here.

```
X0est = 0*X0est;
```

Run the simulation, and view the new results.

```
simOut = sim(mdl);
open_system([mdl '/Scope'])
```
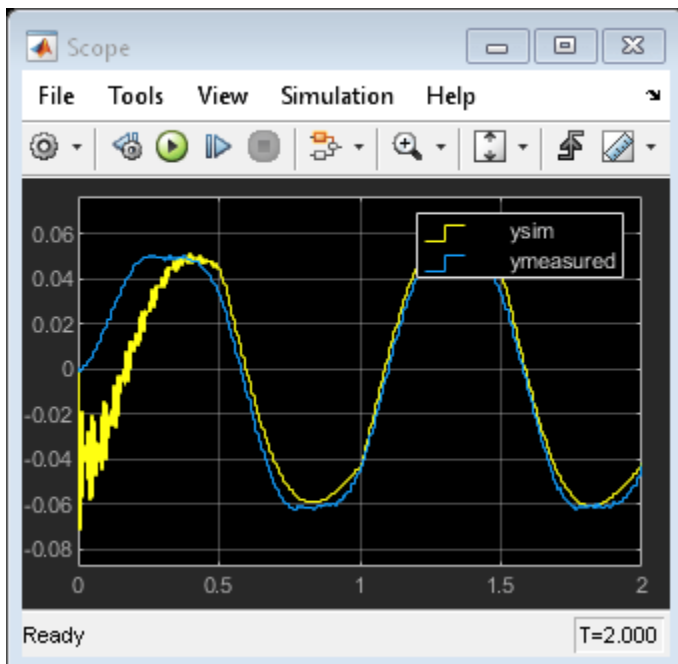
In this plot, the simulated output has a significant transient at the start, but this transient settles out in time.

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 Hammerstein-Wiener Model in Simulink

Compare the simulated output of a Hammerstein-Wiener Model block to the measured output of a system. You improve the agreement between the measured and simulated responses by estimating initial state values.

Load the sample data.

```
load twotankdata
```

Create an `iddata` object from the sample data. Set `'Tstart'` to 0 so that the data start time matches the Simulink start time 0f 0 s.

```
z1 = iddata(y,u,0.2,'Tstart',0,'Name','Two tank system');
```

Estimate a Hammerstein-Wiener model using the data.

```
mw1 = nlhw(z1,[1 5 3],idPiecewiseLinear,idPiecewiseLinear);
```

You can now simulate the output of the estimated model in Simulink using the input data in `z1`. To do so, open a preconfigured Simulink model.

```
model = 'ex_idnlhw_block';
open_system(model);
```


Copyright 2000-2021 The MathWorks Inc.

The model uses the Iddata Source, Hammerstein-Wiener Model, and Scope blocks. The following block parameters have been preconfigured to specify the estimation data, estimated model, and initial conditions:

Block parameters of Iddata Source block:

- **IDDATA Object** - `z1`

Block parameters of Hammerstein-Wiener Model block:

- **Model** - `mw1`
- **Initial conditions** - `Zero` (default)

Run the simulation.

View the difference between measured output and model output by using the Scope block.

```
simOut = sim(model);
open_system([model '/Scope'])
```


To improve the agreement between the measured and simulated responses, estimate an initial state vector for the model from the estimation data `z1`, using `findstates`. Specify the maximum number of iterations for estimation as 100. Specify the prediction horizon as `Inf`, so that the algorithm computes the initial states that minimize the simulation error.

```
opt = findstatesOptions;
opt.SearchOptions.MaxIterations = 100;
x0 = findstates(mw1,z1,Inf,opt);
```

Set the **Initial conditions** block parameter value of the Hammerstein-Wiener Model block to `State`
`Values`. The default initial states are `x0`.

```
set_param([model '/Hammerstein-Wiener Model'],'IC','State values');
```

Run the simulation again, and view the difference between measured output and model output in the
Scope block. The difference between the measured and simulated responses is now reduced.

```
simOut = sim(model);
```


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.

Simulate Nonlinear ARX Model in Simulink

This example shows how to compare the simulated output of a Nonlinear ARX Model block to the measured output of a system. You improve the agreement between the measured and simulated responses by estimating initial state values.

Load the sample data and create an `iddata` object. Set `'Tstart'` to 0 so that the data start time matches the Simulink start time 0f 0 s.

```
load twotankdata
z = iddata(y,u,0.2,'Tstart',0,'Name','Two tank system');
z1 = z(1:1000);
```

Estimate a nonlinear ARX model.

```
mnlarx1 = nlarx(z1,[5 1 3],idWaveletNetwork(8));
```

You can now simulate the output of the estimated model in Simulink using the input data in `z1`. To do so, open a preconfigured Simulink model.

```
model = 'ex_idnlarx_block';
open_system(model);
```


Copyright 2000-2021 The MathWorks Inc.

The model uses the Iddata Source, Nonlinear ARX Model, and Scope blocks. The following block parameters have been preconfigured to specify the estimation data, estimated model, and input and output levels:

Block parameters of Iddata Source block:

- **IDDATA Object** - `z1`

Block parameters of Nonlinear ARX Model block:

- **Model** - `mnlarx1`
- **Initial conditions** - Input and output values (default)

- **Input level** - 10
- **Output level** - 0.1

Run the simulation.

View the difference between measured output and model output by using the Scope block.

```
simOut = sim(model);
open_system([model '/Scope'])
```
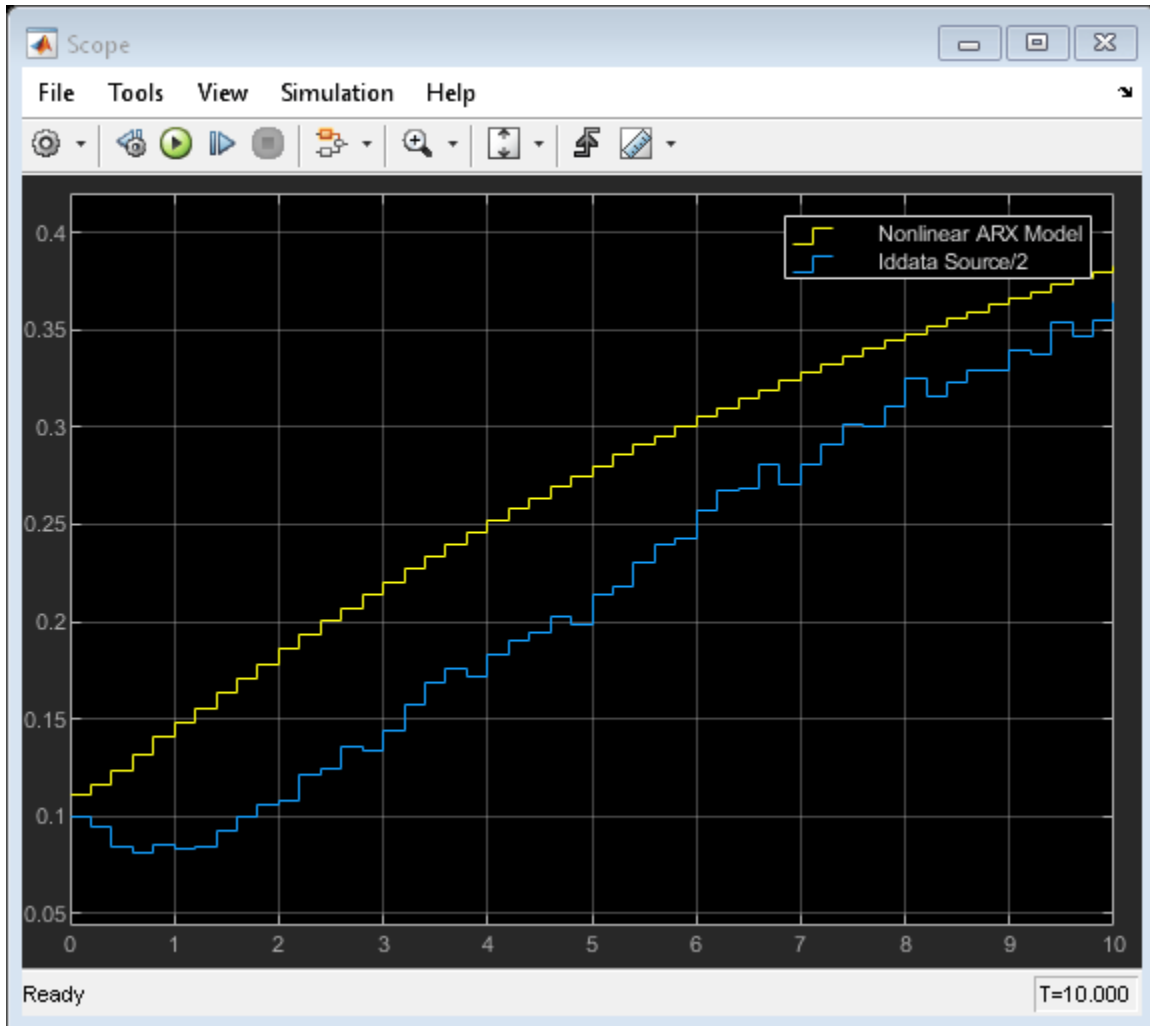

To improve the agreement between the measured and simulated responses, estimate an initial state vector for the model from the estimation data, z1.

```
x0 = findstates(mnlarx1,z1,Inf);
```

Set the **Initial Conditions** block parameter value of the Nonlinear ARX Model block to `State Values`. Specify initial states as `x0`.

```
set_param([model '/Nonlinear ARX Model'],'ICspec','State values','X0','x0');
```

Run the simulation again, and view the difference between measured output and model output in the Scope block. The difference between the measured and simulated responses is now reduced.

```
simOut = sim(model);
```
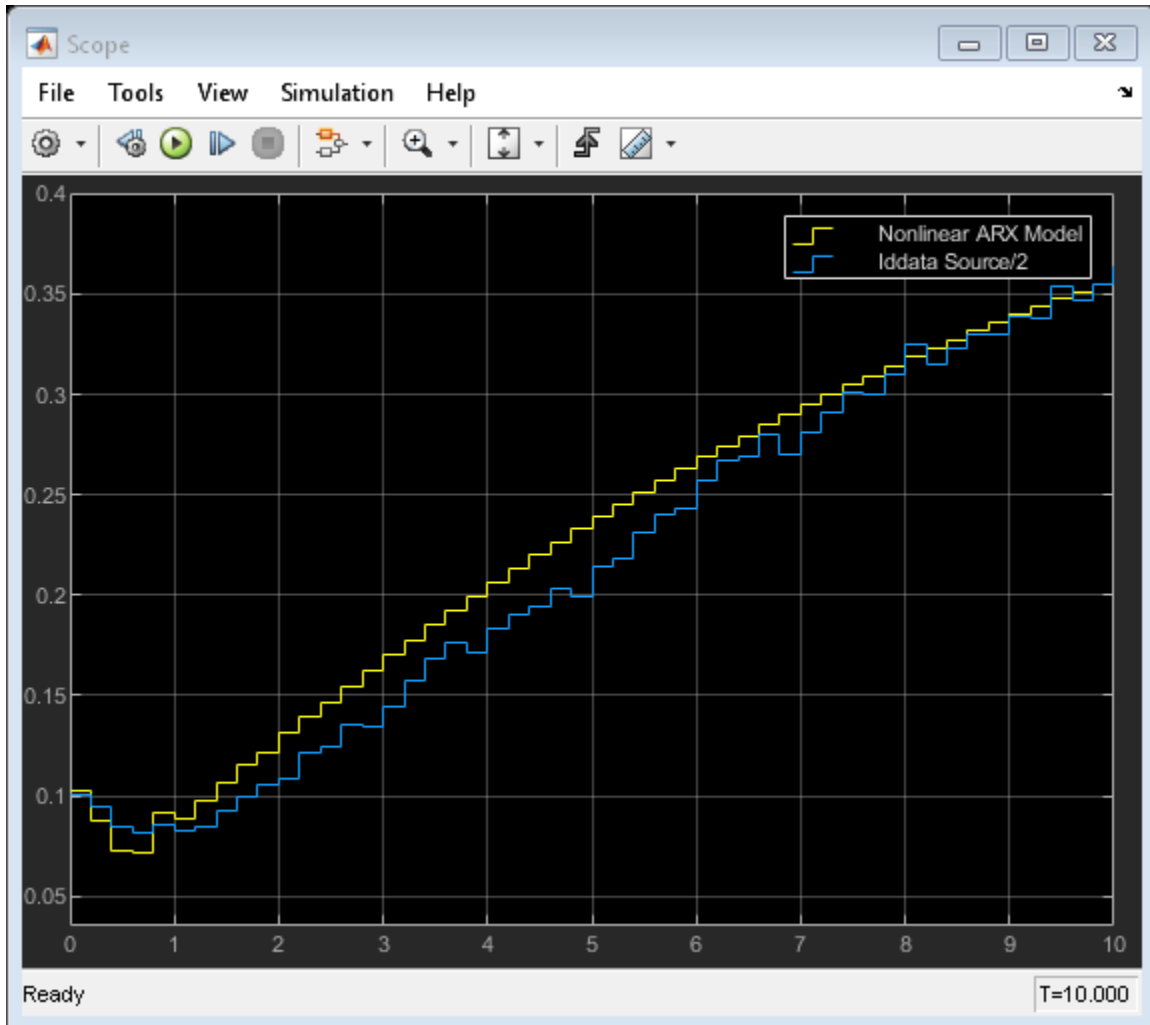

See Also
IDNLHW Model | IDNLARX Model | Iddata Sink | Iddata Source | Idmodel

More About

*   "Simulate and Predict Identified Model Output" on page 17-6
*   "Estimate Initial Conditions for Simulating Identified Models" on page 20-26
*   "Reproduce Command Line or System Identification App Simulation Results in Simulink" on page 20-15

Reproduce Command Line or System Identification App Simulation Results in Simulink

Once you identify a model, you can simulate it at the command line, in the **System Identification** app, or in Simulink. If you start with one simulation method, and migrate to a second method, you may find that the results do not precisely match. This mismatch does not mean that one of the simulations is implemented incorrectly. Each method uses a unique simulation algorithm, and yields results with small numeric differences.

Generally, command-line simulation functions such as `compare` and `sim`, as well as the **System Identification** app, use similar default settings and solvers. Simulink uses somewhat different default settings and a different solver. If you want to validate your Simulink implementation against command-line or app simulation results, you must ensure that your Simulink settings are consistent with your command-line or app simulation settings. The following paragraphs list these settings in order of decreasing significance.

Initial Conditions

The most significant source of variation when trying to reproduce results is usually the initial conditions. The initial conditions that you specify for simulation in Simulink blocks must match the initial conditions used by `compare`, `sim`, or the **System Identification** app. You can determine the initial conditions returned in the earlier simulations. Or, if you are simulating against measurement data, you can estimate a new set of initial conditions that best match that data. Then, apply those conditions to any simulation methods that you use. For more information on estimating initial conditions, see "Estimate Initial Conditions for Simulating Identified Models" on page 20-26.

Once you have determined the initial conditions `x0` to use for a model `m` and a data set `z`, implement them in your simulation tool.

- For `compare` or `sim`, use `compareOptions` or `simOptions`.
- For Simulink, use the Idmodel, Nonlinear ARX Model, or Hammerstein-Wiener Model block, specifying `m` for the model and `x0` for the initial states. With Idmodel structures, you can specify initial states only for `idss` and `idgrey` models. If your linear model is of any other type, convert it first to `idss`. See "Simulate Identified Model in Simulink" on page 20-5 or the corresponding block reference pages.
- For the **System Identification** app, you cannot specify initial conditions other than zero. You can specify only the method of computing them.

Start Time of Input Data

In Simulink, the simulation default start time is zero, while in the app or command line the default start time is the sample time of the data.

Before exporting an `iddata` object for simulation in Simulink, set the object `Tstart` property to zero. Then export the object to Simulink using an Iddata Source block.

Discretization of Continuous-Time Data

Simulink software assumes a first-order-hold interpolation on input data.

20-15

When using `compare`, `sim`, or the app, set the `InterSample` property of your `iddata` object to `'foh'`.

Solvers for Continuous-Time Models

The `compare` command and the app simulate a continuous-time model by first discretizing the model using `c2d` and then propagating the difference equations. Simulink defaults to a variable-step true ODE solver. To better match the difference-equation propagation, set the solver in Simulink to a fixed-step solver such as `ode5`.

Match Output of sim Command and Nonlinear ARX Model Block

Reproduce command-line `sim` results for an estimated nonlinear system in Simulink. When you have an estimated system model, you can simulate it either with the command-line `sim` command or with Simulink. The two outputs do not match precisely unless you base both simulations on the same initial conditions. You can achieve this match by estimating the initial conditions in your MATLAB model and applying both in the `sim` command and in your Simulink model.

Since you are directly comparing the `sim` and Simulink results, you also need to synchronize the timestamps of the first data points by setting the first time stamp in the data, `Tstart` to 0. You set this time because, by default, command-line simulation assigns the sample time to the first timestamp, while Simulink assigns a timestamp of 0 seconds.

Load the estimation data you are using to identify a nonlinear ARX model. Create an `iddata` data object from the data. Specify the sample time as 0.2 seconds, and `Tstart` as 0 seconds.

```
load twotankdata
z = iddata(y,u,0.2,'Tstart',0);
```

Use the first 1000 data points to estimate a nonlinear ARX model `mw1` with orders [5 1 3] and wavelet network nonlinearity.

```
z1 = z(1:1000);
mw1 = nlarx(z1,[5 1 3],idWaveletNetwork);
```

To initialize your simulated response consistently with the measured data, estimate an initial state vector `x0` from the estimation data `z1` using the `findstates` function. The third argument for `findstates` in this example, `Inf` , sets the prediction horizon to infinity in order to minimize the simulation error.

```
x0 = findstates(mw1,z1,Inf);
```

Simulate the model `mw1` output using `x0` and the first 50 seconds (250 samples) of input data.

```
opt = simOptions('InitialCondition',x0);
data_sim = sim(mw1,z1(1:251),opt);
```

Now simulate the output of `mw1` in Simulink using a Nonlinear ARX Model block, and specify the same initial conditions in the block. The Simulink model `ex_idnlarx_block_match_sim` is preconfigured to specify the estimation data, nonlinear ARX model, initial conditions `x0`, and a simulation time of 50 seconds.

Open the Simulink model.

```
model = 'ex_idnlarx_block_match_sim';
open_system(model);
```
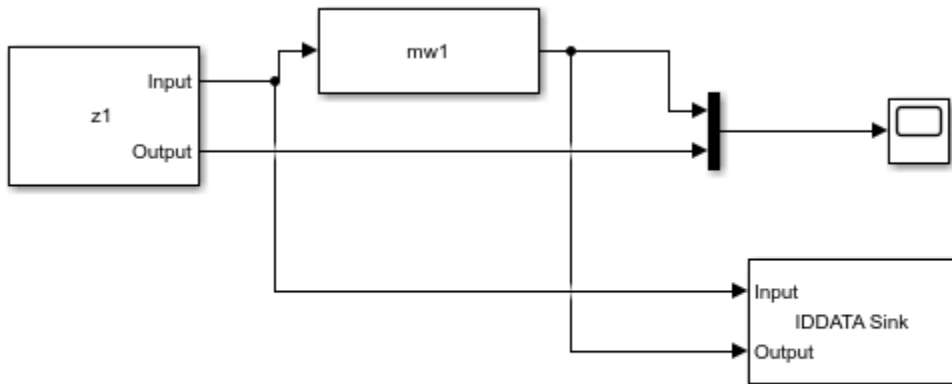

Copyright 2000-2021 The MathWorks Inc.

Simulate the nonlinear ARX model output for 50 seconds.

```
sim(model);
```

The IDDATA Sink block outputs the simulated output, `data`, to the MATLAB® workspace.

Plot and compare the simulated outputs you obtained using the `sim` command and Nonlinear ARX block.

```
figure
plot(data_sim.SamplingInstants,data_sim.OutputData,'b*',...
    data.SamplingInstants,data.OutputData,'-r')
xlabel('Time(sec)');
ylabel('Simulated output');
legend('Output of sim command','Output of Nonlinear ARX block','location','southeast')
title('Estimated Initial Conditions for Simulink')
```
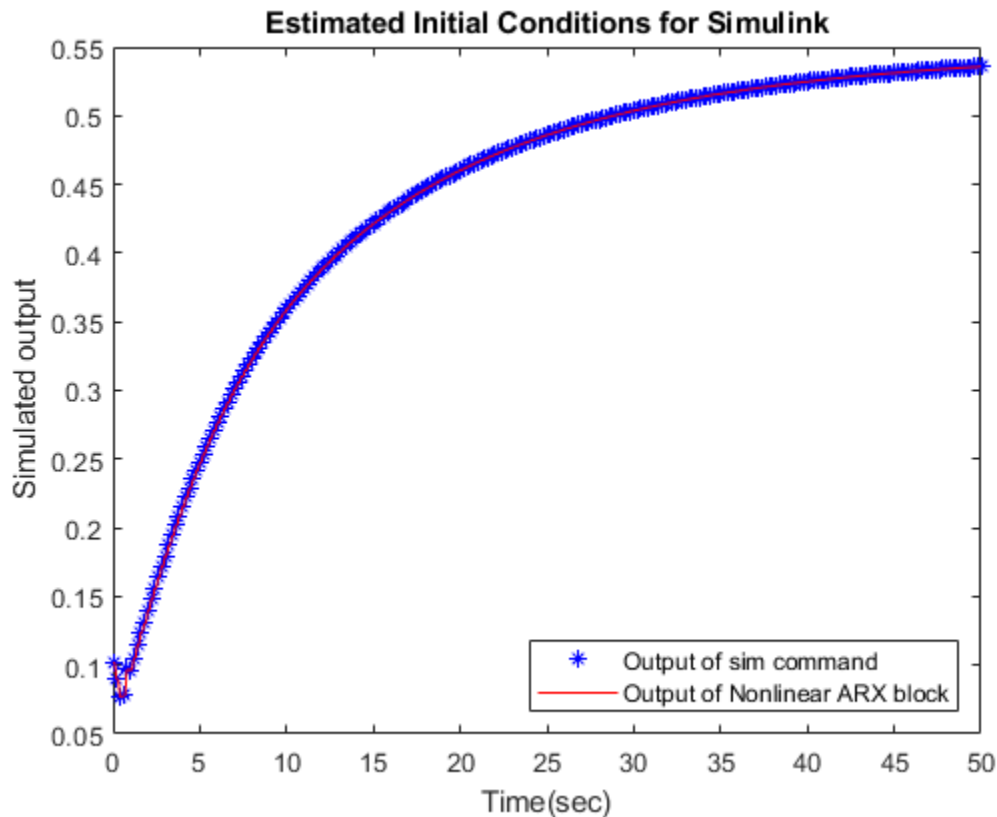
The simulated outputs are the same because you specified the same initial condition when using `sim` and the Nonlinear ARX Model block.

If you want to see what the comparison would look like without using the same initial conditions, you can rerun the Simulink version with no initial conditions set.

Nullify initial-condition vector `x0`, and simulate for 50 seconds as before. This null setting of `x0` is equivalent to the Simulink initial conditions default.

```
x0 = [0;0;0;0;0;0;0;0];
sim(model);
```

Plot the difference between the command-line and Simulink methods for this case.

```
figure
plot(data_sim.SamplingInstants,data_sim.OutputData,'b*',...
    data.SamplingInstants,data.OutputData,'-r')
xlabel('Time(sec)');
ylabel('Simulated output');
legend('Output of sim command','Output of Nonlinear ARX block','location','southeast')
title('Default Initial Conditions for Simulink')
```
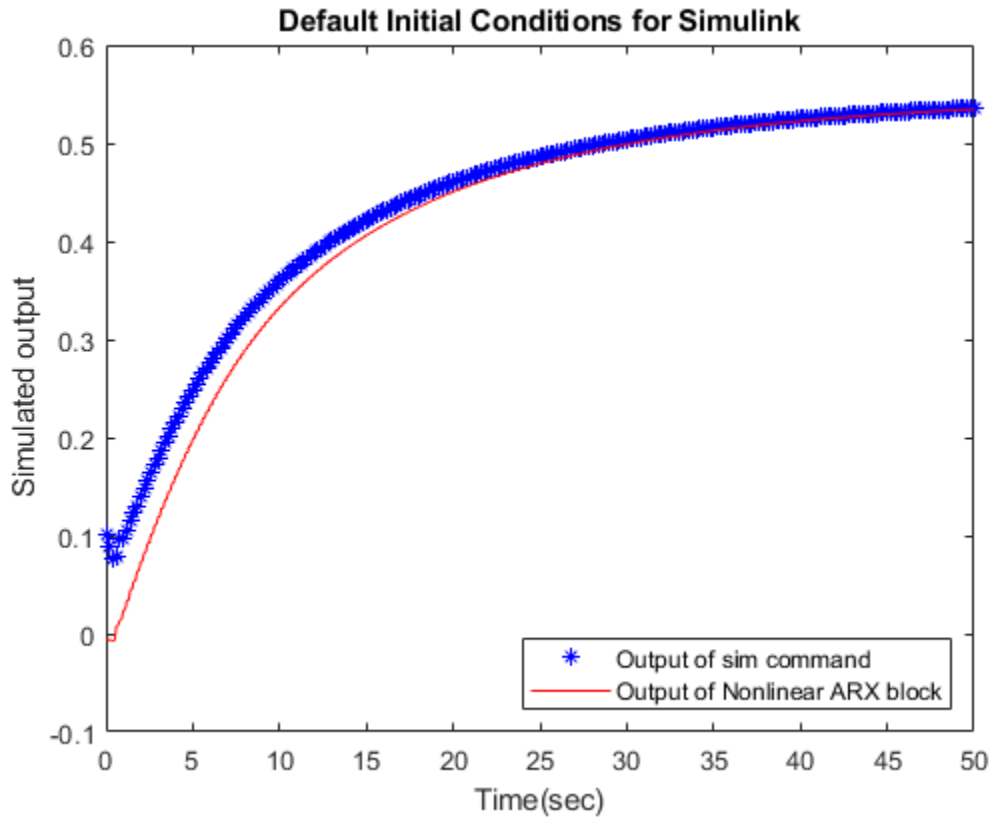
The Simulink version starts at a different point than the `sim` version, but the two versions eventually converge. The sensitivity to initial conditions is important for verifying reproducibility of the model, but is usually a transient effect.

Match Output of compare Command and Nonlinear ARX Model Block

In this example, you first use the `compare` command to match the simulated output of a nonlinear ARX model to measured data. You then reproduce the simulated output using a Nonlinear ARX Model block.

Load estimation data to estimate a nonlinear ARX model. Create an `iddata` object from the data. Specify the sample time as 0.2 seconds and the start time of the data as 0 seconds.

```
load twotankdata
z = iddata(y,u,0.2,'Tstart',0);
```

Use the first 1000 data points to estimate a nonlinear ARX model `mw1` with orders [5 1 3] and wavelet network nonlinearity.

```
z1 = z(1:1000);
mw1 = nlarx(z1,[5 1 3],idWaveletNetwork);
```

Use the `compare` command to match the simulated response of `mw1` to the measured output data in `z1`.

```
data_compare = compare(mw1,z1);
```

The `compare` function uses the first `nx` samples of the input-output data as past data, where `nx` is the largest regressor delay in the model. `compare` uses this past data to compute the initial states, and then simulates the model output for the remaining `nx+1:end` input data.

To reproduce this result in Simulink, compute the initial state values used by `compare`, and specify the values in the Nonlinear ARX Model block. First compute the largest regressor delay in the model. Use this delay to compute the past data for initial condition estimation.

```
nx = max(getDelayInfo(mw1));
past_data = z1(1:nx);
```

Use `data2state` to get state values using the past data.

```
x0 = data2state(mw1,z1(1:nx));
```

Now simulate the output of `mw1` in Simulink using a Nonlinear ARX Model block, and specify `x0` as the initial conditions in the block. Use the remaining `nx+1:end` data for simulation.

```
zz = z1(nx+1:end);
zz.Tstart = 0;
```

The Simulink model `ex_idnlarx_block_match_compare` is preconfigured to specify the estimation data, nonlinear ARX model, and initial conditions.

Open the Simulink model.

```
model = 'ex_idnlarx_block_match_compare';
open_system(model);
```
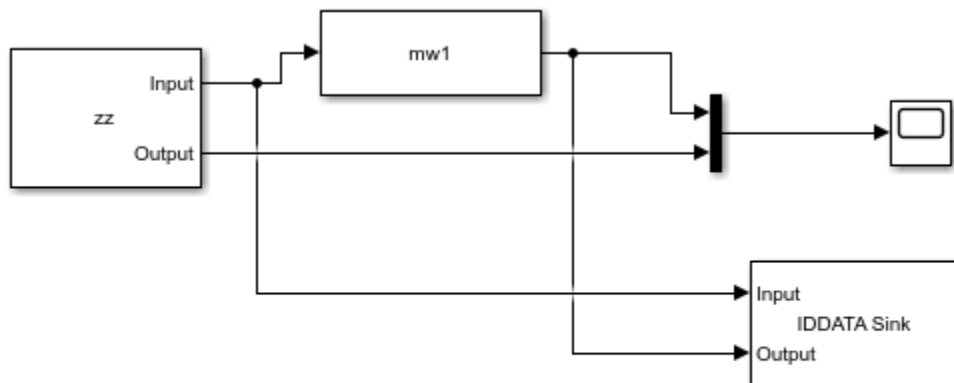

Copyright 2000-2021 The MathWorks Inc.

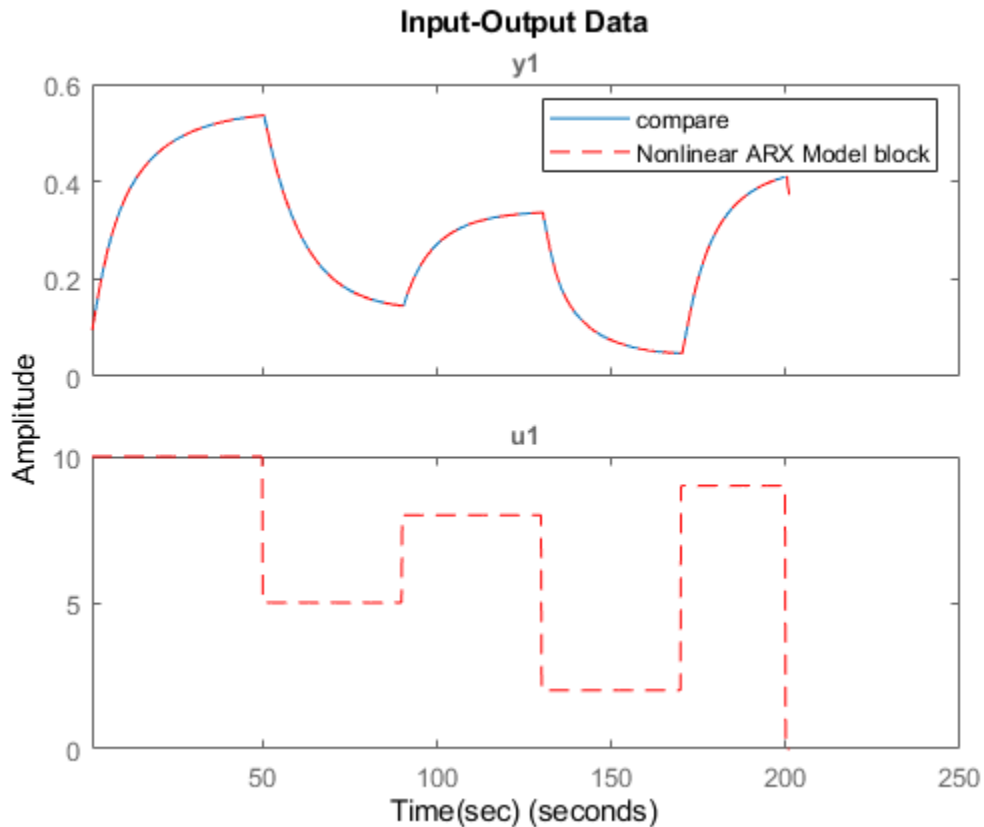Simulate the nonlinear ARX model output for 200 seconds.

```
sim(model);
```

The IDDATA Sink block outputs the simulated output, `data`, to the MATLAB® workspace.

Compare the simulated outputs you obtained using the `compare` command and Nonlinear ARX block. To do so, plot the simulated outputs.

To compare the output of `compare` to the output of the Nonlinear ARX Model block, discard the first `nx` samples of `data_compare`.

```
data_compare = data_compare(nx+1:end);
data.Tstart=1;
plot(data_compare,data,'r--')
xlabel('Time(sec)');
legend('compare','Nonlinear ARX Model block')
```


The simulated outputs are the same because you specified the same initial condition when using `compare` and the Nonlinear ARX Model block.

See Also

Nonlinear ARX Model | Hammerstein-Wiener Model | Nonlinear Grey-Box Model | `compare` | `sim`

More About

- "Simulate Identified Model in Simulink" on page 20-5
- "Estimate Initial Conditions for Simulating Identified Models" on page 20-26
- "Resolve Fit Value Differences Between Model Identification and compare Command" on page 20-22

Resolve Fit Value Differences Between Model Identification and compare Command

This example shows how NRMSE fit values computed by model identification functions and by the `compare` function can differ because of differences in initial conditions and prediction horizon settings.

When you identify a model, the model identification algorithm returns a value for the fit percentage to the measurement data you used. If you then use `compare` to plot the model simulation results against the measured data, fit results are not always identical. This difference is because:

- The model-identification algorithm and the `compare` algorithm use different methods to estimate initial conditions.
- The model-identification algorithm uses a default of one-step `prediction` focus, but `compare` uses a default of `Inf` — equivalent to a `simulation` focus. See "Simulate and Predict Identified Model Output" on page 17-6.

These differences are generally small, and they should not impact your model selection and validation workflow. If you do want to reproduce the estimation fit results precisely, you must reconcile the prediction horizon and initial condition settings used in `compare` with the values used during model estimation.

Load the data, and estimate a three-state state-space model.

```
load iddata1 z1;
sys = ssest(z1,3);
```

Estimate a state-space model from the measured data.

Allow `ssest` to use its default `'Focus'` setting of `'prediction'`, and its default `'InitialState'` estimation setting of 'auto'. The `'auto'` setting causes `ssest` to select:

- `'zero'` if the initial prediction error is not significantly higher than an estimation-based error, or if the system is FRD
- `'estimate'` for most single-experiment data sets, if `'zero'` is not acceptable
- `'backcast'` for multi-experiment data sets, if `'zero'` is not acceptable

For more information, see `ssestOptions`.

You can retrieve the automatically selected initial-state option, the initial state vector, and the NRMSE fit percentage from the estimation report for `sys`.

```
sys_is = sys.Report.InitialState

sys_is =
'zero'

sys_x0 = sys.Report.Parameters.X0

sys_x0 = 3×1

     0
     0
```

```
      0
```

```
sys_fit = sys.Report.Fit.FitPercent
```

```
sys_fit = 76.4001
```

ssest selected 'zero', rather than estimating the initial states, indicating that the contribution of the initial states to prediction error is small.

Use compare to simulate the model and plot the results against the measured data. For this case, allow compare to use its own method for estimating initial conditions and its default of Inf for the prediction horizon.
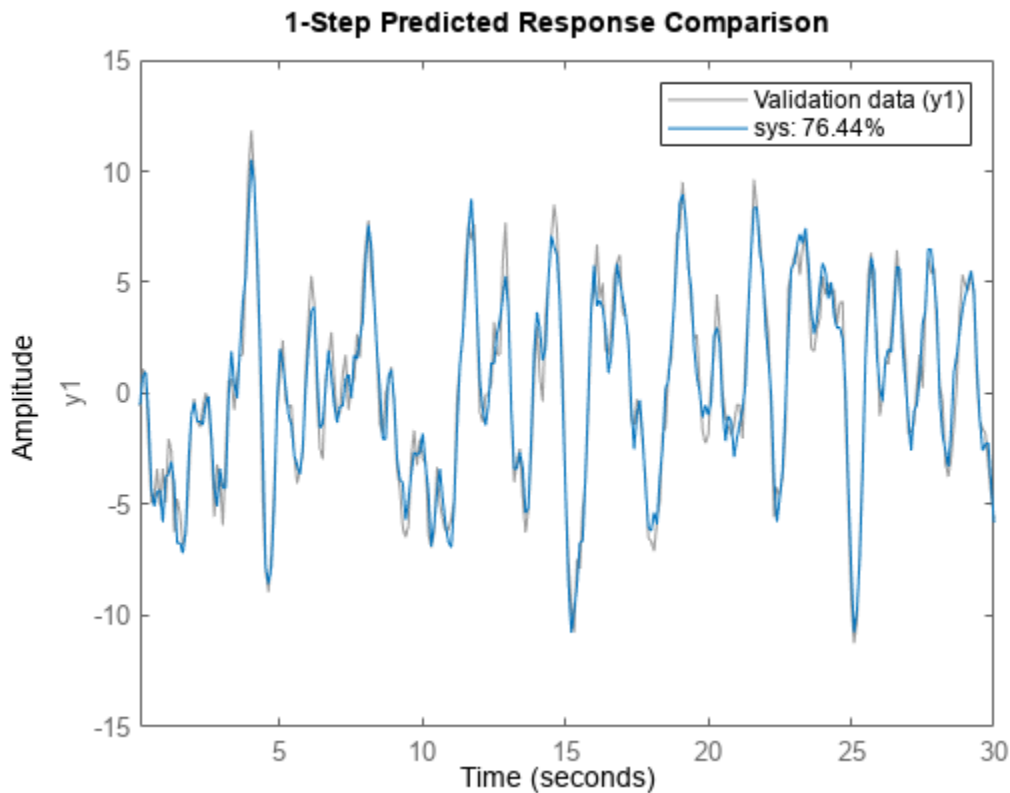
```
figure
compare(z1,sys)
```
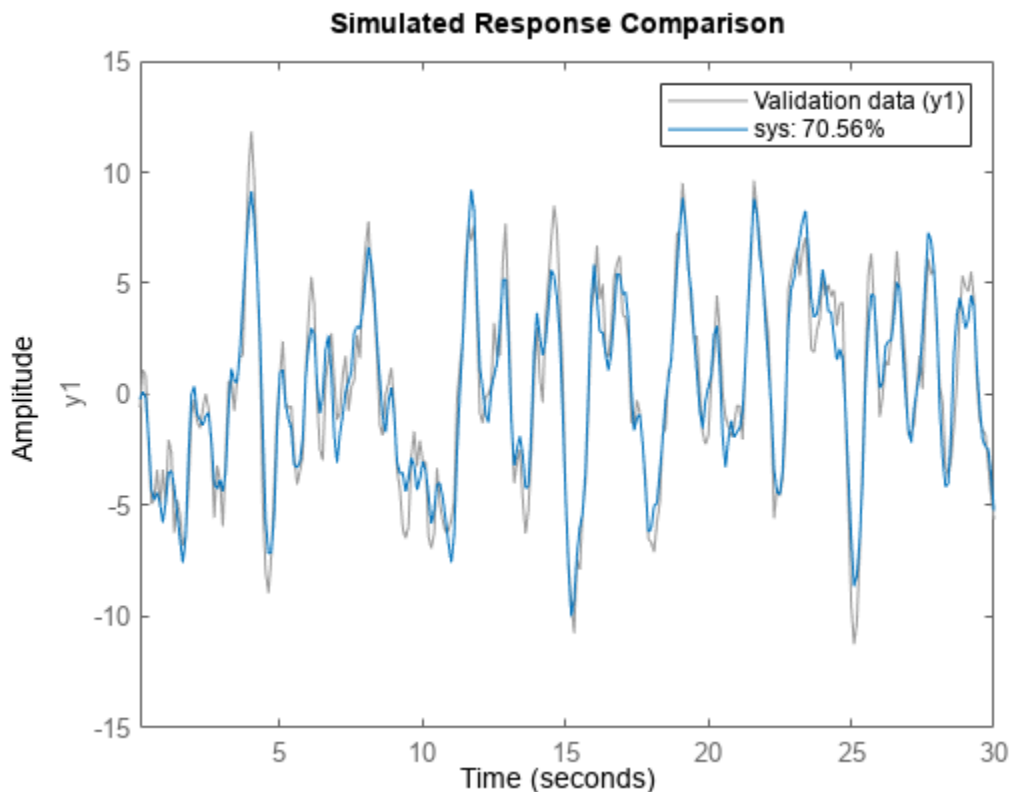

The NRMSE fit percentage computed by compare is lower than the fit that ssest computed. This lower value is primarily because of the mismatch between prediction horizons for ssest (default of 1) and compare (default of Inf).

Rerun compare using an argument of 1 for the prediction horizon.

```
figure
compare(z1,sys,1)
```

The change to the prediction horizon resolved most of the discrepancy between fit values. Now the `compare` fit exceeds the `ssest` fit. This higher value is because `ssest` set initial states to 0, while `compare` estimated initial states. You can investigate how far apart the initial-state vectors are by rerunning `compare` with output arguments.

```
[y1,fit1,x0c] = compare(z1,sys,1);
x0c
```
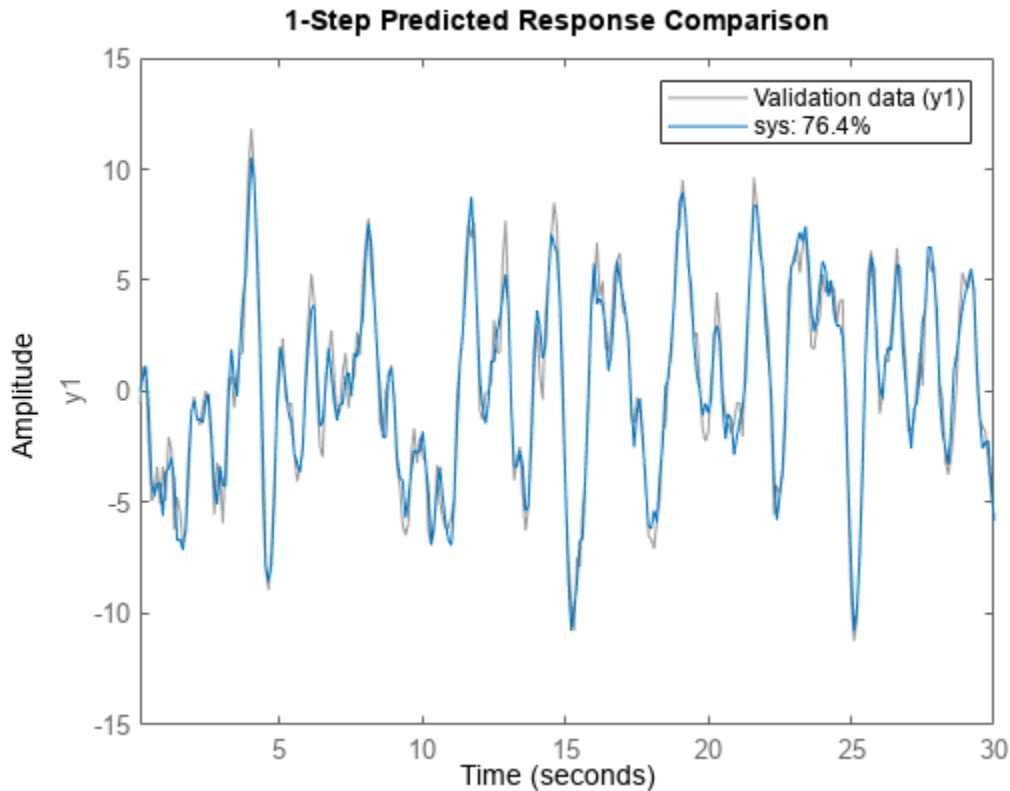
```
x0c = 3×1

   -0.0063
   -0.0012
    0.0180
```

x0c is small, but not zero like `sys_x0`. To reproduce the `ssest` initial conditions in `compare`, directly set the `'InitialCondition'` option to `sys_x0`. Then run `compare` using the updated option set.

```
compareOpt = compareOptions('InitialCondition',sys_x0);
figure
compare(z1,sys,1,compareOpt)
```

The fit result now matches the fit result from the original estimation.

This method confirms that the `ssest` algorithm and the `compare` algorithm yield the same results with the same initial-condition and prediction-horizon settings.

This example deals only with reproducing results when you want to confirm agreement between the estimation algorithm and the compare algorithm NRMSE fit values as they relate to the original estimation data. However, when you use `compare` to validate your model against validation data, or to compare the goodness of multiple candidate models, use the `compare` defaults for `'InitialCondition'` and prediction horizon as a general rule.

See Also
compare

Related Examples
- "Simulate and Predict Identified Model Output" on page 17-6
- "Estimate Initial Conditions for Simulating Identified Models" on page 20-26

Estimate Initial Conditions for Simulating Identified Models

When you simulate a dynamic system that does not start from rest, you must initialize that simulation to match the starting conditions. For example, you may be comparing your simulated response to measured data, or to a previous simulation or computation. You might want to start your simulation at a steady-state condition, or continue a simulation from a previous simulation.

Summary of Initial-Condition Estimation Approaches

System Identification Toolbox provides multiple techniques for determining initial conditions (ICs) to use. These techniques include using:

- Model identification results
- Current or past input/output data measurements
- State information from previous simulations

The following table summarizes what approaches are available and when to use them. For conciseness, this table uses the term "simulating" broadly to mean "simulating or predicting." For more information, see "Simulate and Predict Identified Model Output" on page 17-6.

Approaches for Estimating Initial Conditions

When to Use	Approach	Method Summary

You are simulating an `idss`,`idtf`, `idproc`, `idpoly`, or `idgrey` model response using the same input data with which you identified your model.	Extract ICs from model identification results (returned values or estimation report). Estimation returns IC information in one of the following forms.
	
	• An initial state vector (`idss` and `idgrey` models)
	• An `initialCondition` object (`idtf`, `idproc`, and `idpoly` models)
	
	See "Estimate Initial Conditions from Measured Data" on page 20-29.
		`[sys,x0est] = ssest(z,__)`
		
		`[sys,x0est] = greyest(z,__)`
		
		`x0est = sys.Report.Parameters.X0`
		
		`[sys,icest] = tfest(z,__)`
		
		`[sys,~,icest] = procest(z,__)`
		
		`[sys,icest] = idpoly(z,__)`
		
		Examples:
		
		"Resolve Fit Value Differences Between Model Identification and compare Command" on page 20-22
		
		"Apply Initial Conditions When Simulating Identified Linear Models" on page 20-45

You want to reproduce the results generated by `compare` (or generated by the **Model Output** plot in the app).	Use returned ICs from `compare`.
	
	See "Estimate Initial Conditions from Measured Data" on page 20-29.
		`[y,fit,x0] = compare(sys,z)`
		
		`[y,fit,ic] = compare(sys,z)`
		
		Examples:
		
		"Match Output of compare Command and Nonlinear ARX Model Block" on page 20-19
		
		"Apply Initial Conditions When Simulating Identified Linear Models" on page 20-45

| When to Use	Approach	Method Summary

You are simulating model response to a given input signal, and want to start the simulation at a point that leads to the closest match to a given output signal.	Use `compare` to return ICs for `idss`, `idtf`, `idproc`, `idpoly`, or `idgrey` models.
	
	Use `findstates` to return the initial state vector for `idss` models only.
	
	If the original model is `idtf`, `idproc`, or `idpoly`, you can use `findstates` by first converting the model to state-space form using `idss`. See "Estimate Initial Conditions for Simulating Linear Models" on page 20-30.
	
	See "Estimate Initial Conditions from Measured Data" on page 20-29.
		`[y,fit,x0] = compare(sys,z)`
		
		`[y,fit,ic] = compare(sys,z)`
		
		`x0 = findstates(sys_ss,z)`
		
		Examples:
		
		"Apply Initial Conditions When Simulating Identified Linear Models" on page 20-45
		
		"Estimate Initial Conditions for Linear Model Using Measured Data" on page 20-30

You are continuing a simulation, such as for online processing. You are using a split data set, such as for `nlarx` model when `findstates` is too computationally expensive.	`data2state`
	If original model is `idtf`, `idproc`, or `idpoly`, convert to state-space form first using `idss`.
	You can also use `data2state` implicitly by specifying your past data as a proxy for initial conditions when using `sim` or `predict` with any dynamic model.
	For more information, see "Estimate Initial Conditions from Measured Data" on page 20-29.
		`sys_ss = idss(sys)`
		`x0 = data2state(sys,z_p)`
		or
		`IO = struct ('Input',zp.InputData, 'Output',zp.OutputData);`
		`opt = simOptions ('InitialCondition',IO)`
		or
		`opt = predictOptions ('InitialCondition',IO).`
		examples: "Estimate Initial Conditions for a Nonlinear ARX Model" on page 20-40, "Understand Use of Historical Data for Model Simulation" on page 20-31

You are working with a continuing simulation, and want to base initial conditions for the current simulation segment on the previous segment.
You intend to start your main simulation in steady state, and you want to have a prequel which simulates the run-up to steady state from rest.
	Use state information from previous simulation or presimulation.
	If original model is `idtf`, `idproc`, or `idpoly`, convert to state-space form first using `idss`.
	For more information, see "Use Initial Conditions from Past State Information" on page 20-30.
		`sys_ss = idss(sys)`
		`x0 = x(end)` from
		`[y,y_sd,x] = sim(sys_ss,z_p)`
		example: "Continue Simulation of an Identified Model using Past States" on page 20-33

Initial-Condition Estimation Techniques

The preceding table calls out various techniques for initial-condition estimation. The following sections summarize these techniques by the estimation data source.

Estimate Initial Conditions from Measured Data

Estimate the initial conditions from the measurement data from which you are getting your simulation inputs. Techniques include the following.

- Extract initial conditions from model-identification results, such as from the identification report for your model or as returned values from `compare`. X_0 represents an initial state vector and *IC* represents an `initialCondition` object. For examples, see "Resolve Fit Value Differences Between Model Identification and compare Command" on page 20-22, "Match Output of compare Command and Nonlinear ARX Model Block" on page 20-19, and "Apply Initial Conditions When Simulating Identified Linear Models" on page 20-45.

- Estimate initial conditions using measured input/output data, using the function `findstates` to estimate the initial states consistent with the data set. For an example, see "Estimate Initial Conditions for Linear Model Using Measured Data" on page 20-30.
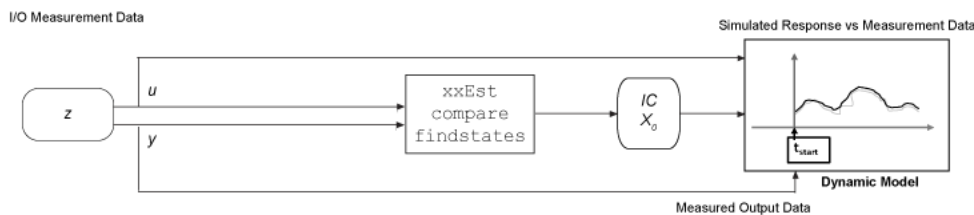
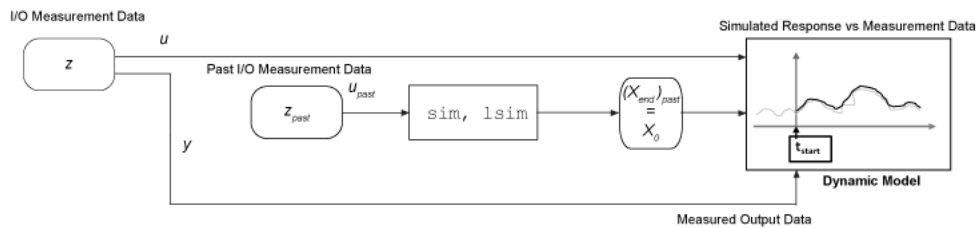### Estimate Initial Conditions from Past Measurement Data

Estimate the initial conditions from the measurement data immediately preceding your simulation start. Techniques include the following.

- Using the function `data2state` to estimate that data set end state. The past end state then becomes the initial state for follow-on simulation. For an example, see "Estimate Initial Conditions for a Nonlinear ARX Model" on page 20-40.

- Using your past data as a proxy for initial conditions when using `sim` or `predict`. When you run your simulation, the model internally calls `data2state` for you. For an example, see "Understand Use of Historical Data for Model Simulation" on page 20-31.

- Splitting the data set you are using into two parts. Make the first part just large enough to allow `data2state` to calculate an accurate end state. This split-data approach can be much less computationally intensive than using `findstates` on the full data set, especially for nonlinear ARX models. For an example, see "Estimate Initial Conditions for a Nonlinear ARX Model" on page 20-40.
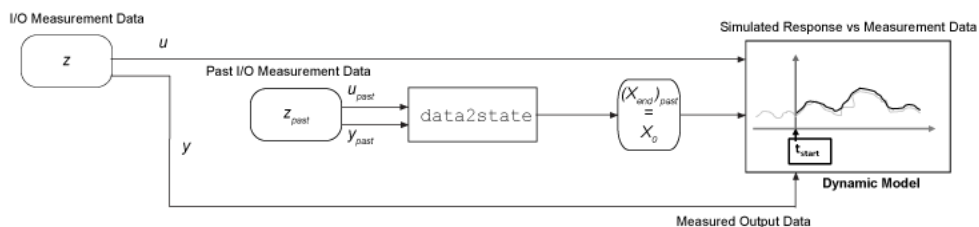
Use Initial Conditions from Past State Information

Use state information from previous simulations or presimulations. If no previous simulation is available, you can presimulate from rest or a known initial condition up to the point of your simulation start. The end states of the past or presimulation are the starting states for the simulation you want to perform. For an example, see "Continue Simulation of an Identified Model using Past States" on page 20-33.

Estimate Initial Conditions for Simulating Linear Models

The `findstates` and `data2state` functions are the most general tools for determining initial or end states from any measurements of input/output data. However, to use these functions, you must have a model structure that includes explicit state representation. Linear models other than `idss` and `idgrey` do not use explicit states. To find the equivalent initial states, you can convert these models to `idss` models and find the initial states from the `idss` versions. As an alternative to `findstates`, you can also use `compare` to obtain initial conditions without converting your models into state-space form. For these models, `compare` returns an `initialCondition` object that contains information on the model free response, in state-space form, and the corresponding initial states.

After you estimate the initial conditions, you can incorporate them into your command-line simulation syntax, your Simulink model, or the **System Identification** app.

The following examples illustrate initial-condition estimation for linear models. The first example uses `findstates` and `compare`. The second and third examples use historical data. That is, the examples determine initial conditions from the data immediately preceding the simulation start time.

Estimate Initial Conditions for Linear Model Using Measured Data

Use different methods to obtain initial conditions for a linear model and compare the results.

Estimate a transfer function for measured data. Return the initial conditions in `ic`.

```
load iddata1 z1;
[sys,ic] = tfest(z1,3);
ic
```

```
ic =
  initialCondition with properties:

    A: [3x3 double]
   X0: [3x1 double]
    C: [0.2303 5.9117 2.2283]
   Ts: 0
```

```
ic.X0
```

```
ans = 3×1

   -1.7569
    2.6195
   -6.5177
```

sys is a continuous-time identified transfer function (idtf) model. ic is an initialCondition object. An initialCondition object represents the free response of the system in state-space form. The object includes the converted A and C state-space matrices that correspond to sys and the estimated initial state vector X0.

You can also use findstates to find initial conditions for linear models, but only if they are in idss or idgrey form.

Convert sys to state-space form.

```
sys_ss = idss(sys);
```

Use findstates to estimate the initial states.

```
X0 = findstates(sys_ss,z1)
```

```
X0 = 3×1

   -1.7569
    2.6195
   -6.5177
```

Now use compare to estimate the initial states, using the original transfer function model.

```
[y_tf,fit,ic2] = compare(z1,sys);
ic2
```

```
ic2 =
  initialCondition with properties:

    A: [3x3 double]
   X0: [3x1 double]
    C: [0.2303 5.9117 2.2283]
   Ts: 0
```

```
ic2.X0
```

```
ans = 3×1

   -1.7569
    2.6195
   -6.5177
```

The initial state vectors are identical in all three cases.

Understand Use of Historical Data for Model Simulation

Use historical input-output data as a proxy for initial conditions when simulating your model. You first simulate using the `sim` command and specify the historical data using the `simOptions` option set. You then reproduce the simulated output by manually mapping the historical data to initial states.

Load a two-input, one-output data set.

```
load iddata7 z7
```

Identify a fifth-order state-space model using the data.

```
sys = n4sid(z7,5);
```

Split the data set into two parts.

```
zA = z7(1:15);
zB = z7(16:end);
```

Simulate the model using the input signal in `zB`.

```
uSim = zB;
```

Simulation requires initial conditions. The signal values in `zA` are the historical data, that is, they are the input and output values for the time immediately preceding data in `zB`. Use `zA` as a proxy for the required initial conditions.

```
IO = struct('Input',zA.InputData,'Output',zA.OutputData);
opt = simOptions('InitialCondition',IO);
```

Simulate the model.

```
ysim = sim(sys,uSim,opt);
```

Now reproduce the output by manually mapping the historical data to initial states of `sys`. To do so, use the `data2state` command.

```
xf = data2state(sys,zA);
```
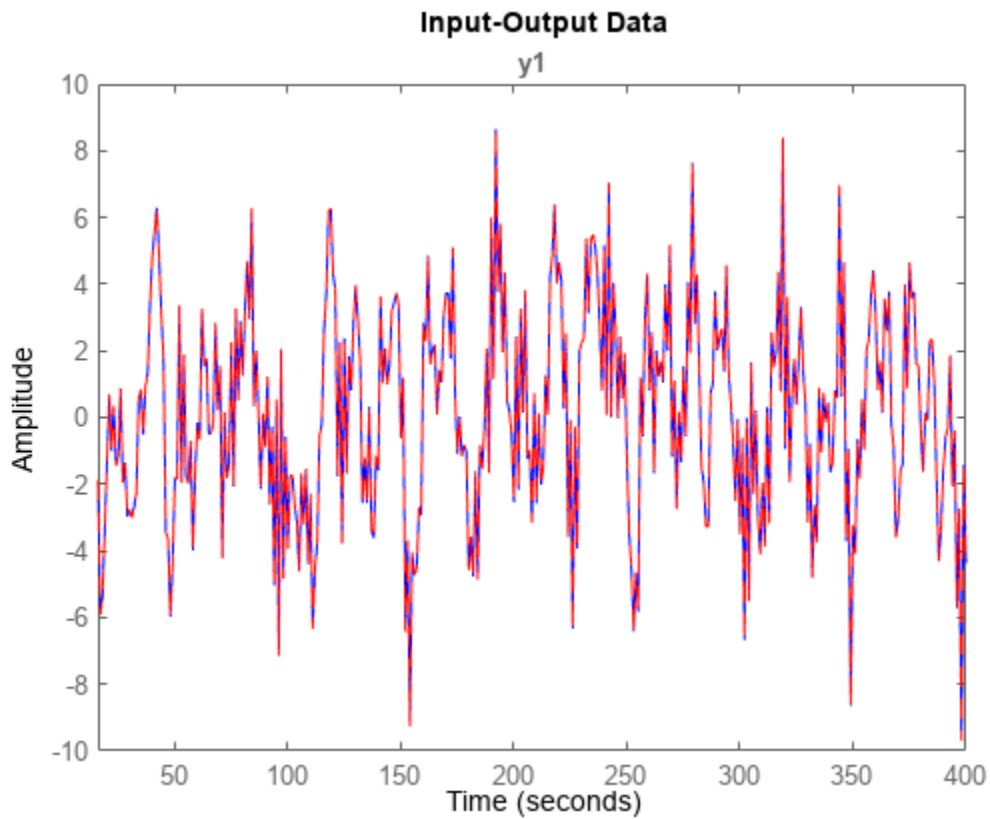
`xf` contains the state values of `sys` at the time instant immediately after the most recent data sample in `zA`.

Simulate the system using `xf` as the initial states.

```
opt2 = simOptions('InitialCondition',xf);
ysim2 = sim(sys,uSim,opt2);
```

Plot the output of the `sim` command `ysim` and the manually computed results `ysim2`.

```
plot(ysim,'b',ysim2,'--r')
```

Input-Output Data

`ysim2` is the same as `ysim`.

Continue Simulation of an Identified Model using Past States

Estimate the initial conditions for a simulation segment of an identified model by using the end states of the previous simulation segment.

Load data z1, which is an `iddata` object containing input and output data. Estimate a fifth-order linear model from the data.

```
load iddata1 z1
plot(z1)
```

Input-Output Data
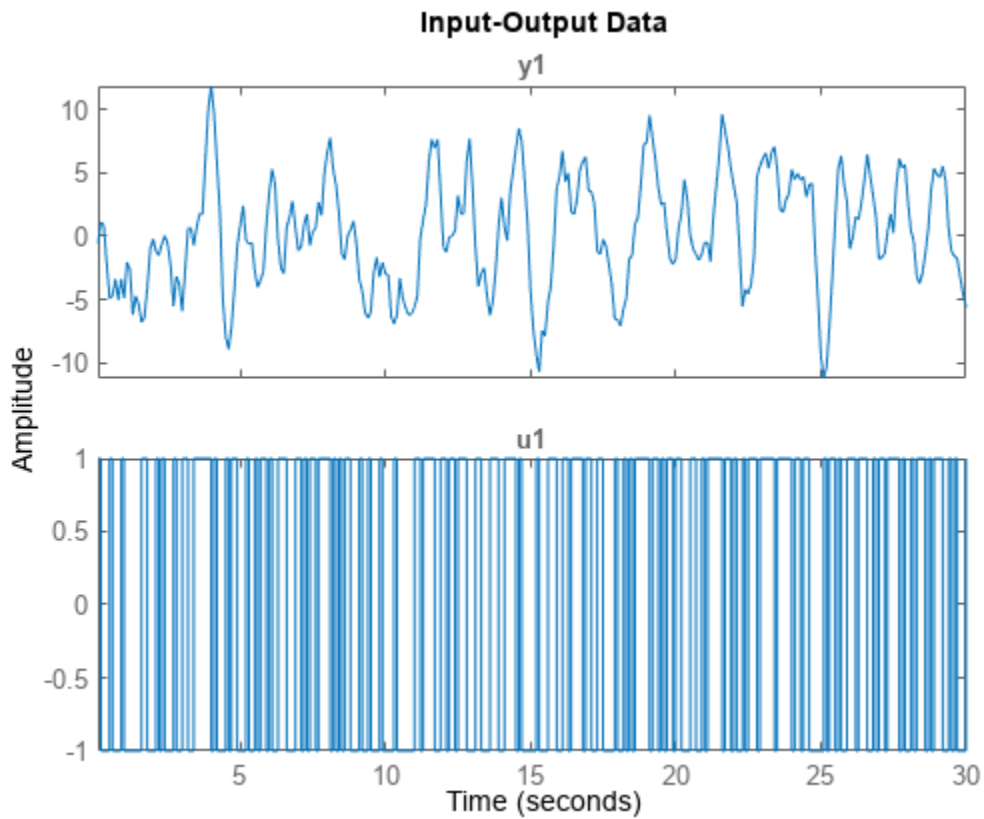

```
sys = n4sid(z1,5);
```

Divide the data into two segments. One segment `z1past` represents data for a past simulation, and starts at rest. The second segment `z1next` represents data for a follow-on simulation.

```
z1past = z1(1:150);
z1next = z1(150:300);
```

Simulate the response to the input data in `z1next` without specifying initial conditions.

```
ynext_no_ic = sim(z1next,sys);
plot(ynext_no_ic,z1next)
legend
title('Simulation Continuation Without Setting ICs')
```

Simulation Continuation Without Setting ICs
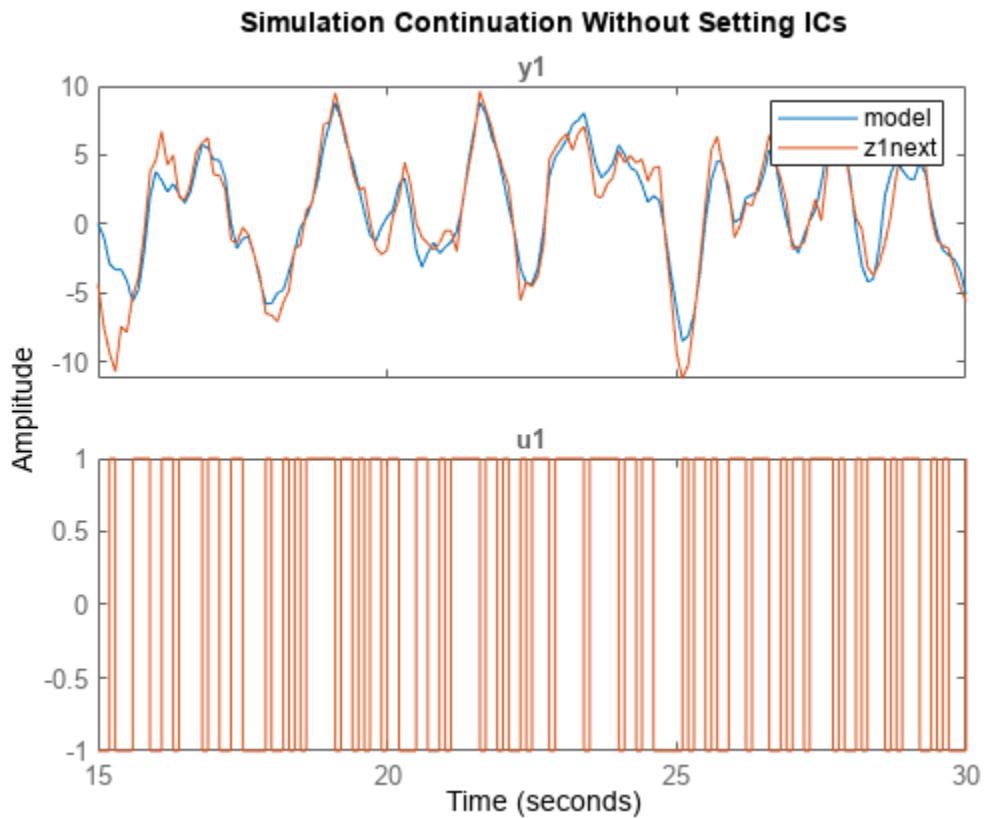
Now determine the end states, x0, for the response to `z1past` input.

```
[ypast,y_sd,xpast] = sim(z1past,sys);
x0 = xpast(end,:)';
```

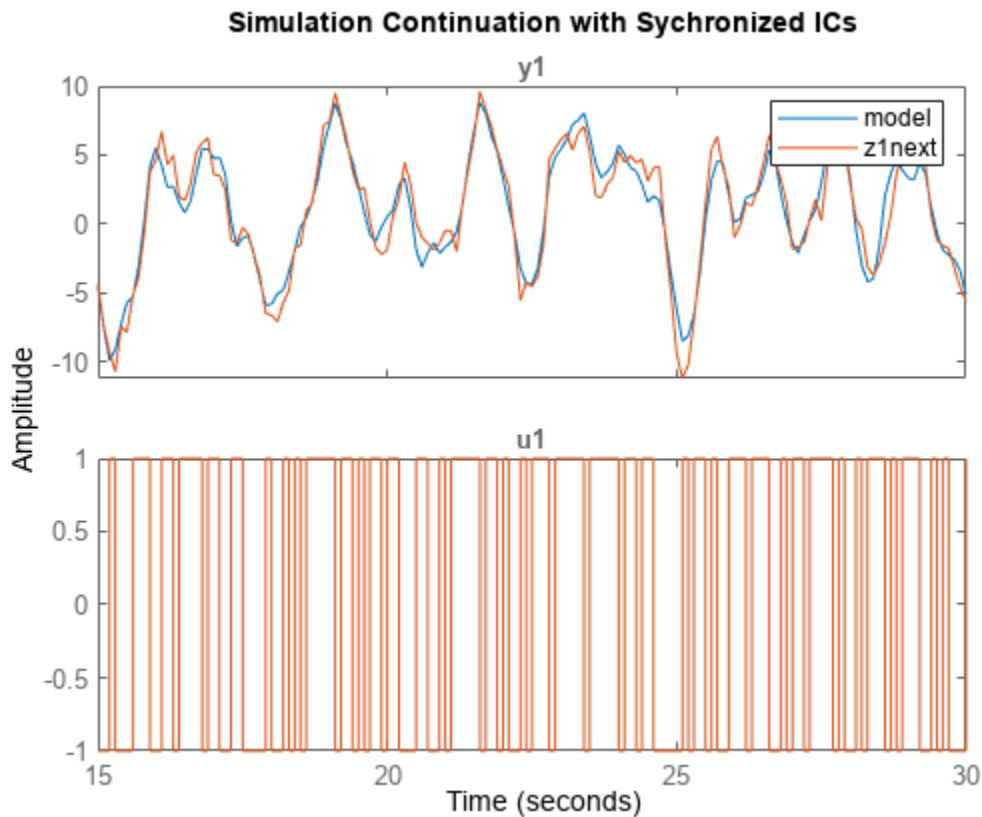Use the end states to specify the initial states for the follow-on response to `z1next`.

```
opt = simOptions('InitialCondition',x0);
ynext_ic = sim(z1next,sys,opt);
figure
plot(ynext_ic,z1next)
legend
title('Simulation Continuation with Sychronized ICs')
```

20-35

Estimate Initial Conditions for Simulating Hammerstein-Wiener and Nonlinear Grey-Box Models

When you are simulating a Hammerstein-Wiener or a nonlinear grey-box model and want to use initial conditions consistent with a measurement data set z, use `findstates`.

Simulate Hammerstein-Wiener Model in Simulink

Compare the simulated output of a Hammerstein-Wiener Model block to the measured output of a system. You improve the agreement between the measured and simulated responses by estimating initial state values.

Load the sample data.

```
load twotankdata
```

Create an `iddata` object from the sample data. Set `'Tstart'` to 0 so that the data start time matches the Simulink start time 0f 0 s.

```
z1 = iddata(y,u,0.2,'Tstart',0,'Name','Two tank system');
```

Estimate a Hammerstein-Wiener model using the data.

```
mw1 = nlhw(z1,[1 5 3],idPiecewiseLinear,idPiecewiseLinear);
```

You can now simulate the output of the estimated model in Simulink using the input data in z1. To do so, open a preconfigured Simulink model.

```
model = 'ex_idnlhw_block';
open_system(model);
```


Copyright 2000-2021 The MathWorks Inc.

The model uses the Iddata Source, Hammerstein-Wiener Model, and Scope blocks. The following block parameters have been preconfigured to specify the estimation data, estimated model, and initial conditions:

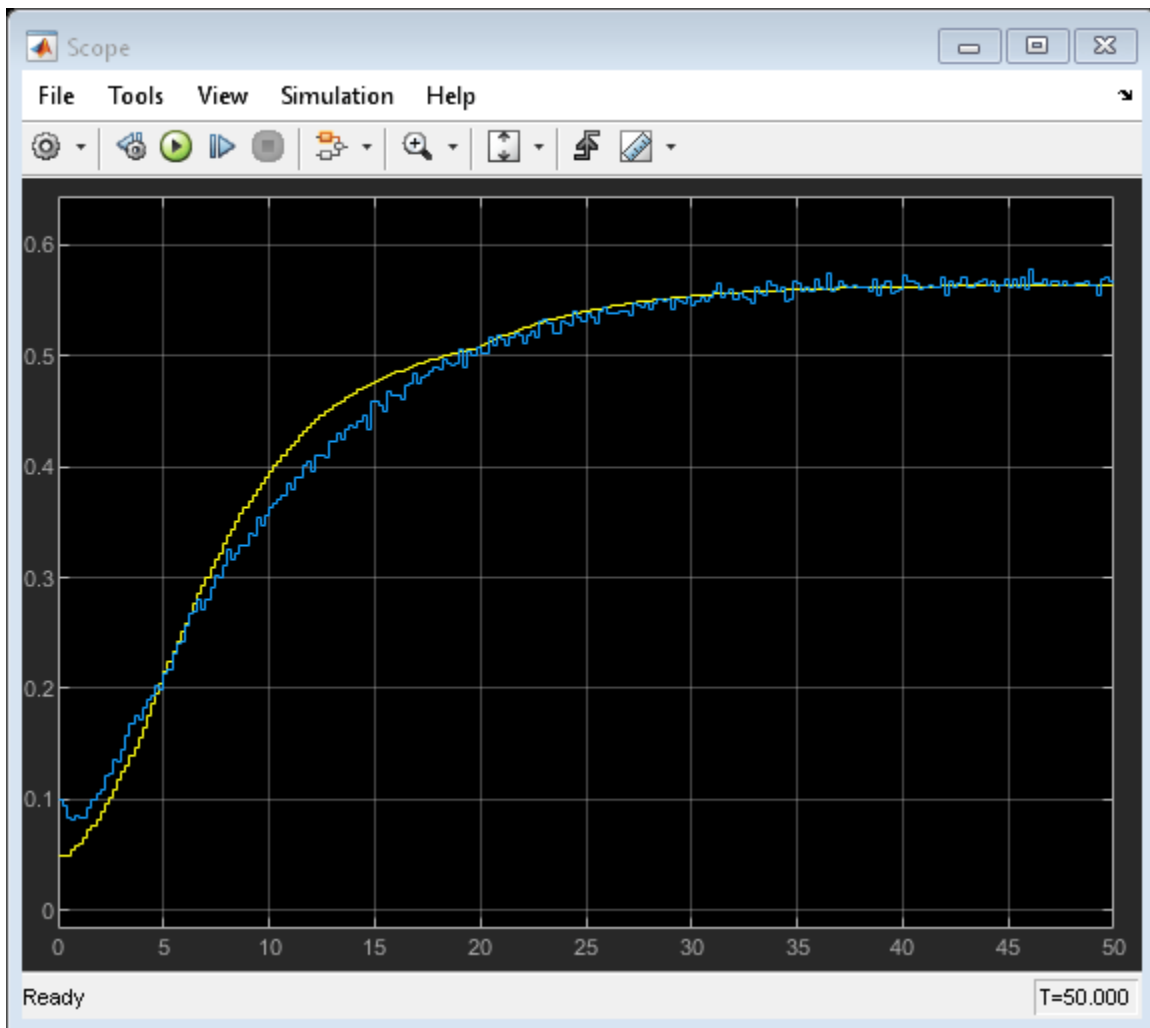**Block parameters of Iddata Source block:**

* **IDDATA Object** - z1

Block parameters of Hammerstein-Wiener Model block:

* **Model** - mw1
* **Initial conditions** - Zero (default)

Run the simulation.

View the difference between measured output and model output by using the Scope block.

```
simOut = sim(model);
open_system([model '/Scope'])
```

To improve the agreement between the measured and simulated responses, estimate an initial state vector for the model from the estimation data `z1`, using `findstates`. Specify the maximum number of iterations for estimation as 100. Specify the prediction horizon as `Inf`, so that the algorithm computes the initial states that minimize the simulation error.

```
opt = findstatesOptions;
opt.SearchOptions.MaxIterations = 100;
x0 = findstates(mw1,z1,Inf,opt);
```

Set the **Initial conditions** block parameter value of the Hammerstein-Wiener Model block to `State Values`. The default initial states are `x0`.

```
set_param([model '/Hammerstein-Wiener Model'],'IC','State values');
```

Run the simulation again, and view the difference between measured output and model output in the Scope block. The difference between the measured and simulated responses is now reduced.

```
simOut = sim(model);
```

Estimate Initial Conditions for Simulating Nonlinear ARX Models

When you are simulating a nonlinear ARX model and want to use initial conditions consistent with a measurement data set z, you have a choice for which approach to take.

- You can use findstates. For nonlinear ARX models however, this approach has the cost of being computationally expensive if the data set is large.

```
X0 = findstates(m,z,Inf);
```

- You can break the data set into two portions. Use the first nx samples of the input/output data as past data, where nx is the largest regressor delay in the model. Then use data2state to compute the end state of the "past data." You are then able to simulate the model for the remaining nx +1:end input data with whatever simulation approach you choose. Both compare and the **System Identification** app use this approach automatically for nonlinear ARX models.

```
nx = max(getDelayInfo(m));  % Find the largest regressor delay
past_data = z1(1:nx);
X0 = data2state(mw1,z1(1:nx));
```

The following example illustrates the use of split-data technique for a nonlinear ARX model.

Estimate Initial Conditions for a Nonlinear ARX Model

Use two different techniques to estimate the initial conditions for a nonlinear ARX model. Simulate each alongside the measurement data, and compare with a simulation that initializes at rest.

Load the input/output data z and plot it. Use the first 1000 points of z to estimate a nonlinear ARX model.

```
load twotankdata
z = iddata(y,u,0.2);
z1 = z(1:1000);
plot(z1)
title('Estimation data')
```
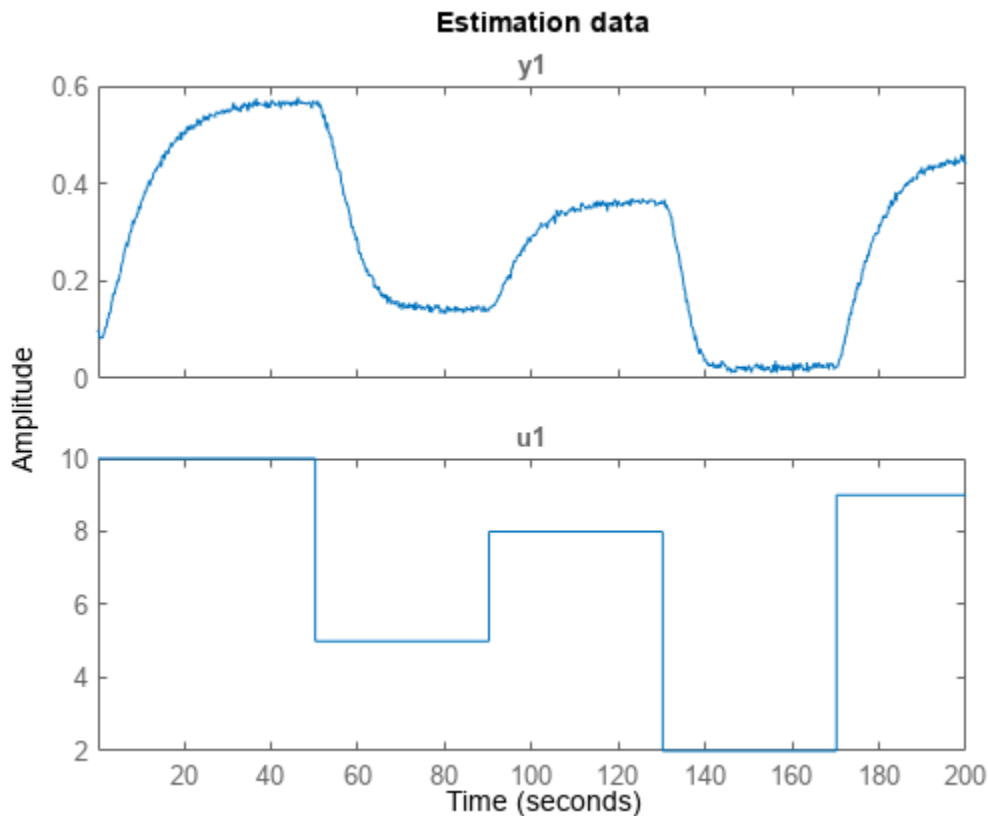

mdlnlarx = nlarx(z1,[5 1 3],idWaveletNetwork);

Extract the first 250 points of z1 (50 seconds) as `z1sim` for comparison simulations. Simulate `mdlarx` without setting initial conditions, and plot the response alongside the measured output in `z1sim`.

```
z1sim = z1(1:250);
data_no_ic = sim(mdlnlarx,z1sim);
plot(data_no_ic,z1sim)
title('Nonlinear ARX Model with No Initial Condition Setting')
grid on
legend('location','se')
```

Nonlinear ARX Model with No Initial Condition Setting
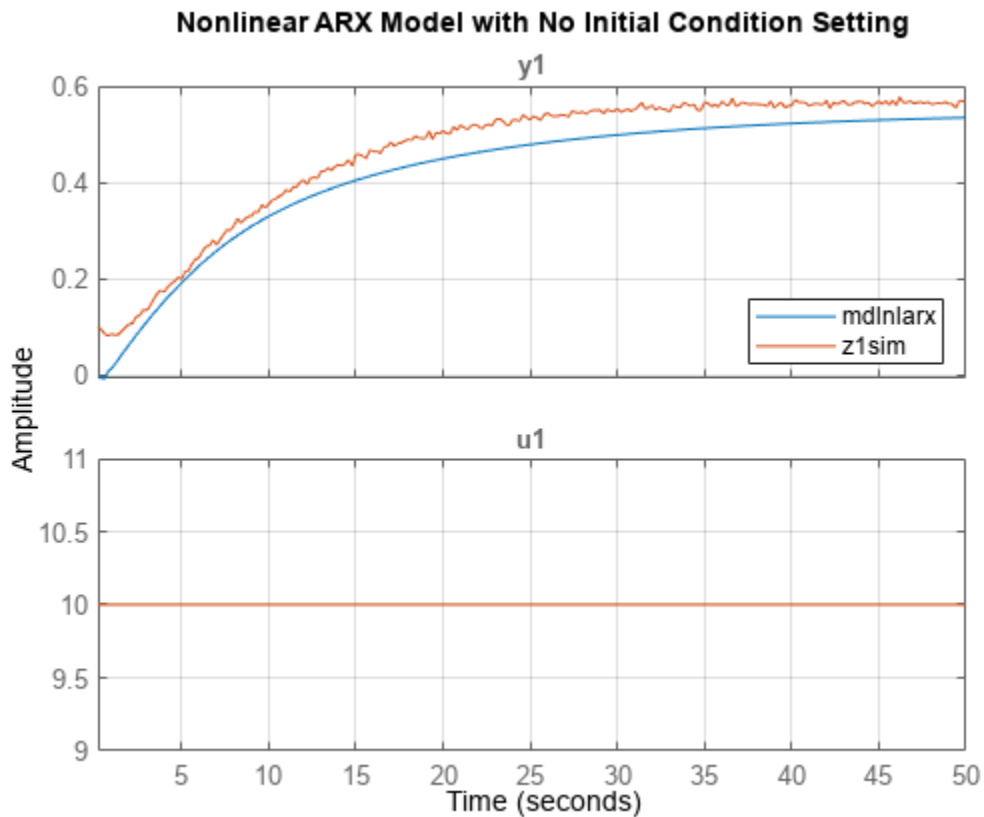
Use `findstates` to estimate the initial conditions. In order to minimize the simulation error, specify `Inf` for the prediction horizon. Then set the `sim` option for `InitialCondition` to the `findstates` results. Plot the response alongside the measured output.

```
x0fs = findstates(mdlnlarx,z1sim,Inf);
opt = simOptions('InitialCondition',x0fs);
data_fs = sim(mdlnlarx,z1sim,opt);
figure
plot(data_fs,z1sim)
title('Nonlinear ARX Model with IC Estimation Using findstates')
grid on
legend('location','se')
```
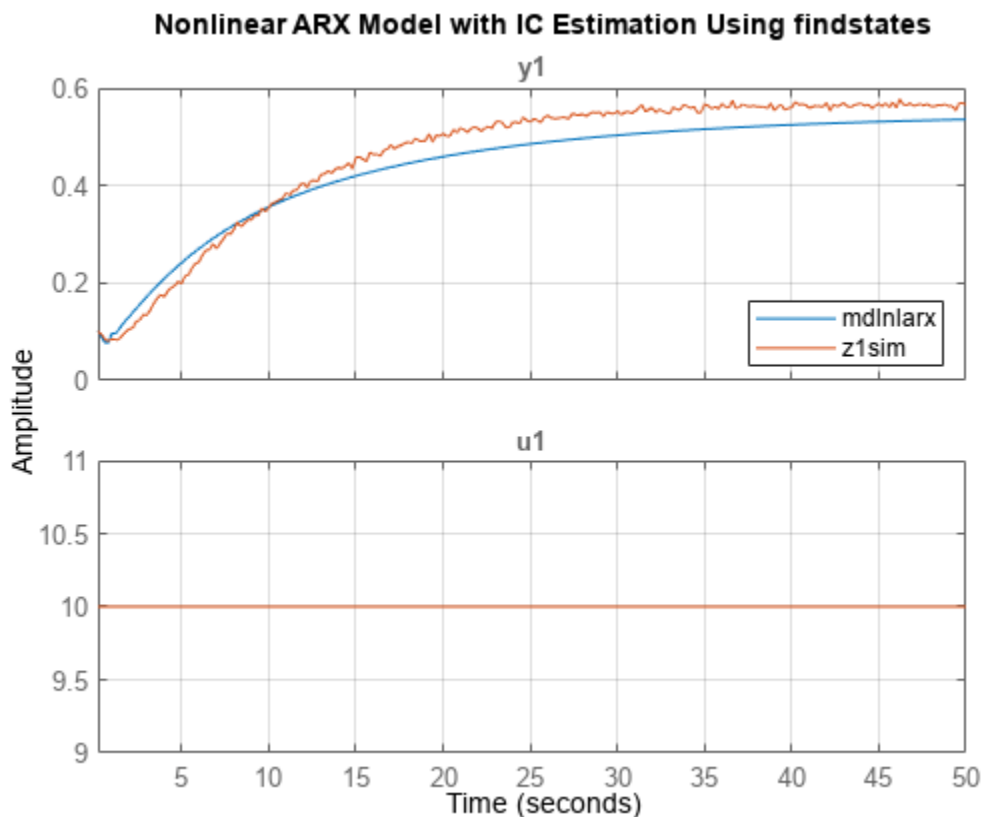
The response and the measured output now start at roughly the same point.

Now use the `data2states` method to estimate the initial conditions. First, break the data set into two portions. Use the first `nx` samples of `z1sim` as "past data", where `nx` is the largest regressor delay in the model.

```
nx = max(getDelayInfo(mdlnlarx))
```

```
nx = 5
```

```
z1past = z1sim(1:nx);
z1sim2 = z1sim(nx+1:end);
```

Use `data2state` to compute the end state of the "past data." Simulate the response using input starting at `z1sim(nx+1)`. Plot the response alongside the full `z1sim` measurement data so that you can compare with the `findstates` plot.

```
x0d2s = data2state(mdlnlarx,z1past);
opt = simOptions('InitialCondition',x0d2s);
data_d2s = sim(mdlnlarx,z1sim2,opt);
figure
plot(data_d2s,z1sim)
title('Nonlinear ARX Model with IC Estimation Using data2state')
grid on
legend('location','se')
```
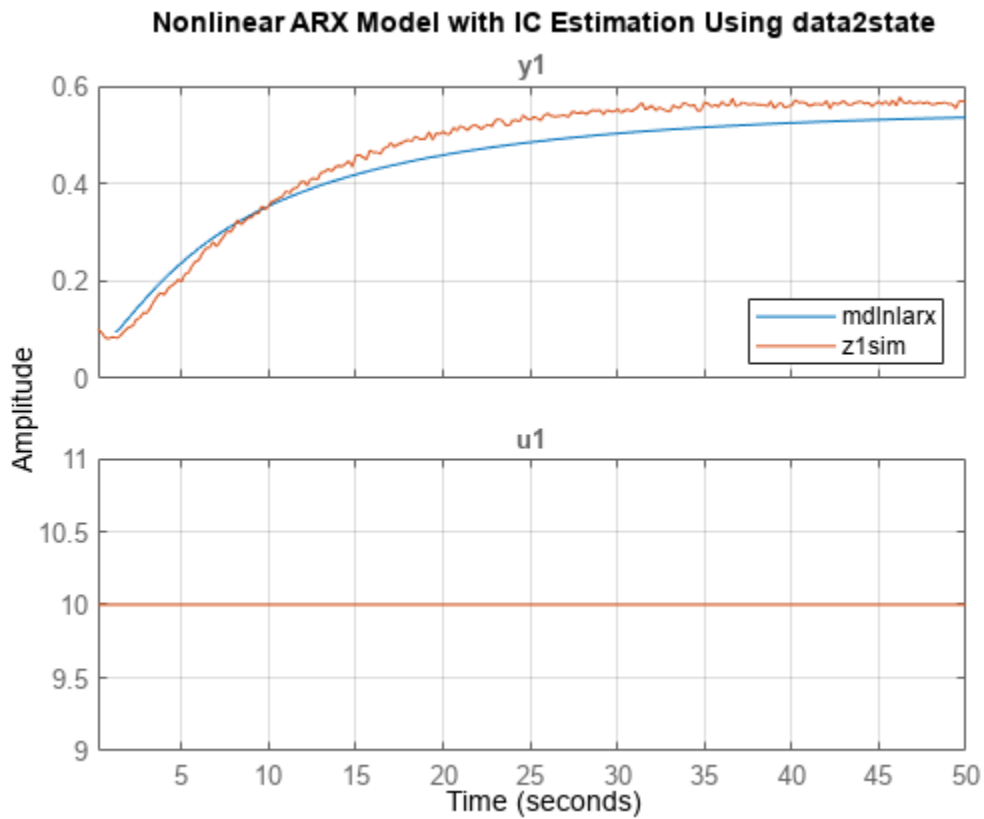
Nonlinear ARX Model with IC Estimation Using data2state

If you executed each section separately, you may find that the `data2states` method completed more quickly than the `findstates` method.

Now compare the responses for all three initial-condition cases.

```
plot(data_fs,data_d2s,data_no_ic)
title('Nonlinear ARX model Responses for Three IC Approaches Compared')
legend('location','se')
```
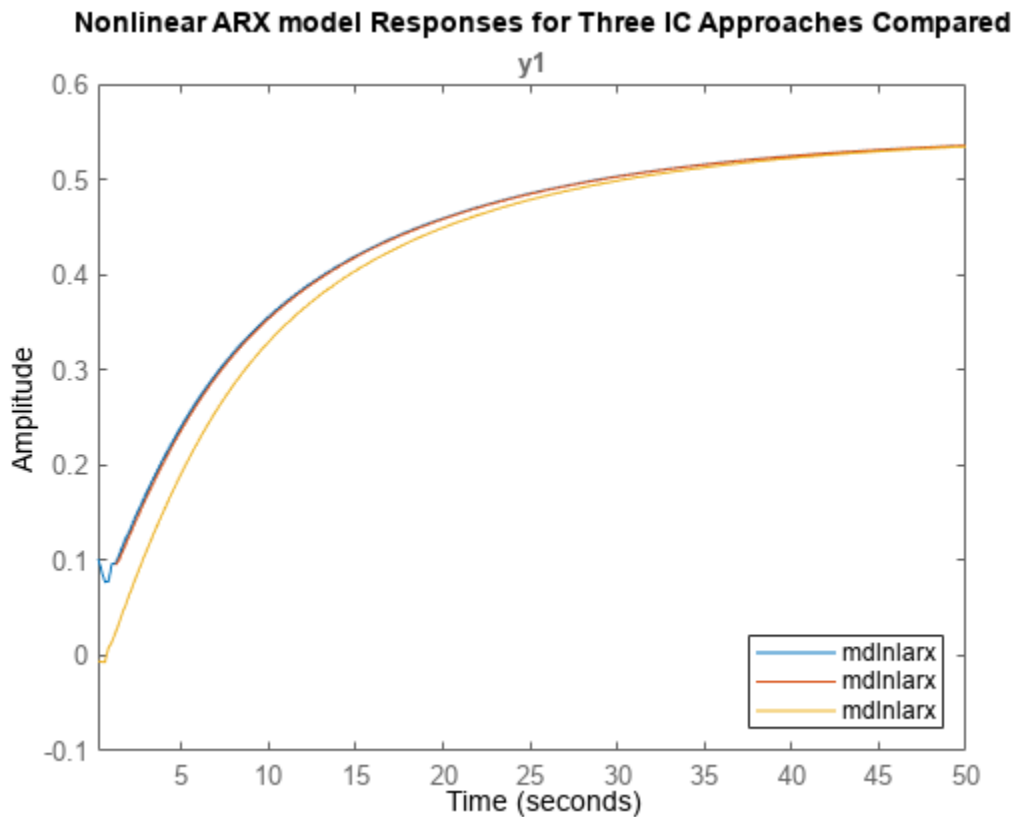
The responses for cases using `findstates` and `data2state` are virtually the same. The response for the case where the initial conditions were not set converges eventually, but not until after 30 seconds.

See Also

compare | `compareOptions` | `sim` | `simOptions` | `predict` | `predictOptions` | `findstates` | `data2state`

More About

* "Simulate Identified Model in Simulink" on page 20-5
* "Reproduce Command Line or System Identification App Simulation Results in Simulink" on page 20-15

Apply Initial Conditions When Simulating Identified Linear Models

This example shows the workflow for obtaining and using estimated initial conditions when you simulate models to validate model performance against measured data.

Use initial conditions (ICs) when you want to simulate a model in order to validate model performance by comparing the simulated response to measured data. If your measurement data corresponds to a system that does not start at rest, a simulation that assumes a resting start point results in a mismatch. The simulated and measured responses do not agree at the beginning of the simulation.

For state-space models, the initial state vector is sufficient to describe initial conditions. For other LTI models, the `initialCondition` object allows you to represent ICs in the form of the free response of your model to the initial conditions. This representation is in state-space form, as shown in the following equation:

$$G(s) = C(\text{s}I - A)^{-1}\text{X0}$$

Here, $G(s)$ is the free response that the `initialCondition` encapsulates. A and C are the A and C matrices of the state-space form of your model. X0 is the corresponding initial state vector. The free response does not include B and D matrices because the IC free response is independent of input signals. Although the `initialCondition` object packages the state-space form, you can use the information to model a free response for any LTI system. The simulation software computes the free and forced responses separately and then adds them together to obtain the total response.

Prepare Data

Load the data and split it into estimation and validation data sets. For this example, the splits occur where the output data has a visibly nonzero start point.

```
load iddata2 z2
z2e = z2(10:210);
z2v = z2(210:400);
plot(z2e,z2v)
```

Input-Output Data
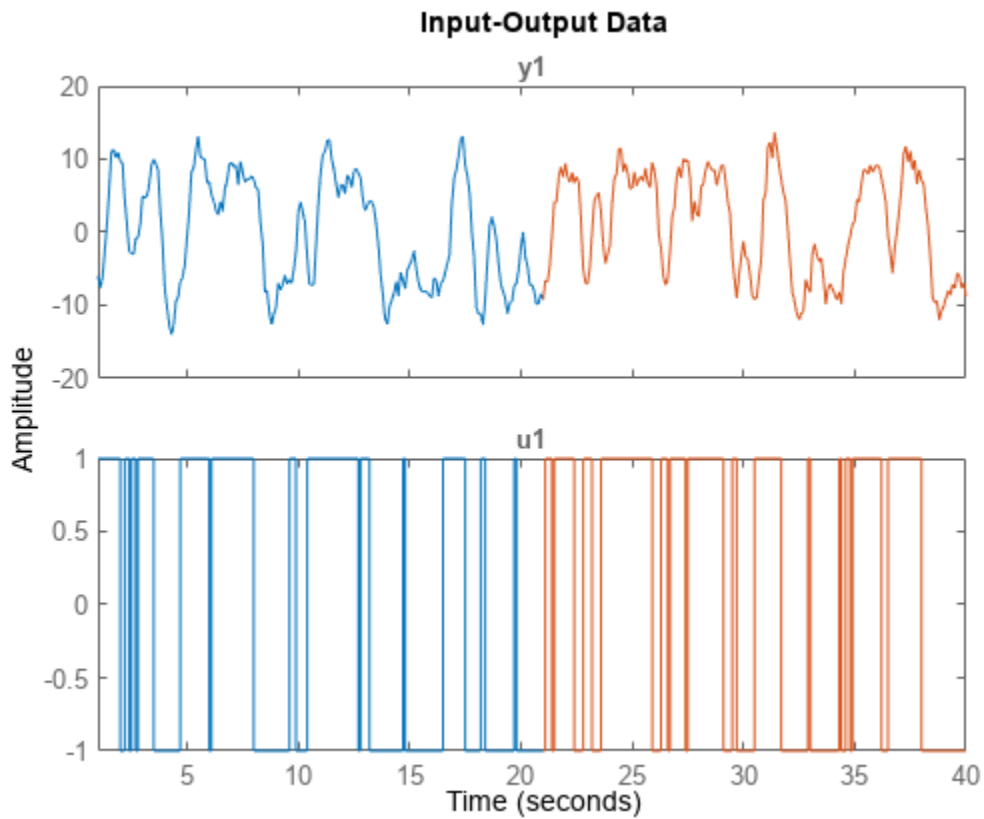
Display the first output sample for each data set.

```
z2e.y(1)
```

```
ans = -5.9588
```

```
z2v.y(1)
```

```
ans = -9.2141
```

Estimate Transfer Function Model

Using the estimation data, estimate a second-order transfer function model and return the initial condition ic. Display ic.

```
np = 2;
nz = 1;
[sys_tf,ic] = tfest(z2e,np,nz);
ic
```

```
ic =
  initialCondition with properties:

     A: [2x2 double]
    X0: [2x1 double]
     C: [-1.6158 5.1969]
    Ts: 0
```

`ic` represents the free response of the transfer function model, in state-space form, to the initial condition.

```
A = ic.A
```

A = 2×2

```
   -3.4145   -5.6635
    4.0000         0
```

```
C = ic.C
```

C = 1×2

```
   -1.6158    5.1969
```

The `X0` property contains the initial state vector.

```
X0 = ic.X0
```

X0 = 2×1

```
   -0.5053
   -1.2941
```

Simulate Model

Simulate the model. First, as a reference, simulate the model without incorporating `ic`. Plot the simulated response with the estimation data.

```
y_no_ic = sim(sys_tf,z2e);
plot(y_no_ic,z2e)
legend('Model Response','Output Data')
title('Transfer Function Model - No IC Specification')
```
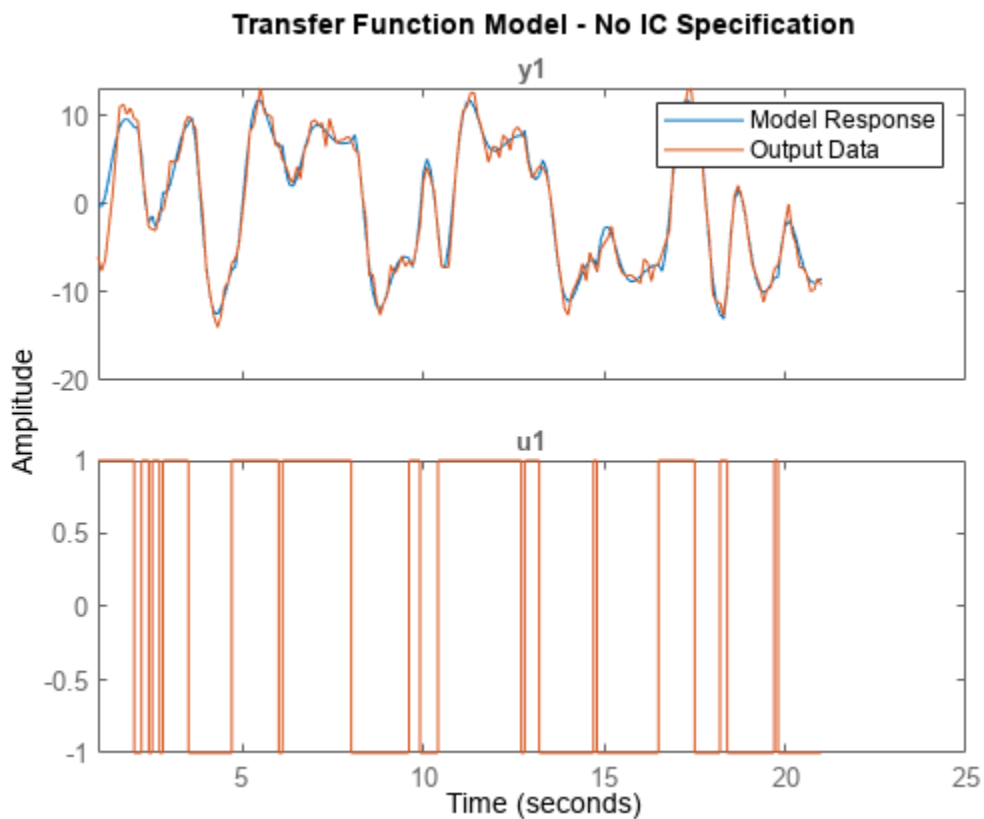
The simulated and measured responses do not agree at the start of the simulation.

Incorporate `ic`. To do so, first initialize `opt` to the option set `simOptions`. Specify `ic` as the `'InitialCondition'` setting. Simulate the model and plot the results.

```
opt = simOptions;
opt.InitialCondition = ic;
y_ic = sim(sys_tf,z2e,opt);
plot(y_ic,z2e)
legend('Model Response','Output Data')
title('Transfer Function Model - ICs Specified')
```
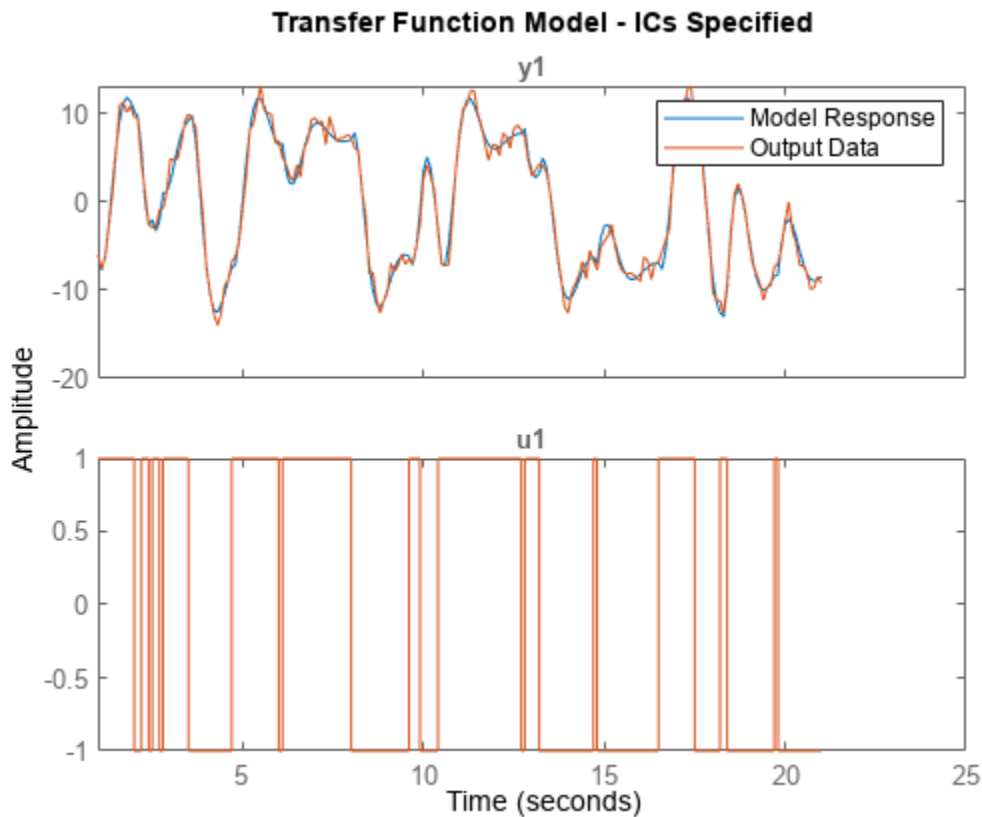
The responses now agree more closely at the start of the simulation.

Obtain ICs for Validation Data

`ic` represents the ICs only for the estimation data set. If you want to run a simulation using the validation inputs and compare the results with the validation output, you must obtain the ICs for the validation data set. To do so, use `compare`. You can use `compare` to estimate ICs for any combination of model and measurement data.

```
[yv,fitv,icv] = compare(z2v,sys_tf);
icv
```

```
icv =
  initialCondition with properties:

    A: [2x2 double]
   X0: [2x1 double]
    C: [-1.6158 5.1969]
   Ts: 0
```

Display the `A`, `C`, and `X0` properties of `icv`.

```
Av = icv.A
```

```
Av = 2×2

  -3.4145   -5.6635
```

```
    4.0000          0
```

```
Cv = icv.C
```

Cv = *1×2*

```
   -1.6158    5.1969
```

```
X0v = icv.X0
```

X0v = *2×1*

```
    0.4512
   -1.5068
```

For this case, the A and C matrices representing the free-response model are identical to the `ic.A` and `ic.C` values in the original estimation. However, the initial state vector `X0v` is different from `ic.X0`.

Specify `icv` as the `'InitialCondition'` setting in `opt` and simulate the model using the validation data. Plot the simulated and measured responses.

```
opt.InitialCondition = icv;
y_ic = sim(sys_tf,z2v,opt);
plot(y_ic(:,:,[]),z2v(:,:,[]))
legend('Model Response','Output Data')
title('Transfer Function Model Response For Validation Data')
```

Transfer Function Model Response For Validation Data
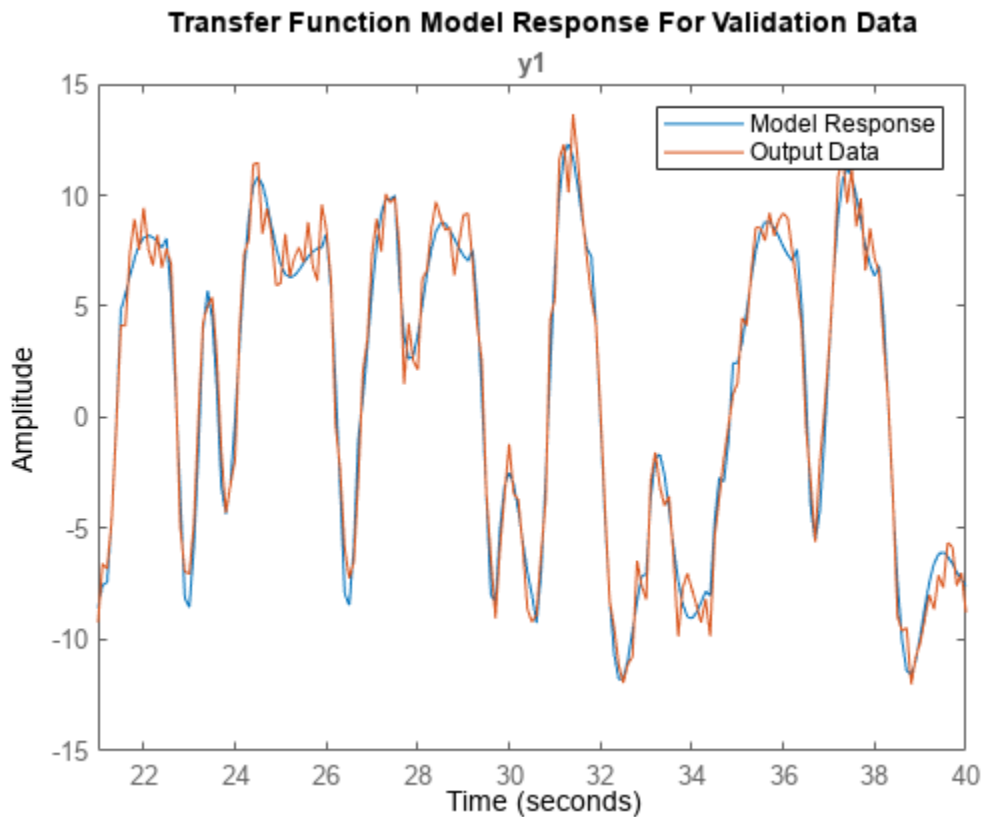
The simulated and measured responses have good agreement at the start of the simulation.

Apply ICs to Simulation of Converted Model

You can apply the ICs when you convert your model to another form.

Convert `sys_tf` to an `idpoly` model. Simulate the converted model, preserving the current SimOptions `'InitialCondition'` setting, `icv`, in `opt`.

```
sys_poly = idpoly(sys_tf);
y_poly = sim(sys_poly,z2v,opt);
plot(y_poly(:,:,[]),y_ic(:,:,[]),z2v(:,:,[]))
legend('idpoly Model Response','idtf Model Response','Output Data')
title('Converted Polynomial Model')
```

20-51
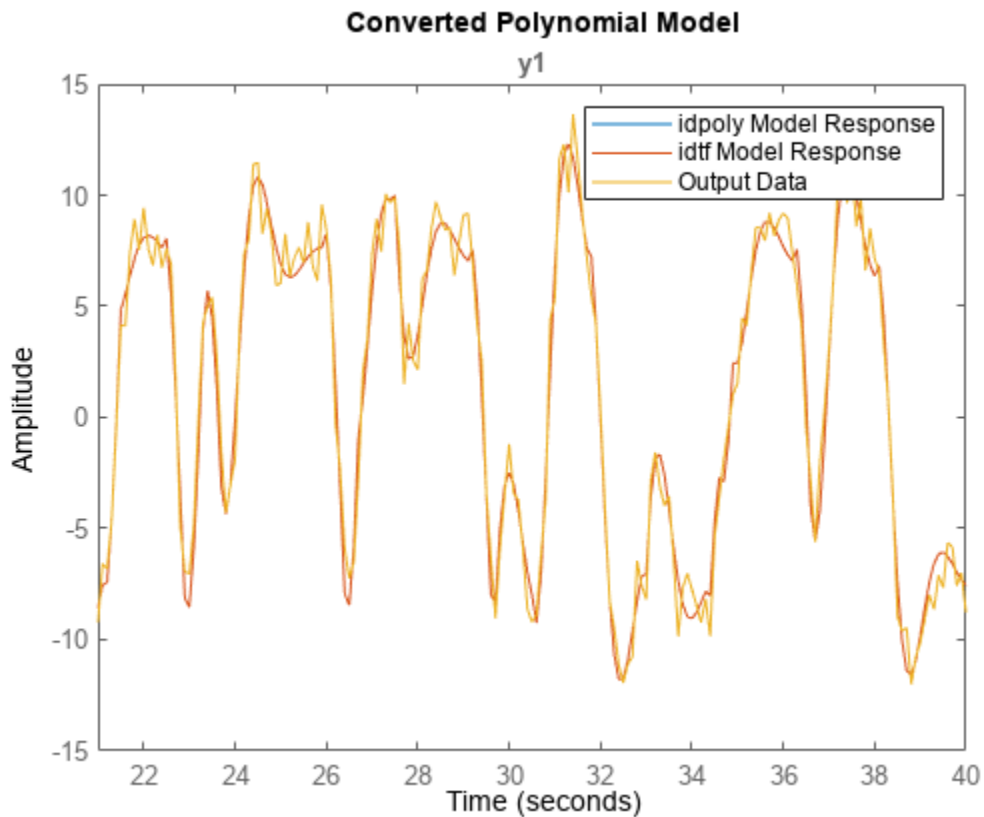
The `sys_tf` and `sys_poly` responses appear identical.

Apply ICs to Simulation of State-Space Model

State-space models can use initial conditions represented either by a single numeric initial state vector or by an `initialCondition` object.

When you convert `sys_tf` to an `idss` model, you can again use `icv` by retaining the `icv` setting in `opt`.

```
sys_ss = idss(sys_tf);
y_ss = sim(sys_ss,z2v,opt);
plot(y_ss(:,:,[]),z2v(:,:,[]))
legend('Model Response','Output Data')
title('Converted State-Space Model')
```
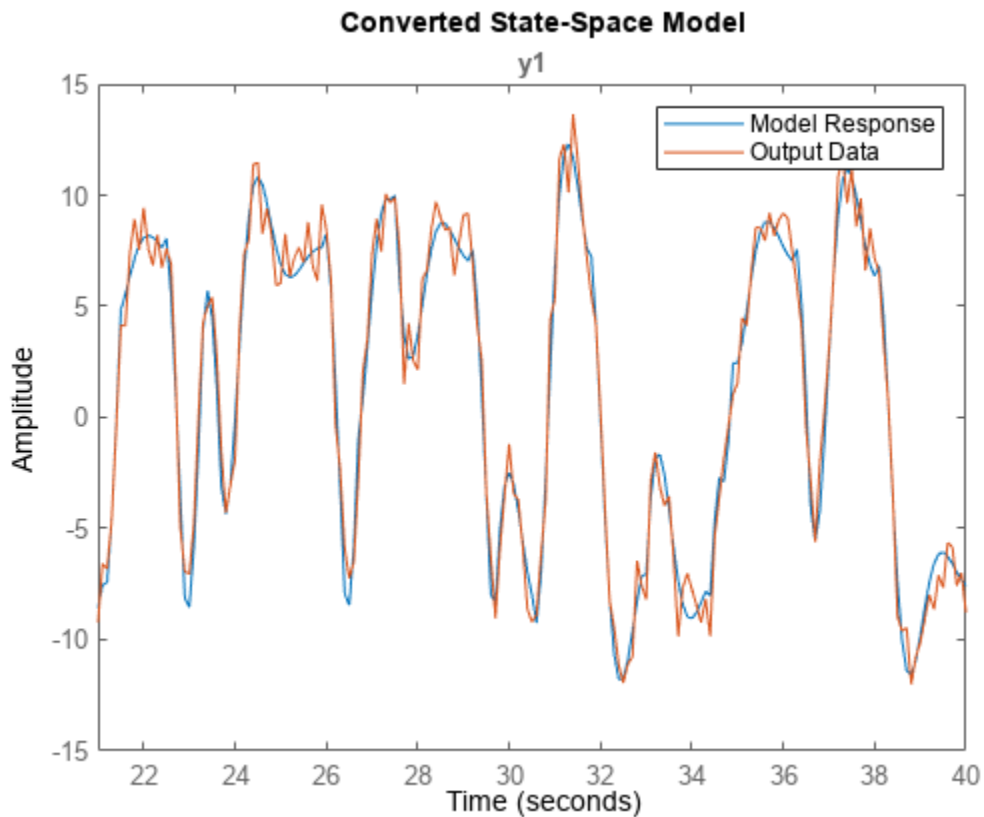
When you use `compare` or a state-space estimation function such as `ssest`, the function returns the initial state vector `x0`. Estimate the state-space model `sys_ss2` using `z2e` and use `compare` to obtain ICs corresponding to `z2v`.

```
sys_ss2 = ssest(z2e,2);
[yvss,fitvss,x0] = compare(z2v,sys_ss2);
x0
```

x0 = *2×1*

```
   -0.1061
    0.0097
```

`x0` is a numeric vector. Specify `x0` as the `'InitialCondition'` setting in `opt` and simulate the response.

```
opt.InitialCondition = x0;
y_ss2 = sim(sys_ss2,z2v,opt);
plot(y_ss2(:,:,[]),z2v(:,:,[]))
legend('Model Response','Output Data')
title('State-Space Model Simulated with Initial State Vector')
```

State-Space Model Simulated with Initial State Vector

Convert State-Space Initial State Vector to `initialCondition` Object

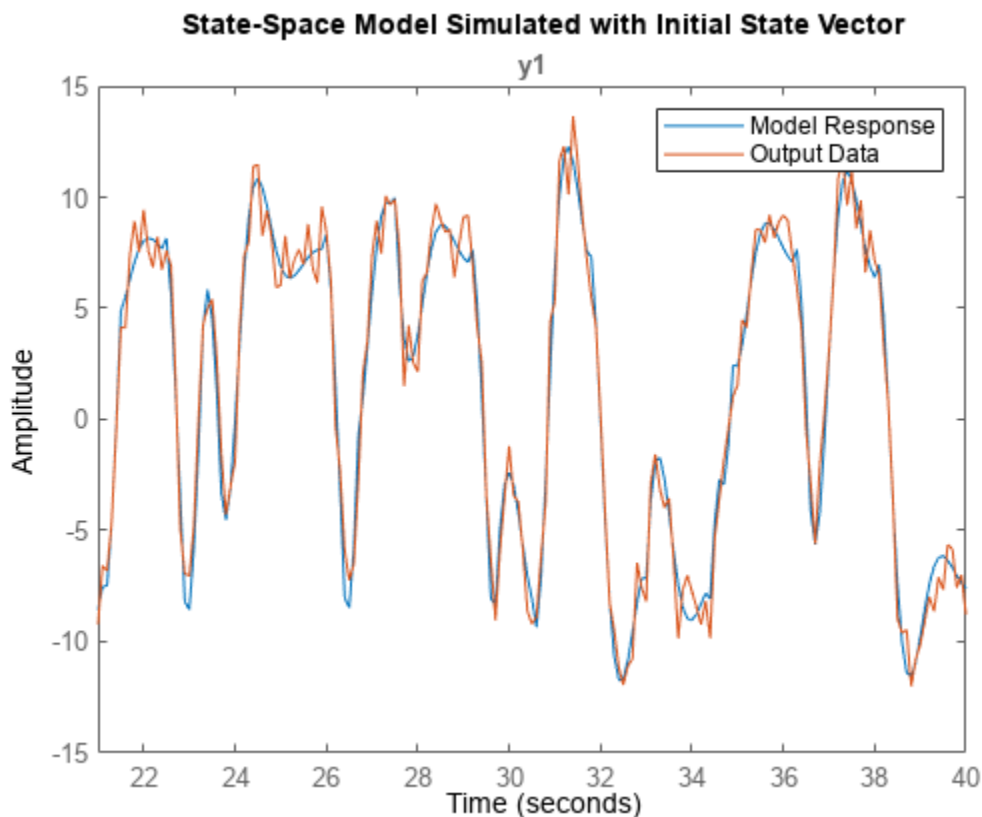If your original model is a state-space model and you want to convert the model into a polynomial or transfer function model and apply the same initial conditions, you must convert the initial state vector into an `initialCondition` object.

Extract and display the A, C, and Ts properties from `sys_ss2`.

```
As = sys_ss2.A
```

```
As = 2×2
```

```
    -1.7643    -3.7716
     5.2918    -1.7327
```

```
Cs = sys_ss2.C
```

```
Cs = 1×2
```

```
    82.9765    25.5146
```

```
Tss = sys_ss2.Ts
```

```
Tss = 0
```

The A and C matrices that are estimated using `ssest` have different values than the A and C matrices estimated in `icv` using `tfest`. There are infinitely many state-space representations of a given linear

model. The two pairs of matrices, along with the associated initial state vectors, are equivalent and produce the same free response.

Create the `initialCondition` object `ic_ss2` using `sys_ss2` model properties and the initial state vector `x0` you obtained when you used `compare`.

```
ic_ss2 = initialCondition(As,x0,Cs,Tss)

ic_ss2 =
  initialCondition with properties:

     A: [2x2 double]
    X0: [2x1 double]
     C: [82.9765 25.5146]
    Ts: 0
```

Convert `sys_ss2` into a transfer function model and simulate the converted model using `ic_ss2` as the `'InitialCondition'` setting.

```
sys_tf2 = idtf(sys_ss2);
opt.InitialCondition = ic_ss2;
y_tf2 = sim(sys_tf2,z2v,opt);
plot(y_tf2(:,:,[]),z2v(:,:,[]))
legend('Model Response','Output Data')
title('Transfer Function Model Converted from State-Space Model')
```

Transfer Function Model Converted from State-Space Model
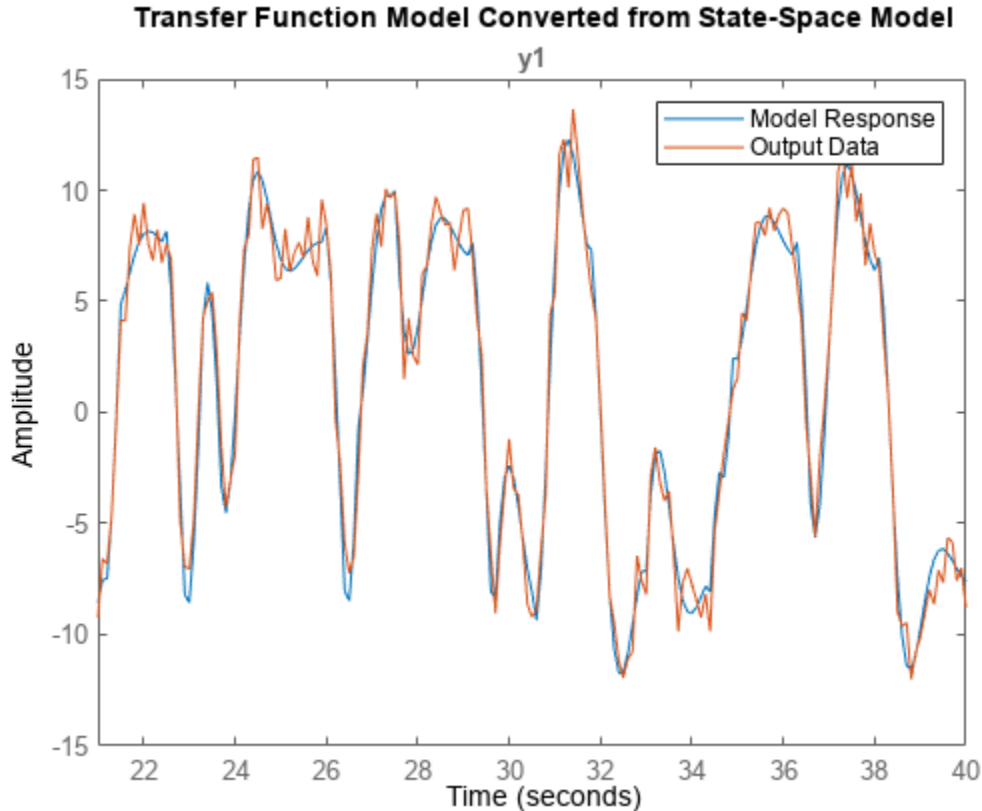
Using the constructed `initialCondition` object `ic_ss2` produces a similar response to simulated responses that use directly estimated `initialCondition` objects.

See Also

initialCondition | compare | ssest | tfest | idss | idpoly | simOptions | sim

More About

- "Estimate Initial Conditions for Simulating Identified Models" on page 20-26
- "Compare Simulated Output with Measured Validation Data" on page 17-23

System Identification App

- "Steps for Using the System Identification App" on page 21-2
- "Working with System Identification App" on page 21-3

Steps for Using the System Identification App

A typical workflow in the **System Identification** app includes the following steps:

1 Import your data into the MATLAB workspace, as described in "Representing Data in MATLAB Workspace" on page 2-8.

2 Start a new session in the System Identification app, or open a saved session. For more information, see "Starting a New Session in the App" on page 21-3.

3 Import data into the app from the MATLAB workspace. For more information, see "Represent Data".

4 Plot and preprocess data to prepare it for system identification. For example, you can remove constant offsets or linear trends (for linear models only), filter data, or select data regions of interest. For more information, see "Preprocess Data".

5 Specify the data for estimation and validation. For more information, see "Specify Estimation and Validation Data in the App" on page 2-22.

6 Select the model type to estimate using the **Estimate** menu.

7 Validate models. For more information, see "Model Validation".

8 Export models to the MATLAB workspace for further analysis. For more information, see "Exporting Models from the App to the MATLAB Workspace" on page 21-8.

Working with System Identification App

Starting and Managing Sessions

What Is a System Identification Session?

A *session* represents the total progress of your identification process, including any data sets and models in the **System Identification** app.

You can save a session to a file with a `.sid` extension. For example, you can save different stages of your progress as different sessions so that you can revert to any stage by simply opening the corresponding session.

To start a new session, see "Starting a New Session in the App" on page 21-3.

For more information about the steps for using the System Identification app, see "Steps for Using the System Identification App" on page 21-2.

Starting a New Session in the App

To start a new session in the System Identification app, type `systemIdentification` in the MATLAB Command Window:

```
systemIdentification
```

Alternatively, you can start a new session by selecting the **Apps** tab of MATLAB desktop. In the **Apps** section, click **System Identification**. This action opens the System Identification app.
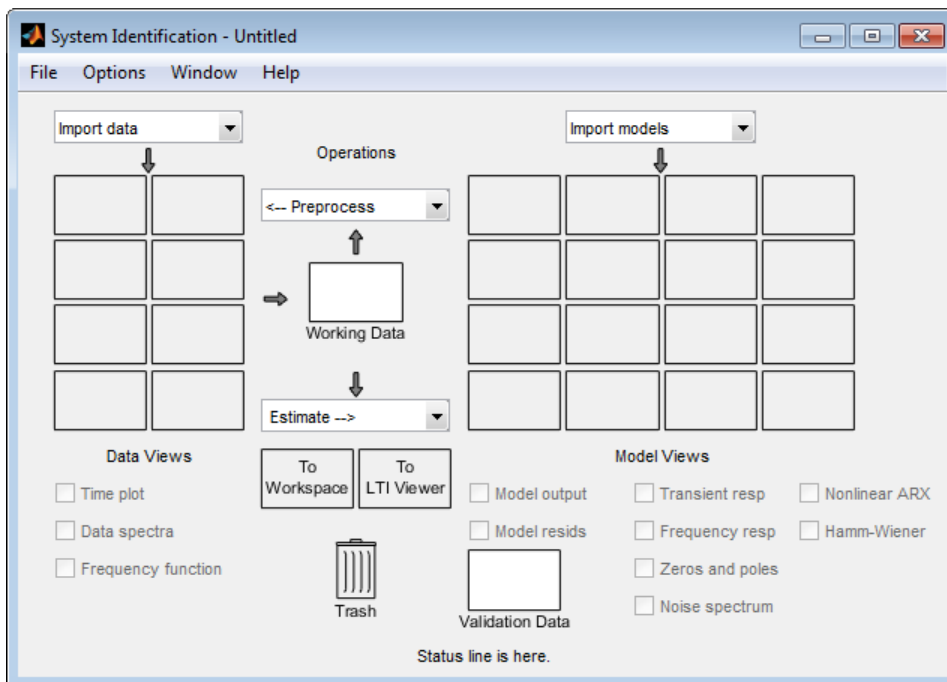
Note Only one session can be open at a time.

You can also start a new session by closing the current session using **File > Close session**. This toolbox prompts you to save your current session if it is not already saved.

Description of the System Identification App Window

The following figure describes the different areas in the System Identification app.

The layout of the window organizes tasks and information from left to right. This organization follows a typical workflow, where you start in the top-left corner by importing data into the System Identification app using the **Import data** menu and end in the bottom-right corner by plotting the characteristics of your estimated model on model plots. For more information about using the System Identification app, see "Steps for Using the System Identification App" on page 21-2.

The **Data Board** area, located below the **Import data** menu in the System Identification app, contains rectangular icons that represent the data you imported into the app.

The Model Board, located to the right of the **<--Preprocess** menu in the System Identification app, contains rectangular icons that represent the models you estimated or imported into the app. You can drag and drop model icons in the Model Board into open dialog boxes.

Opening a Saved Session

You can open a previously saved session using the following syntax:

`systemIdentification(session,path)`

`session` is the file name of the session you want to open and `path` is the location of the session file. Session files have the extension `.sid`. When the session file in on the `matlabpath`, you can omit the `path` argument.

If the System Identification app is already open, you can open a session by selecting **File > Open session**.

Note If there is data in the System Identification app, you must close the current session before you can open a new session by selecting **File > Close session**.

Saving, Merging, and Closing Sessions

The following table summarizes the menu commands for saving, merging, and closing sessions in the System Identification app.

Task	Command	Comment
Close the current session and start a new session.	**File > Close session**	You are prompted to save the current session before closing it.

Task	Command	Comment
Merge the current session with a previously saved session.	**File > Merge session**	You must start a new session and import data or models before you can select to merge it with a previously saved session. You are prompted to select the session file to merge with the current. This operation combines the data and the models of both sessions in the current session.
Save the current session.	**File > Save**	Useful for saving the session repeatedly after you have already saved the session once.
Save the current session under a new name.	**File > Save As**	Useful when you want to save your work incrementally. This command lets you revert to a previous stage, if necessary.

Deleting a Session

To delete a saved session, you must delete the corresponding session file.

Managing Models

Importing Models into the App

You can import System Identification Toolbox models from the MATLAB workspace into the System Identification app. If you have Control System Toolbox software, you can also import any models (LTI objects) you created using this toolbox.

The following procedure assumes that you begin with the System Identification app already open. If this window is not open, type the following command at the prompt:

```
systemIdentification
```

To import models into the System Identification app:

1   Select **Import** from the **Import models** list to open the Import Model Object dialog box.

2   In the **Enter the name** field, type the name of a model object. Press **Enter**.

3   (Optional) In the **Notes** field, type any notes you want to store with this model.

4   Click **Import**.

5   Click **Close** to close the Import Model Object dialog box.

Viewing Model Properties

You can get information about each model in the System Identification app by right-clicking the corresponding model icon.

The Data/model Info dialog box opens. This dialog box describes the contents and the properties of the corresponding model. It also displays any associated notes and the command-line equivalent of the operations you used to create this model.

Renaming Models and Changing Display Color

You can rename a model and change its display color by double-clicking the model icon in the System Identification app.

The Data/model Info dialog box opens. This dialog box describes both the contents and the properties of the model. The object description area displays the syntax of the operations you used to create the model in the app.

To rename the model, enter a new name in the **Model name** field.

You can also specify a new display color using three RGB values in the **Color** field. Each value is between 0 to 1 and indicates the relative presence of red, green, and blue, respectively. For more information about specifying default data color, see "Customizing the System Identification App" on page 21-11.

Finally, you can enter comments about the origin and state of the model in the **Diary And Notes** area.

To view model properties in the MATLAB Command Window, click **Present**.

Organizing Model Icons

You can rearrange model icons in the System Identification app by dragging and dropping the icons to empty Model Board rectangles.

When you need additional space for organizing model icons, select **Options > Extra model/data board** in the System Identification app. This action opens an extra session window with blank rectangles. The new window is an extension of the current session and does not represent a new session.

Tip When you import or estimate models and there is insufficient space for the icons, an additional session window opens automatically.

You can drag and drop model icons between the main System Identification app and any extra session windows.

Type comments in the **Notes** field to describe the models. When you save a session, as described in "Saving, Merging, and Closing Sessions" on page 21-4, all additional windows and notes are also saved.

Deleting Models in the App

To delete models in the System Identification app, drag and drop the corresponding icon into **Trash**. You can also use the **Delete** key on your keyboard to move items to the **Trash**. Moving items to **Trash** does not permanently delete these items.

To restore a model from **Trash**, drag its icon from **Trash** to the Model Board in the System Identification app. You can view the **Trash** contents by double-clicking the **Trash** icon.

Note You must restore a model to the Model Board; you cannot drag model icons to the Data Board.
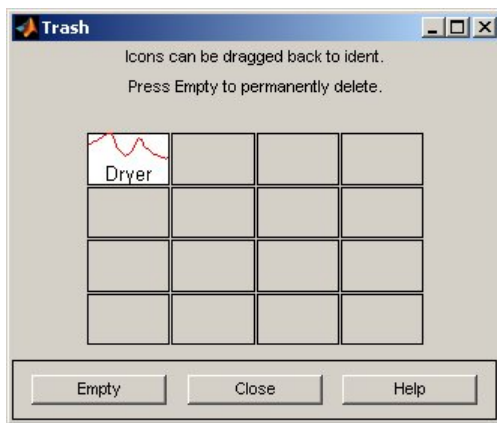
To permanently delete all items in **Trash**, select **Options > Empty trash**.

Exiting a session empties **Trash** automatically.

Exporting Models from the App to the MATLAB Workspace

The models you create in the System Identification app are not available in the MATLAB workspace until you export them. Exporting is necessary when you need to perform an operation on the model that is only available at the command line. Exporting models to the MATLAB workspace also makes them available to the Simulink software or another toolbox, such as the Control System Toolbox product.

To export a model to the MATLAB workspace, do one of the following:

- Drag and drop the corresponding icon to the **To Workspace** rectangle.
- Right-click the icon to open the Data/model Info dialog box. Click **Export** to export the model.

When you export models to the MATLAB workspace, the resulting variables have the same name as in the System Identification app.

Working with Plots

Identifying Data Sets and Models on Plots

You can identify data sets and models on a plot by color: the color of the line in the data or model icon in the System Identification app matches the line color on the plots.

You can also display data tips for each line on the plot by clicking a plot curve and holding down the mouse button.

Note You must disable zoom by selecting **Style > Zoom** before you can display data tips. For more information about enabling zoom, see "Magnifying Plots" on page 21-9.

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.
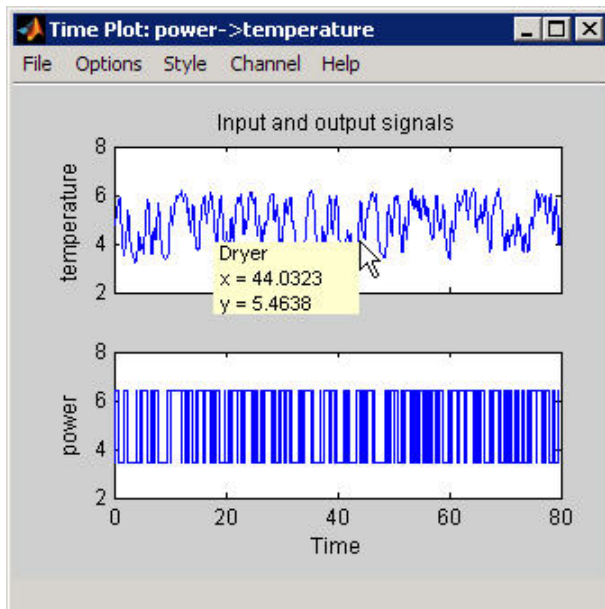
Data Tip on a Plot

Changing and Restoring Default Axis Limits

There are two ways to change which portion of the plot is currently in view:

- Magnifying plots
- Setting axis limits

Magnifying Plots

Enable zoom by selecting **Style > Zoom** in the plot window. To disable zoom, select **Style > Zoom** again.

Tip To verify that zoom is active, click the **Style** menu. A check mark should appear next to **Zoom**.

You can adjust magnification in the following ways:

- To zoom in default increments, left-click the portion of the plot you want to center in the plot window.
- To zoom in on a specific region, click and drag a rectangle that identifies the region for magnification. When you release the mouse button, the selected region is displayed.
- To zoom out, right-click on the plot.

Note To restore the full range of the data in view, select **Options > Autorange** in the plot window.

Setting Axis Limits

You can change axis limits for the vertical and the horizontal axes of the input and output channels that are currently displayed on the plot.

1   Select **Options > Set axes limits** to open the Limits dialog box.

2   Specify a new range for each axis by editing its lower and upper limits. The limits must be entered using the format *[LowerLimit UpperLimit]*. Click **Apply**. For example:

```
[0.1 100]
```

Note To restore full axis limits, select the **Auto** check box to the right of the axis name, and click **Apply**.

3   To plot data on a linear scale, clear the **Log** check box to the right of the axis name, and click **Apply**.

Note To revert to base-10 logarithmic scale, select the **Log** check box to the right of the axis name, and click **Apply**.

4   Click **Close**.

Note To view the entire data range, select **Options > Autorange** in the plot window.

Selecting Measured and Noise Channels in Plots

Model inputs and outputs are called *channels*. When you create a plot of a multivariable input-output data set or model, the plot only shows one input-output channel pair at a time. The selected channel names are displayed in the title bar of the plot window.

Note When you select to plot multiple data sets, and each data set contains several input and output channels, the **Channel** menu lists channel pairs from all data sets.

You can select a different input-output channel pair from the **Channel** menu in any System Identification Toolbox plot window.

The **Channel** menu uses the following notation for channels: u1->y2 means that the plot displays a transfer function from input channel u1 to output channel y2. System Identification Toolbox estimates as many noise sources as there are output channels. In general, e@ynam indicates that the noise source corresponds to the output with name ynam.

For example, e@y3->y1 means that the transfer function from the noise channel (associated with y3) to output channel y2 is displayed. For more information about noise channels, see "Separation of Measured and Noise Components of Models" on page 4-33.

Tip When you import data into the System Identification app, it is helpful to assign meaningful channel names in the Import Data dialog box. For more information about importing data, see "Represent Data".

Grid and Line Styles in Plots

There are several **Style** options that are common to all plot types.

Grid Lines

To toggle showing or hiding grid lines, select **Style > Grid**.

Solid or Dashed Lines

To display currently visible lines as a combination of solid, dashed, dotted, and dash-dotted line style, select **Style > Separate linestyles**.

To display all solid lines, select **Style > All solid lines**. This choice is the default.

All line styles match the color of the corresponding data or model icon in the System Identification app.

Opening a Plot in a MATLAB Figure Window

The MATLAB Figure window provides editing and printing commands for plots that are not available in the System Identification Toolbox plot window. To take advantage of this functionality, you can first create a plot in the System Identification app, and then open it in a MATLAB Figure window to fine-tune the display.

After you create the plot, as described in "Plot Models in the System Identification App" on page 17-5, select **File > Copy figure** in the plot window. This command opens the plot in a MATLAB Figure window.

Printing Plots

To print a System Identification Toolbox plot, select **File > Print** in the plot window. In the Print dialog box, select the printing options and click **OK**.

Customizing the System Identification App

Types of App Customization

The System Identification app lets you customize the window behavior and appearance. For example, you can set the size and position of specific dialog boxes and modify the appearance of plots.

You can save the session to save the customized app state.

You might choose to edit the file that controls default settings, as described in "Modifying idlayout.m" on page 21-12 (advanced usage).

Saving Session Preferences

Use **Options > Save preferences** to save the current state of the System Identification app. This command saves the following settings to a preferences file, `idprefs.mat`:

- Size and position of the System Identification app
- Sizes and positions of dialog boxes
- Four recently used sessions
- Plot options, such as line styles, zoom, grid, and whether the input is plotted using zero-order hold or first-order hold between samples

You can only edit `idprefs.mat` by changing preferences in the app.

The `idprefs.mat` file is located in the same folder as `startup.m`, by default. To change the location where your preferences are saved, use the `midprefs` command with the new path as the argument. For example:

```
midprefs('c:\matlab\toolbox\local\')
```

You can also type `midprefs` and browse to the desired folder.

To restore the default preferences, select **Options > Default preferences**.

Modifying idlayout.m

You might want to customize the default plot options by editing `idlayout.m` (advanced usage).

To customize `idlayout.m` defaults, save a copy of `idlayout.m` to a folder in your `matlabpath` just above the `ident` folder level.

Caution Do not edit the original file to avoid overwriting the `idlayout.m` defaults shipped with the product.

You can customize the following plot options in `idlayout.m`:

* Order in which colors are assigned to data and model icons
* Line colors on plots
* Axis limits and tick marks
* Plot options, set in the plot menus
* Font size

Note When you save preferences using **Options > Save preferences** to `idprefs.mat`, these preferences override the defaults in `idlayout.m`. To give `idlayout.m` precedence every time you start a new session, select **Options > Default preferences**.

System Identification UI Help

Initial Conditions

Specify initial conditions for the system under **Systems**. To see the choices, click the entry under **Initial Conditions**. You can select one of the following options.

- `Estimate` — Estimate initial conditions such that the prediction error for observed output is minimized.
- `Zero` — Zero initial conditions.
- `Absorb Delays and Estimate` — Similar to `Estimate`, but this option also absorbs nonzero delays into the model coefficients. The software converts delays into explicit model states, and estimates and returns the initial values of those states.

  Use this option for linear models only.
- `Custom` — Specify the initial conditions. Use this option when you already know or have an estimate for the initial conditions. Specify a column vector of length equal to the order of the model.

For nonlinear grey-box models, when you select `Estimate`, the software estimates only those initial states `i` that are designated as free in the model, as indicated by the property `sys.InitialStates(i).Fixed` equal to `false`. To estimate all the states of the model, first specify all the `Nx` states of the `idnlgrey` model `sys` as free.

```
for i = 1:Nx
sys.InitialStates(i).Fixed = false;
end
```

Similarly, to fix all the initial states to values specified in `sys.InitialStates`, first specify all the states as fixed in the `sys.InitialStates` property of the nonlinear grey-box model.

When you have completed your selection, regenerate the response plot by clicking **Simulate**.

Diagnostics and Prognostics

# Time Series Prediction and Forecasting for Prognosis

This example shows how to create a time series model and use the model for prediction, forecasting, and state estimation. The measured data is from an induction furnace whose slot size erodes over time. The slot size cannot be measured directly but the furnace current and consumed power are measured. It is known that as the slot size increases, the slot resistance decreases. The ratio of measured current squared to measured power is thus proportional to the slot size. You use the measured current-power ratio (both current and power measurements are noisy) to create a time series model and use the model to estimate the current slot size and forecast the future slot size. Through physical inspection the induction furnace slot size is known at some points in time.

Load and Plot the Measured Data

The measured current-power ratio data is stored in the `iddata_TimeSeriesPrediction` MATLAB file. The data is measured at hourly intervals and shows that over time the ratio increases indicating erosion of the furnace slot. You develop a time series model using this data. Start by separating the data into an identification and a validation segment.

```
load iddata_TimeSeriesPrediction
n = numel(y);
ns = floor(n/2);
y_id = y(1:ns,:);
y_v = y((ns+1:end),:);
data_id = iddata(y_id, [], Ts, 'TimeUnit', 'hours');
data_v  = iddata(y_v, [], Ts, 'TimeUnit', 'hours', 'Tstart', ns+1);

plot(data_id,data_v)
legend('Identification data','Validation data','location','SouthEast');
```
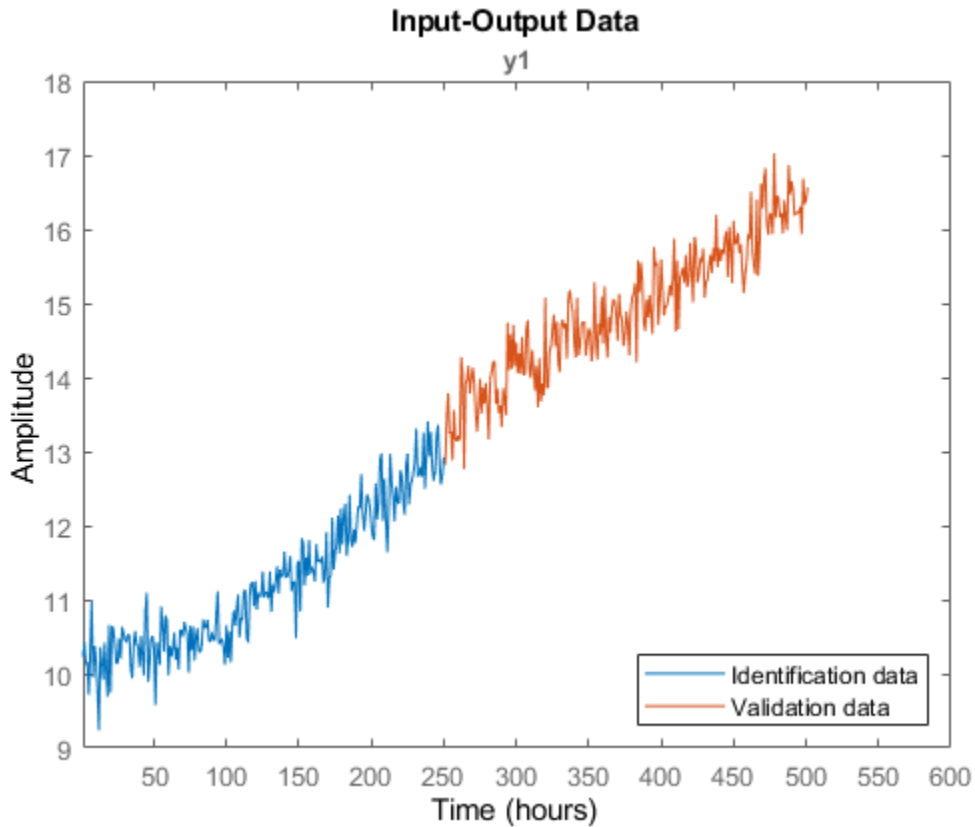
Model Identification

The slot erosion can be modelled as a state-space system with noise input and measured current-power ratio as output. The measured current-power ratio is proportional to the system state, or

$$x_{n+1} = Ax_n + Ke_n$$

$$y_n = Cx_n + e_n$$

Where x_n the state vector, contains the slot size; y_n is the measured current-power ratio; e_n noise and A, C, K are to be identified.

Use the `ssest()` command to identify a discrete state-space model from the measured data.

```
sys = ssest(data_id,1,'Ts',Ts,'form','canonical')
```

```
sys =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + K e(t)
        y(t) = C x(t) + e(t)

  A =
          x1
    x1  1.001

  C =
```

```
          x1
    y1     1

  K =
             y1
    x1   0.09465

Sample time: 1 hours

Parameterization:
   CANONICAL form with indices: 1.
   Disturbance component: estimate
   Number of free coefficients: 2
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on time domain data "data_id".
Fit to estimation data: 67.38% (prediction focus)
FPE: 0.09575, MSE: 0.09348
```

The identified model minimizes the 1-step ahead prediction. Validate the model using a 10 step ahead predictor, i.e., given $y_0, \cdots, y_n$ use the model to predict y_{n+10}. Note that the error between the measured and predicted values, $y_0 - \hat{y}_0, \cdots, y_n - \hat{y}_n$, are used to make the y_{n+10} prediction.

Use the 10 step ahead predictor for the identification data and the independent validation data.

```
nstep = 10;
compare(sys,data_id,nstep)   % comparison of 10-step prediction to estimation data
grid('on');
```

10-Step Predicted Response Comparison
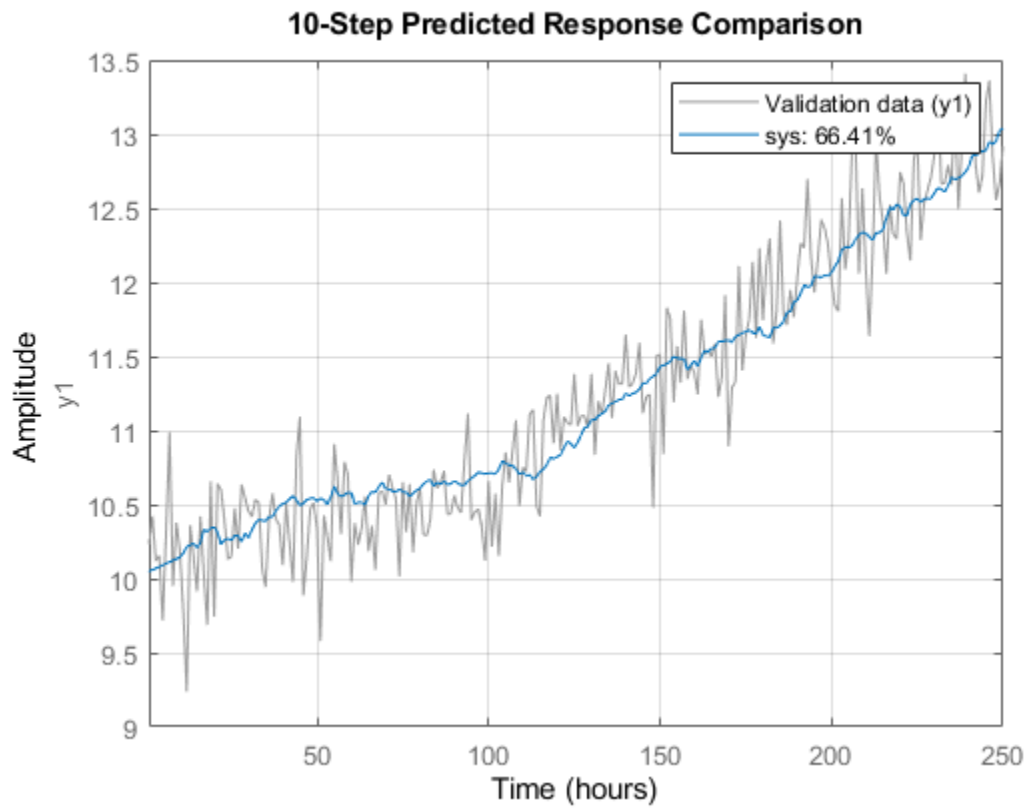
```
figure; compare(sys,data_v,nstep)   % comparison to validation data
grid('on');
```

10-Step Predicted Response Comparison
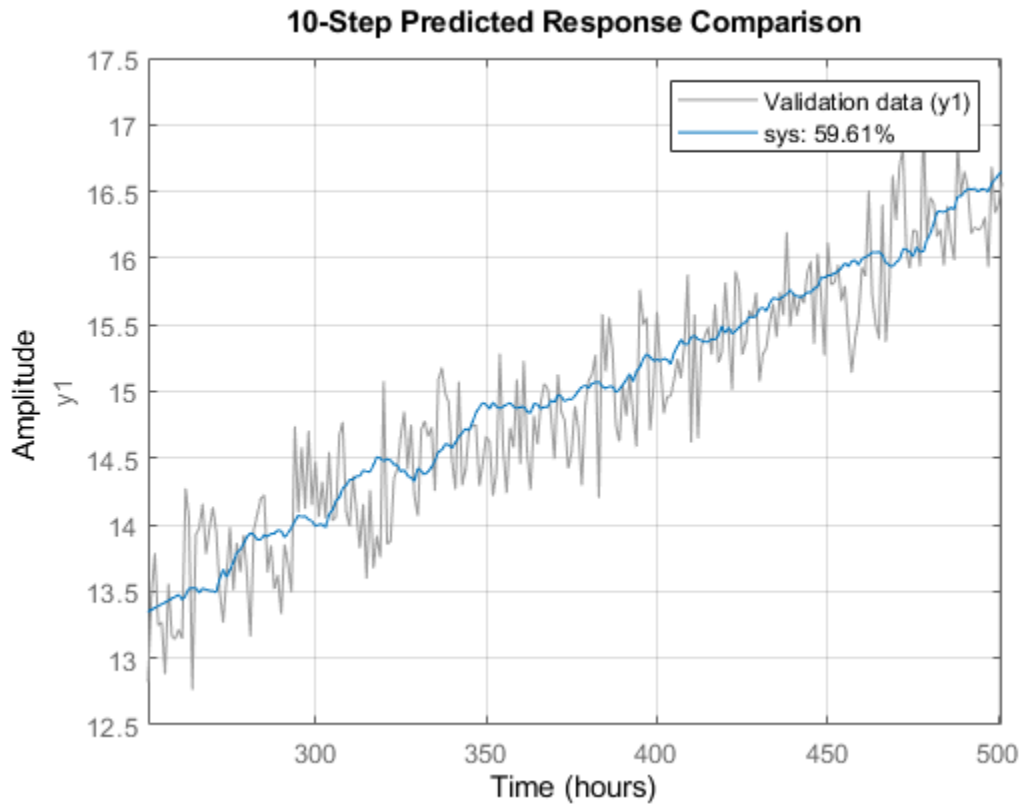
The above exercise Both data sets show that the predictor matches the measured data.

Forecasting is used to further verify the model. Forecasting uses the measured data record $y_0, y_1, ..., y_n - \hat{y}_n$ to compute the model state at time step n. This value is used as initial condition for forecasting the model response for a future time span. We forecast the model response over the time span of the validation data and then compare the two. We can also compute the uncertainty in forecasts and plot +/- 3 sd of their values.

```
MeasuredData = iddata(y, [], Ts, 'TimeUnit', 'hours'); % = [data_id;data_v]
t0 = MeasuredData.SamplingInstants;

Horizon = size(data_v,1); % forecasting horizon
[yF, ~, ~, yFSD]  = forecast(sys, data_id, Horizon);
% Note: yF is IDDATA object while yFSD is a double vector
t = yF.SamplingInstants; % extract time samples
yFData = yF.OutputData;      % extract response as double vector
plot(MeasuredData)
hold on
plot(t, yFData, 'r.-', t, yFData+3*yFSD, 'r--', t, yFData-3*yFSD, 'r--')
hold off
title('Forecasted response over the validation data''s time span')
grid on
```

**Forecasted response over the validation data's time span**
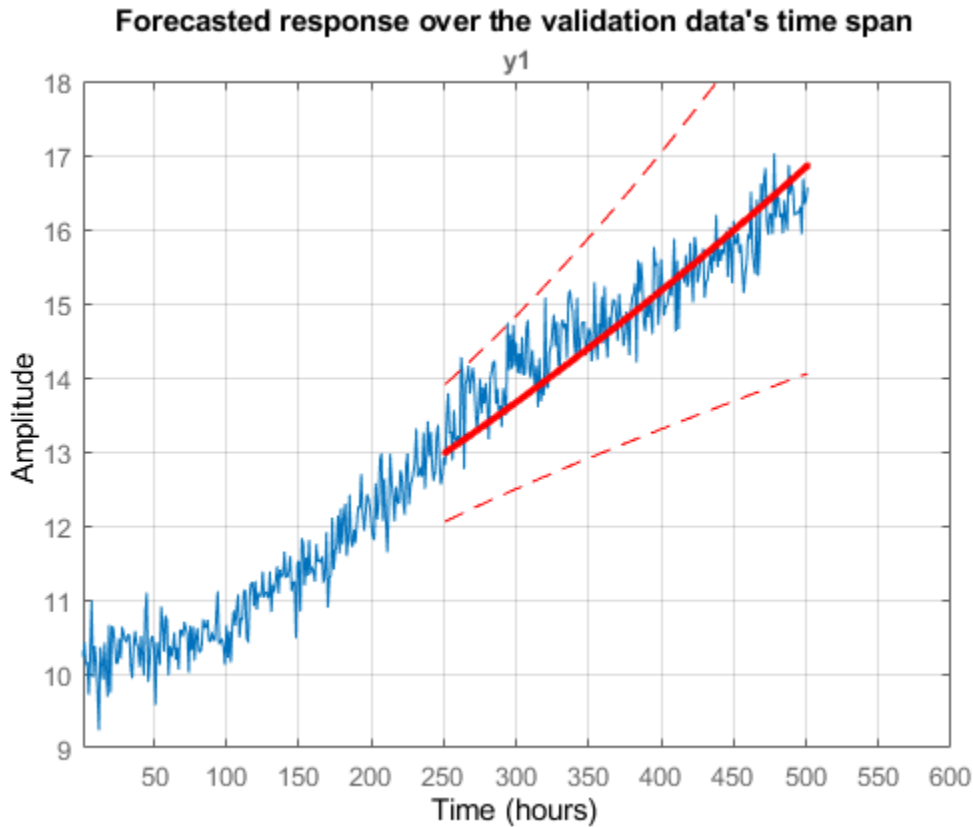
The plot shows that the model response with confidence intervals (indicated by the red colored dashed curves) overlap the measured value for the validation data. The combined prediction and forecasting results indicate that the model represents the measured current-power ratio.

The forecasting results also show that over large horizons the model variance is large and for practical purposes future forecasts should be limited to short horizons. For the induction furnace model a horizon of 200 hours is appropriate.

Finally we use the model to forecast the response 200 steps into future for the time span of 502-701 hours.

```
Horizon = 200; % forecasting horizon
[yFuture, ~, ~, yFutureSD] = forecast(sys, MeasuredData, Horizon);
t = yFuture.SamplingInstants; % extract time samples
yFutureData = yFuture.OutputData;      % extract response as double vector
plot(t0, y,...
   t, yFutureData, 'r.-', ...
   t, yFutureData+3*yFutureSD, 'r--', ...
   t, yFutureData-3*yFutureSD, 'r--')
title('Forecasted response (200 steps)')
grid on
```
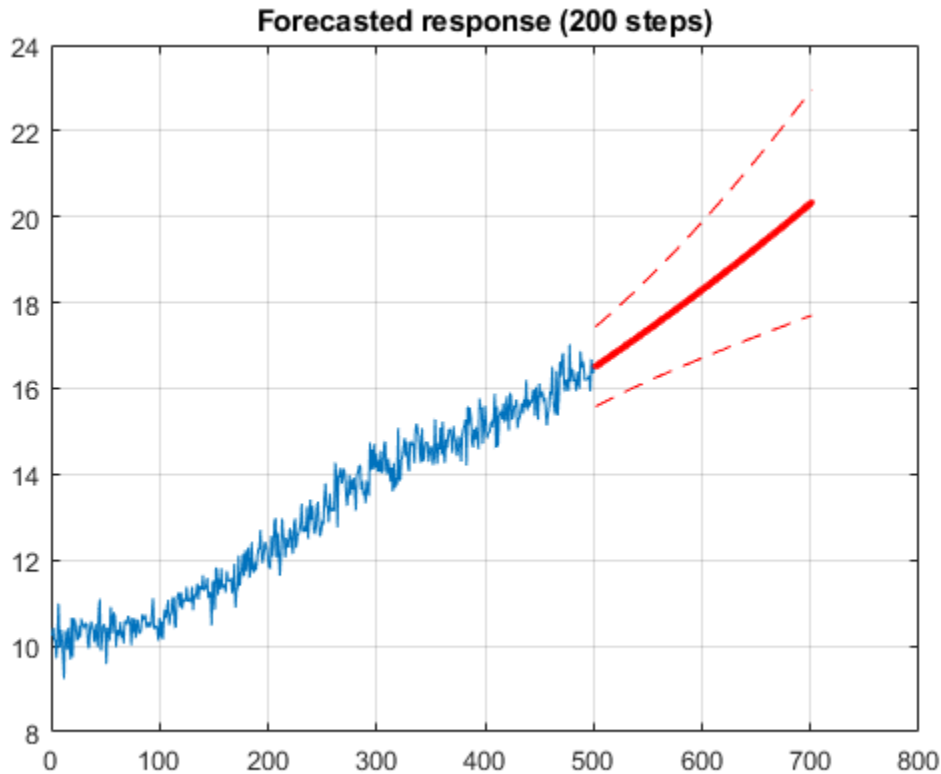
The blue curve shows the measured data that spans over 1-501 hours. The red curve is the forecasted response for 200 hours beyond the measured data's time range. The red dashed curves shows the 3 sd uncertainty in the forecasted response based on random sampling of the identified model.

State Estimation

The identified model matches the measured current-power ratio but we are interested in the furnace slot size which is a state in the model. The identified model has an arbitrary state that can be transformed so that the state has meaning, in this case the slot size.

Create a predictor for the arbitrary state. The identified model covariances need to be translated to the predictor model using the `translatecov()` command. The `createPredictor()` function simply extracts the third output argument of the `predict()` function to be used with `translatecov()`.

```
type createPredictor
est = translatecov(@(s) createPredictor(s,data_id),sys)
```

```
function pred = createPredictor(mdl,data)
%CREATEPREDICTOR Return 1-step ahead predictor.
%
%    sys = createPredictor(mdl,data)
%
%    Create a 1-step ahead predictor model sys for the specified model mdl
%    and measured data. The function is used by
%    |TimeSeriedPredictionExample| and the |translatecov()| command to
```

```
%    translate the identified model covariance to the predictor.

% Copyright 2015 The MathWorks, Inc.
[~,~,pred] = predict(mdl,data,1);

est =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + B u(t)
        y(t) = C x(t) + D u(t)

  A =
             x1
   x1   0.9064

  B =
             y1
   x1   0.09465

  C =
        x1
   y1    1

  D =
        y1
   y1    0

Sample time: 1 hours

Parameterization:
   CANONICAL form with indices: 1.
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 2
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

The model `est` is a 1-step ahead predictor expressed in the same state coordinates as the original model `sys`. How do we transform the state coordinates so that the model's state corresponds to the (time dependent) slot size? The solution is to rely on actual, direct measurements of the slot size taken intermittently. This is not uncommon in practice where the cost of taking direct measurements is high and only be done periodically (such as when the component is being replaced).

Specifically, transform the predictor state, x_n, to z_n, so that $y_n = C z_n$ where y_n the measured current-power ratio, and z_n is the furnace slot size. In this example, four direct measurements of the furnace slot size, `sizeMeasured`, and furnace current-power ratio, `ySizeMeasured`, are used to estimate C. In transforming the predictor the predictor covariances also need to be transformed. Hence we use the `translatecov()` command to carry out the state coordinate transformation.

```
Cnew = sizeMeasured\ySizeMeasured;
est  = translatecov(@(s) ss2ss(s,s.C/Cnew),est)


est =
  Discrete-time identified state-space model:
```

```
    x(t+Ts) = A x(t) + B u(t)
        y(t) = C x(t) + D u(t)

  A =
            x1
   x1  0.9064

  B =
            y1
   x1  0.9452

  C =
            x1
   y1  0.1001

  D =
       y1
   y1   0

Sample time: 1 hours

Parameterization:
   CANONICAL form with indices: 1.
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 2
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

The predictor is now expressed in the desired state coordinates. It has one input that is the measured system output (the furnace current-power ratio) and one output that is the predicted system output (the furnace slot size). The predictor is simulated to estimate the system output and system state.

```
opts = simOptions;
opts.InitialCondition = sizeMeasured(1);
U = iddata([],[data_id.Y; data_v.Y],Ts,'TimeUnit','hours');
[ye,ye_sd,xe] = sim(est,U,opts);
```

Compare the estimated output and slot size with measured and known values.

```
yesdp   = ye;
yesdp.Y = ye.Y+3*ye_sd;
yesdn   = ye;
yesdn.Y = ye.Y-3*ye_sd;
n = numel(xe);
figure, plot([data_id;data_v],ye,yesdp,'g',yesdn,'g')
legend('Measured output','Estimated output','99.7% bound','location','SouthEast')
grid('on')
figure, plot(tSizeMeasured,sizeMeasured,'r*',1:n,xe,1:n,yesdp.Y/est.C,'g',1:n,yesdn.Y/est.C,'g')
legend('Measured state','Estimated state','99.7% bound','location','SouthEast')
xlabel('Time (hours)')
ylabel('Amplitude');
grid('on')
```
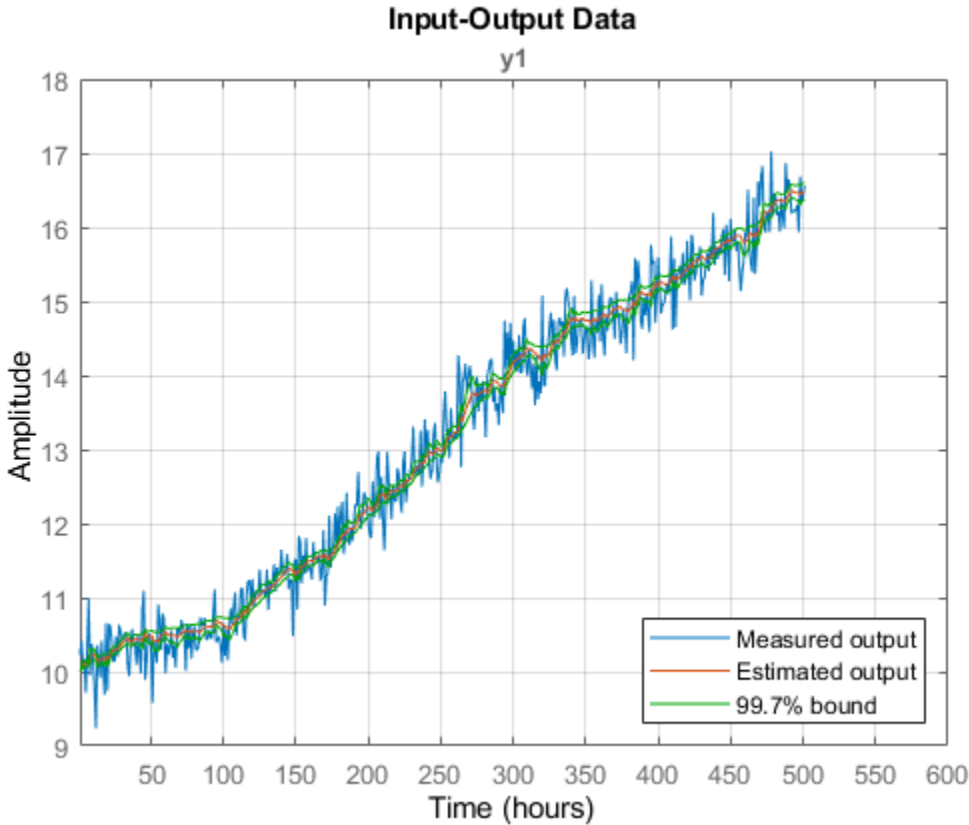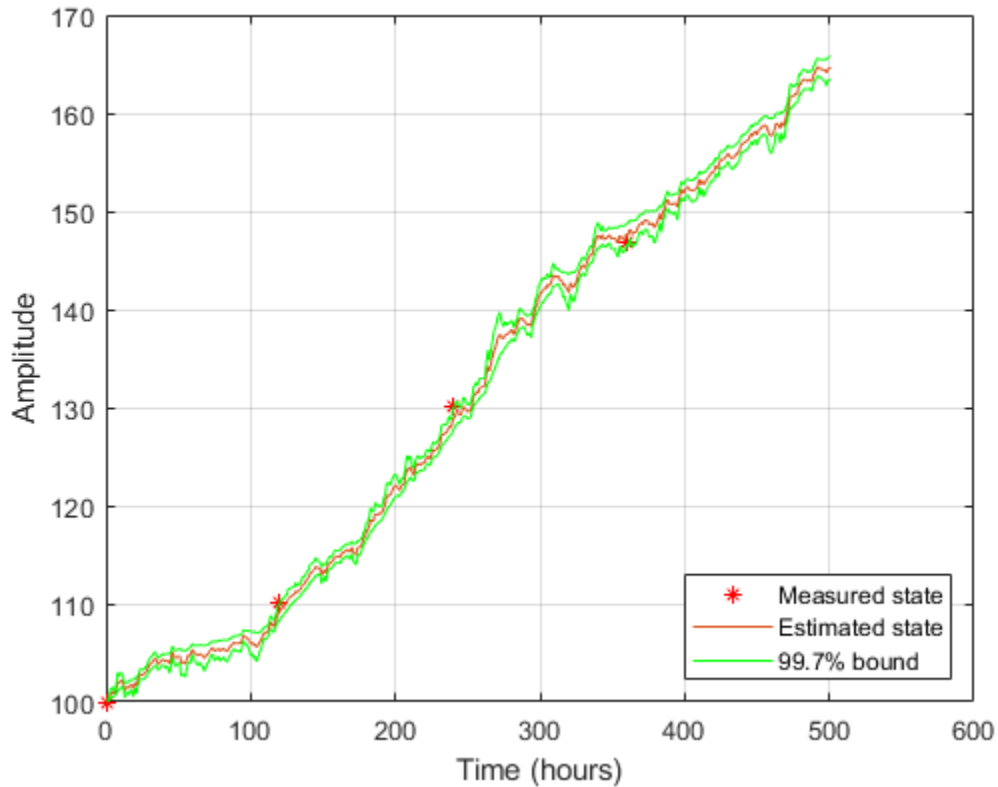
Using Prediction and Forecasting for Prognosis

The combination of predictor model and forecasting allow us to perform prognosis on the induction furnace.

The predictor model allows us to estimate the current furnace slot size based on measured data. If the estimated value is at or near critical values an inspection or maintenance can be scheduled. Forecasting allows us to, from the estimated current state, predict the future system behavior allowing us to predict when an inspection or maintenance may be needed.

Further the predictor and forecast model can be re-identified as more data becomes available. In this example one data set was used to identify the predictor and forecast models but as more data is accumulated the models can be re-identified.

See Also
predict | forecast | compare

More About
- "Forecast Multivariate Time Series" on page 17-25
- "Introduction to Forecasting of Dynamic System Response" on page 14-34