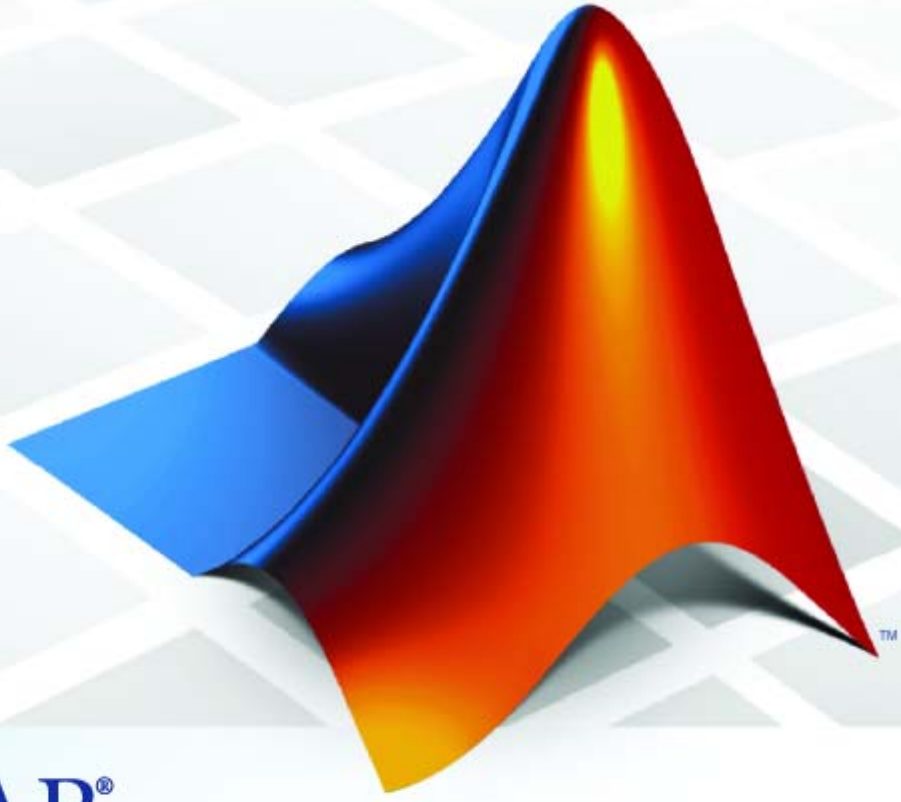
System Identification Toolbox™ 7
Reference

Lennart Ljung



MATLAB®
&SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

System Identification Toolbox™ Reference

© COPYRIGHT 1988–2008 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

September 2007	Online only	Revised for Version 7.1 (Release 2007b)
March 2008	Online only	Revised for Version 7.2 (Release 2008a)

Contents

1 Function Reference

Preparing Data 1-3

Identifying Linear Models 1-5

Identifying Nonlinear Black-Box Models 1-8

Estimating ODE Parameters 1-12

Recursive Techniques for Identifying Linear Models . . 1-12

Validating and Analyzing Models 1-13

Simulating and Predicting Model Output 1-15

Using Models with Other Products 1-16

Customizing and Using GUI 1-17

2 Functions – Alphabetical List

3 Block Reference

Preparing Data 3-2

Identifying Linear Models **3-3**

Simulating and Predicting Model Output **3-4**

4 Blocks — Alphabetical List

Index

Function Reference

Preparing Data (p. 1-3)	Construct data objects and input signals, filter, resample, detrend, transform, identify delay and feedback, and get and set data properties
Identifying Linear Models (p. 1-5)	Estimate nonparametric models using correlation and spectral analysis, compute impulse and step response, and estimate empirical transfer functions, estimate discrete- and continuous-time linear polynomial transfer functions and state-space models from time- and frequency-domain data, and select model structure and order based on loss function, AIC, and MDL criteria
Identifying Nonlinear Black-Box Models (p. 1-8)	Estimate input-output, black-box nonlinear models, including nonlinear ARX and Hammerstein-Wiener models
Estimating ODE Parameters (p. 1-12)	Estimate linear and nonlinear grey-box models, set initial parameters and initial state values, and randomize initial parameter values

Recursive Techniques for Identifying Linear Models (p. 1-12)	Recursively estimate input-output linear models, such as AR, ARX, ARMAX, Box-Jenkins, and Output-Error models
Validating and Analyzing Models (p. 1-13)	Validate and analyze models by comparing model output, plotting models with confidence regions, computing standard deviation and prediction errors, computing loss function, getting advice on estimated models, and extracting numerical information from linear models
Simulating and Predicting Model Output (p. 1-15)	Simulate and predict model output, compute prediction errors, and generate input data
Using Models with Other Products (p. 1-16)	Reduce model order, convert between System Identification Toolbox™ and LTI objects, and perform linear analysis using LTI Viewer
Customizing and Using GUI (p. 1-17)	Start System Identification Toolbox GUI and set preferences

Preparing Data

`advice`	Analysis and recommendations for data or estimated linear polynomial and state-space models
`covf`	Estimate covariance functions for time-domain `iddata` object
`delayest`	Estimate time delay (dead time) from data
`detrend`	Subtract trend from time-domain, frequency-domain, or time-series data signal
`diff`	Difference signals in `iddata` objects
`fcat`	Concatenate frequency-domain signals in `idfrd` and `iddata` objects
`feedback`	Identify possible feedback in `iddata` data
`fft`	Transform `iddata` object to frequency domain
`fselect`	Frequencies from `idfrd` object
`get`	Query properties of data and model objects
`getexp`	Specific experiment(s) from multiple-experiment `iddata` object
`iddata`	Class for storing time-domain and frequency-domain data
`idfilt`	Filter data using user-defined passbands, general filters, or Butterworth filters
`idfrd`	Class for storing frequency-response or spectral-analysis data or frequency-response models
`idinput`	Generate input signals

`idresamp`	Resample time-domain data by decimation or interpolation
`ifft`	Transform `iddata` objects from frequency to time domain
`isreal`	Determine whether model parameters or data values are real
`merge (iddata)`	Merge data sets into one `iddata` object
`misdata`	Reconstruct missing input and output data
`nkshift`	Shift data sequences
`pexcit`	Level of excitation of input signals
`plot`	Plot `iddata` or model objects
`realdata`	Determine whether `iddata` is based on real-valued signals
`resample`	Resample time-domain data by decimation or interpolation (requires Signal Processing Toolbox™ software)
`set`	Set properties of data and model objects
`size`	Dimensions of `iddata`, `idmodel`, and `idfrd` objects
`timestamp`	Return date and time when object was created or last modified

Identifying Linear Models

`ar`	Estimate parameters of AR model for scalar time series returning `idpoly` object
`armax`	Estimate parameters of ARMAX or ARMA model returning `idpoly` object
`arx`	Estimate parameters of ARX or AR model using least squares returning `idpoly` or `idarx` object
`arxdata`	ARX parameters from multiple-output `idarx` or single-output `idpoly` objects with variance information
`arxstruc`	Compute and compare loss functions for single-output ARX models
`balred`	Reduce model order (requires Control System Toolbox™ product)
`bj`	Estimate parameters of Box-Jenkins model returning `idpoly` object
`c2d`	Convert model from continuous to discrete time
`cra`	Estimate impulse response using prewhitened-based correlation analysis
`d2c`	Convert model from discrete to continuous time
`delayest`	Estimate time delay (dead time) from data
`etfe`	Estimate empirical transfer functions and periodograms returning `idfrd` object

`feedback`	Identify possible feedback in `iddata` data
`freqresp`	Frequency-response data from `idmodel` or `idfrd` object
`get`	Query properties of data and model objects
`idarx`	Class for storing multiple-output ARX polynomials and estimated impulse- and step-response models
`idfrd`	Class for storing frequency-response or spectral-analysis data or frequency-response models
`idgrey`	Class for storing linear ODE models
`idmodel`	Superclass for linear models
`idpoly`	Class for storing linear polynomial input-output models
`idproc`	Class for storing low-order, continuous-time process models
`idss`	Class for storing linear state-space models with known and unknown parameters
`impulse`	Plot impulse response with confidence interval
`init`	Set or randomize initial parameter values
`iv4`	Estimate ARX model using four-stage instrumental variable method returning `idpoly` or `idarx` object
`ivar`	Estimate AR model using instrumental variable method returning `idpoly` object

`ivstruc`	Compute loss functions for sets of output-error model structures
`ivx`	Estimate parameters of ARX model using instrumental variable method with arbitrary instruments returning `idpoly` or `idarx` object
`merge`	Merge estimated `idmodel` models
`n4sid`	Estimate state-space models using subspace method returning `idss` object
`nuderst`	Set step size for numerical differentiation
`oe`	Estimate parameters of output-error model returning `idpoly` object
`pem`	Estimate model parameters using iterative prediction-error minimization method
`pexcit`	Level of excitation of input signals
`polydata`	Polynomial model parameters from single-input and single-output `idpoly` object
`selstruc`	Select model order for single-output ARX models
`set`	Set properties of data and model objects
`setpname`	Set mnemonic parameter names for black-box model structures
`setstruc`	Set matrix structure for `idss` objects
`size`	Dimensions of `iddata`, `idmodel`, and `idfrd` objects
`spa`	Estimate frequency response and spectrum using spectral analysis returning `idfrd` object

`spafdr`	Estimate frequency response and spectrum using spectral analysis with frequency-dependent resolution returning `idfrd` object
`ssdata`	State-space matrices from `idmodel` object
`step`	Plot step response with confidence interval
`struc`	Generate model structure matrices for single-input and single-output systems
`tfdata`	Numerator and denominator of transfer function from `idmodel` object
`timestamp`	Return date and time when object was created or last modified
`zpkdata`	Zeros, poles, and gains of transfer function from `idmodel` object

Identifying Nonlinear Black-Box Models

`addreg`	Add custom regressors to `idnalrx` model
`customnet`	Class representing nonlinearity estimator with user-defined unit function for nonlinear ARX and Hammerstein-Wiener models
`customreg`	Class representing custom regressor for nonlinear ARX models
`data2state(idnlarx)`	Map past input/output data to current states of `idnlarx` model

`deadzone`	Class representing dead-zone nonlinearity estimator for Hammerstein-Wiener models
`evaluate`	Value of nonlinearity estimator at given input
`findop(idnlarx)`	Compute operating point for nonlinear ARX model
`findop(idnlhw)`	Compute operating point for Hammerstein-Wiener model
`get`	Query properties of data and model objects
`getDelayInfo(idnlarx)`	Get input/output delay information for `idnlarx` model structure
`getreg`	Names of standard or custom regressors in nonlinear ARX model
`idnlarx`	Class representing nonlinear ARX models
`idnlhw`	Class representing Hammerstein-Wiener input-output models
`idnlmodel`	Superclass for nonlinear models
`init`	Set or randomize initial parameter values
`linapp`	Linear approximation of nonlinear ARX and Hammerstein-Wiener models for given input
`linear`	Specify to estimate nonlinear ARX model that is linear in (nonlinear) custom regressors
`linearize(idnlarx)`	Linearize nonlinear ARX model
`linearize(idnlhw)`	Linearize Hammerstein-Wiener model

`neuralnet`	Class representing neural network object created in Neural Network Toolbox™ product for estimating nonlinear ARX and Hammerstein-Wiener models
`nlarx`	Estimate nonlinear ARX models
`nlhw`	Estimate Hammerstein-Wiener models
`operspec(idnlarx)`	Construct operating point specification object for `idnlarx` model
`operspec(idnlhw)`	Construct operating point specification object for `idnlhw` model
`pem`	Estimate model parameters using iterative prediction-error minimization method
`poly1d`	Class representing single-variable polynomial nonlinear estimator for Hammerstein-Wiener models
`polyreg`	Generate custom regressors by computing powers and products of standard regressors
`pwlinear`	Class representing piecewise-linear nonlinear estimator for Hammerstein-Wiener models
`saturation`	Class representing saturation nonlinearity estimator for Hammerstein-Wiener models
`set`	Set properties of data and model objects

`sigmoidnet`	Class representing sigmoid network nonlinearity estimator for nonlinear ARX and Hammerstein-Wiener models
`treepartition`	Class representing binary-tree nonlinearity estimator for nonlinear ARX models
`unitgain`	Specify absence of nonlinearities for specific input or output channels in Hammerstein-Wiener models
`wavenet`	Class representing wavelet network nonlinearity estimator for nonlinear ARX and Hammerstein-Wiener models

Estimating ODE Parameters

`get`	Query properties of data and model objects
`getinit`	Values of `idnlgrey` model initial states
`getpar`	Parameter values and properties of `idnlgrey` model parameters
`idgrey`	Class for storing linear ODE models
`idnlgrey`	Class representing nonlinear ODE models
`idnlmodel`	Superclass for nonlinear models
`init`	Set or randomize initial parameter values
`pem`	Estimate model parameters using iterative prediction-error minimization method
`set`	Set properties of data and model objects
`setinit`	Set initial states of `idnlgrey` model object
`setpar`	Set initial parameter values of `idnlgrey` model object

Recursive Techniques for Identifying Linear Models

`rarmax`	Estimate recursively parameters of ARMAX or ARMA models
`rarx`	Estimate recursively parameters of ARX or AR models

`rbj`	Estimate recursively parameters of Box-Jenkins models
`roe`	Estimate recursively output-error models (IIR-filters)
`rpem`	Estimate general input-output models using recursive prediction-error minimization method
`rplr`	Estimate general input-output models using recursive pseudolinear regression method
`segment`	Segment data and estimate models for each segment

Validating and Analyzing Models

`advice`	Analysis and recommendations for data or estimated linear polynomial and state-space models
`aic`	Akaike Information Criterion for estimated model
`arxdata`	ARX parameters from multiple-output `idarx` or single-output `idpoly` objects with variance information
`bode`	Plot Bode diagram of frequency response with confidence interval
`compare`	Compare model output and measured output
`ffplot`	Plot frequency response and spectra
`fpe`	Akaike Final Prediction Error for estimated model

`freqresp`	Frequency-response data from `idmodel` or `idfrd` object
`fselect`	Frequencies from `idfrd` object
`impulse`	Plot impulse response with confidence interval
`isreal`	Determine whether model parameters or data values are real
`ivstruc`	Compute loss functions for sets of output-error model structures
`noisecnv`	Transform `idmodel` object with noise channels to model with measured channels only
`nyquist`	Plot Nyquist curve of frequency response with confidence interval
`pe`	Prediction errors associated with model and data set
`plot`	Plot `iddata` or model objects
`polydata`	Polynomial model parameters from single-input and single-output `idpoly` object
`predict`	Predict output k steps ahead
`predict(idnlarx)`	Predict output k steps ahead for nonlinear ARX model
`predict(idnlgrey)`	Predict output k steps ahead for nonlinear ODE model
`predict(idnlhw)`	Predict output k steps ahead for Hammerstein-Wiener model
`present`	Display model information, including estimated uncertainty
`pzmap`	Plot zeros and poles with confidence interval

`resid`	Compute and test model residuals (prediction errors)
`selstruc`	Select model order for single-output ARX models
`sim`	Simulate linear models with confidence interval
`sim(idnlarx)`	Simulate nonlinear ARX model
`sim(idnlgrey)`	Simulate nonlinear ODE model
`sim(idnlhw)`	Simulate Hammerstein-Wiener model
`simsd`	Simulate models with uncertainty using Monte Carlo method
`ssdata`	State-space matrices from `idmodel` object
`step`	Plot step response with confidence interval
`tfdata`	Numerator and denominator of transfer function from `idmodel` object
`view`	Plot model characteristics using Control System Toolbox™ LTI Viewer GUI
`zpkdata`	Zeros, poles, and gains of transfer function from `idmodel` object

Simulating and Predicting Model Output

`findstates(idmodel)`	Estimate initial states of linear model from data
`findstates(idnlarx)`	Estimate initial states of nonlinear ARX model from data

`findstates(idnlgrey)`	Estimate initial states of nonlinear Grey-Box model from data
`findstates(idnlhw)`	Estimate initial states of nonlinear Hammerstein-Wiener model from data
`idmdlsim`	Simulate `idmodel` objects using Simulink® software
`predict`	Predict output k steps ahead
`predict(idnlarx)`	Predict output k steps ahead for nonlinear ARX model
`predict(idnlgrey)`	Predict output k steps ahead for nonlinear ODE model
`predict(idnlhw)`	Predict output k steps ahead for Hammerstein-Wiener model
`sim`	Simulate linear models with confidence interval
`sim(idnlarx)`	Simulate nonlinear ARX model
`sim(idnlgrey)`	Simulate nonlinear ODE model
`sim(idnlhw)`	Simulate Hammerstein-Wiener model
`simsd`	Simulate models with uncertainty using Monte Carlo method

Using Models with Other Products

`balred`	Reduce model order (requires Control System Toolbox™ product)
`frd`	Convert `idfrd` objects to Control System Toolbox frequency-response LTI model

`LTI Commands`	Apply Control System Toolbox commands to `idmodel` objects
`ss`	Convert `idmodel` objects to Control System Toolbox LTI models
`tf`	Convert `idmodel` objects to transfer-function Control System Toolbox LTI models
`view`	Plot model characteristics using Control System Toolbox LTI Viewer GUI
`zpk`	Convert `idmodel` objects to Control System Toolbox state-space LTI models

Customizing and Using GUI

`ident`	Open System Identification Tool GUI
`midprefs`	Set directory for storing `idprefs.mat` containing GUI startup information

2

Functions – Alphabetical List

addreg

Purpose

Add custom regressors to idnalrx model

Syntax

```
addreg(model,R)
addreg(model,R,I)
addreg(model,R1,I1,R2,I2,Rn,In,)
```

Input

`model`

Name of the idnlarx model object.

`R`

For single-output models, `R` can be an array of `customreg` objects. `R` can also be a cell array of strings, where is string is an expression in terms of input and output variables.

For multiple-output models with `ny` outputs, `R` can be a 1-by-`ny` cell array containing names of `customreg` objects or strings of expressions.

`I`

Scalar integer or vector of integers. Each integer specifies the index of a model output channel.

Description

`addreg(model,R)` adds one or more custom regressors `R` to nonlinear ARX model `model`. For multiple-output systems, each element of the `R` cell array is added to the corresponding output channel of the model.

`addreg(model,R,I)` is used for multiple-output models and adds one or more custom regressors `R` to specific output channels.

`addreg(model,R1,I1,R2,I2,Rn,In,)` is used for multiple-output models and specifies the regressor-channel pairs.

Examples

The following example shows how to add regressors to a nonlinear ARX model using a cell array of strings. `u1` and `y2` are input and output variables, respectively:

```
load iddata1
m1=nlarx(z1,[4 2 1],'wave','nlr',[1:3]);
```

```
% Add regressors using cell array of strings
m2=addreg(m1,{'y1(t-2)^2';'u1(t)*y1(t-7)'})
```

Alternatively, you can use the `customreg` constructor to create regressors, and then add them to the model:

```
r1=customreg(@(x,y)x*y,{'y1','u1'},[2 3])
r2=customreg(@(x,y)x+y,{'y1','u1'},[2 3])
m2=addreg(m1,[r1 r2]);
```

See Also

`customreg`

`getreg`

`nlarx`

`polyreg`

advice

Purpose

Analysis and recommendations for data or estimated linear polynomial and state-space models

Syntax

```
advice(model)
advice(data)
```

Arguments

`model`
: Name of the `idarx`, `idgrey`, `idpoly`, `idproc`, or `idss` model object. These model objects belong to the `idmodel` abstract class, representing linear polynomial and state-space models.

`data`
: Name of the `iddata` object.

Description

`advice(model)` displays the following information about the estimated model in the MATLAB® Command Window:

- Does the model capture essential dynamics of the system and the disturbance characteristics?
- Is the model order higher than necessary?
- Is there potential output feedback in the validation data?
- Would a nonlinear ARX model perform better than a linear ARX model?

`advice(data)` displays the following information about the data in the MATLAB Command Window:

- What are the excitation levels of the signals and how does this affects the model orders? See also `pexcit`.
- Does it make sense to remove constant offsets and linear trends from the data? See also `detrend`.
- Is there an indication of output feedback in the data? See also `feedback`.

See Also

detrend

feedback

iddata

pexcit

Purpose

Akaike Information Criterion for estimated model

Syntax

```
am = aic(model)
am = aic(model1,model2,...)
```

Arguments

`model`

Name of an `idarx`, `idgrey`, `idpoly`, `idproc`, or `idss` model object. These model objects belong to the `idmodel` abstract class.

Description

`am = aic(model)` returns a scalar value of the Akaike's Information Criterion (AIC) for the estimated `model`.

`am = aic(model1,model2,...)` returns a row vector containing AIC values for the estimated models `model1,model2,...`.

Remarks

Use Akaike Information Criterion (AIC) to perform a relative comparison of models with different structures. Smaller value of AIC indicates a better model.

AIC is defined by the following equation:

$$AIC = \log V + \frac{2d}{N}$$

where V is the loss function, d is the number of estimated parameters, and N is the number of values in the estimation data set.

For $d << N$

$$AIC = \log\left(V + \left(1 + \frac{2d}{N} \right) \right)$$

The loss function V is

$$V = \det\left(\frac{1}{N} \sum_{1}^{N} \varepsilon\left(t, \theta_N\right) \left(\varepsilon\left(t, \theta_N\right) \right)^T \right)$$

where $\hat{\theta}_N$ represents the estimated parameters.
AIC is formally defined as the negative log-likelihood function Λ, evaluated at the estimated parameters, plus the number of estimated parameters. You can derive AIC from this definition, as follows:

If the disturbance source is Gaussian with the covariance matrix Λ, the logarithm of the likelihood function is

$$L(\theta, \Lambda) = -\frac{1}{2} \sum_{1}^{N} \varepsilon(t, \theta)^T \Lambda^{-1} \varepsilon(t, \theta) - \frac{N}{2} \log\left(\det \Lambda \right) + const$$

Maximizing this analytically with respect to Λ, and then maximizing the result with respect to θ, gives

$$L(\theta,\Lambda) = const + \frac{Np}{2} + \frac{N}{2}\log(V)$$

where p is the number of outputs.

To obtain the AIC expression from the last result, remove the constants and normalize.

References

Ljung, L. *System Identification: Theory for the User*, Upper Saddle River, NJ, Prentice-Hal PTR, 1999. See sections about the statistical framework for parameter estimation and maximum likelihood method and comparing model structures.

See Also

`EstimationInfo`

`fpe`

Purpose

Algorithm properties affecting estimation process for linear models

Syntax

```
idprops idmodel algorithm
Model.algorithm.PropertyName='PropertyValue'
```

Description

`Algorithm` is a property of the `idmodel` class that specifies the estimation algorithm. The `idmodel` subclasses are used to define various linear models, including `idarx`, `idss`, `idpoly`, `idproc`, and `idgrey`. These models inherit the `idmodel` `Algorithm` property.

Property names are not case sensitive. When you type a property name, you only need to enter enough characters to uniquely identify the property. The Algorithm fields can be accessed and modified as for any structure using dot syntax. For example, you can access the `SearchMethod` field by typing `Model.Algorithm.SearchMethod`.

Note You can use the `get` function or dot notation to fetch fields of `Algorithm` as if they were the properties of the model itself. Similarly you can use `set` or dot notation to set a particular field. This shortcut access is available for the fields of `Algorithm` only (not for deeper level struct fields such as Search or Threshold options). For example:

```
method = Model.SearchMethod;
Model.MaxIter = 100;
```

is equivalent to

```
get(Model, 'SearchMethod')
set(Model, 'maxiter', 100);
```

When you create a new model by refining an existing model `m`, the algorithm properties of `m` are inherited by the new model.

Note You can estimate a model with specific algorithm settings and define a structure variable to store the algorithm values. For example:

```
model = n4sid(data,order)
myalg = model.Algorithm;
myalg.Focus='Simulation';
m = pem(data,model,'alg',myalg)
```

You can also specify the algorithm properties (except advanced properties) as property-value pairs when creating the linear model (using `idpoly`, `idss`, etc.) or when estimating them (using `pem`, `n4sid`, `armax`, `oe` etc.).

Algorithm Properties

- `Criterion`: Specifies criterion used during minimization. `Criterion` can have the following values:
  - `'Det'`: Minimize $\det(E'*E)$, where E represents the prediction error. This is the optimal choice in a statistical sense and leads to the maximum likelihood estimates in case nothing is known about the variance of the noise. It uses the inverse of the estimated noise variance as the weighting function. This is the default criterion used for all models, except `idnlgrey` which uses `'Trace'` by default.
  - `'Trace'`: Minimize the trace of the weighted prediction error matrix `trace(E'*E*W)`, where `E` is the matrix of prediction errors, with one column for each output, and `W` is a positive semi-definite symmetric matrix of size equal to the number of outputs. By default, `W` is an identity matrix of size equal to the number of model outputs (so the minimization criterion becomes `trace(E'*E)`, or the traditional least-sum-of-squared-errors criterion. You can specify the relative weighting of prediction errors for each output using the `Weighting` field of the Algorithm property.

Note The difference between the two criteria is meaningful in multiple-output cases only. In single-output models, the two criteria are equivalent. Both the `Det` and `Trace` criteria are derived from a general requirement of minimizing a weighted sum of squares of prediction errors. The `Det` criterion can be interpreted as estimating the covariance matrix of the noise source and using the inverse of that matrix as the weighting. When using the `Trace` criterion, you must specify the weighting using the `Weighting` property.

If you want to achieve better accuracy for a particular channel in multiple-input multiple-output models, you should use `Trace` with weighting that favors that channel. Otherwise it is natural to use the `Det` criterion. When using `Det`, you can check `cond(model.NoiseVariance)` after estimation. If the matrix is ill-conditioned, it may be more robust to use the `Trace` criterion. You can also use `compare` on validation data to check whether the relative error for different channels corresponds you your needs or expectations. Use the `Trace` criterion if you need to modify the relative errors, and check `model.NoiseVariance` to determine what weighting modifications to specify.

The search method of `lsqnonlin` supports the `Trace` criterion only.

- `Focus`: Defines how the errors e between the measured and the modeled outputs are weighed at specific frequencies during the minimization of the following loss function:

$$V = \sum_{i=1}^{N} \lambda_i e_i^2$$

Higher weighting at specific frequencies emphasizes the requirement for a good fit at these frequencies. `Focus` can have the following values:

  - `'Prediction'`: (Default) Automatically calculates the weighting function as a product of the input spectrum and the inverse of the

noise model. This minimizes the one-step-ahead prediction, which typically favors fitting small time intervals (higher frequency range). From a statistical-variance point of view, this is the optimal weighting function. However, this method neglects the approximation aspects (bias) of the fit. Might not result in a stable model. Use `'Stability'` when you want to ensure a stable model.

- `'Simulation'`: Estimates the model using the frequency weighting of the transfer function that is given by the input spectrum. Typically, this method favors the frequency range where the input spectrum has the most power. In other words, the resulting model will produce good simulations for inputs that have the same spectra as used for estimation. For models that have no disturbance model, there is no difference between `'Simulation'` and `'Prediction'`. In this case, $y = Gu + He$ with H=1, which is equivalent to `A=C=D=1` for `idpoly` models and `K = 0` for `idss` models.

  For models that have a disturbance model, G is first estimated with H=1, and then H is estimated using a prediction-error method with a fixed estimated transfer function $\hat{G}$. This guarantees a stable transfer function G.

- `'Stability'`: This weighting is the same as for `'Prediction'`, but the model is forced to be stable. Use only when you know the system is stable. In some cases, forcing the model to be stable can result in a bad model.

- Enter a row vector or a matrix containing frequency values that define desired passbands. For example:

  ```
  [wl,wh]
  [w1l,w1h;w2l,w2h;w3l,w3h;...]
  ```

  where `wl` and `wh` represent upper and lower limits of a passband. For a matrix with several rows defining frequency passbands, the algorithm uses union of frequency ranges to define the estimation passband.

- Enter any SISO linear filter in any of the following ways:

  A single-input-single-output (SISO) `idmodel` object.

  An `ss`, `tf`, or `zpk` model from the Control System Toolbox™ product.

  Using the format `{A,B,C,D}`, which specifies the state-space matrices of the filter.

  Using the format `{numerator, denominator}`, which specifies the numerator and denominator of the filter transfer function

  This calculates the weighting function as a product of the filter and the input spectrum to estimate the transfer function from input to output, *G*. To obtain a good model fit for a specific frequency range, you must choose the filter with a passband in this range. After estimating *G*, the algorithm computes the disturbance model using a prediction-error method and keeping the estimated transfer function fixed (similar to the `'Simulation'` case). For a model that has no disturbance model, the estimation result is the same if you first prefilter the data using `idfilt`.

- For frequency-domain data only, enter a column vector of weights for `'Focus'`. This vector must have the same size as length of the frequency vector of the data set, `Data.Frequency`. Each input and output response in the data is multiplied by the corresponding weight at that frequency.

- `Maxsize`: `MaxSize`: A positive integer, specified such that the input-output data is split into segments where each contains fewer than `MaxSize` elements. Setting `MaxSize` can improve computational performance. The default value of `MaxSize` is `'Auto'`, which uses the M-file `idmsize` to set the value. You can edit this file to optimize computational speed on a particular computer. `MaxSize` does not affect the numerical properties of the estimate except when you use `InitialState = 'backcast'` for frequency-domain data. In this case, the frequency ranges where backcasting takes place might depend on `MaxSize` and affects estimates.

- `FixedParameter`: Vector of integers containing the indices of parameters that are not estimated and remain fixed at nominal or initial values. Parameter indices refer to their position in the list, stored in the property `'ParameterVector'`. You can also specify parameter names as values from the property `'PName'`. To specify fixed parameters using parameter names, enter `Fixedparameter` as a cell array of strings. For example, to fix parameters with names `'a'` and `'b'`, type `m.FixedParameter = {'a','b','c'}`. Strings can contain wildcards, such as `'*'` to specify the automatic completion of a string, or `'?'` to represent an arbitrary character. For example, to fix three parameters in a disturbance model that start with `'k'`, such as `'k1'`, `'k2'`,`'k3'`, use `FixedParameter = {'k*'}`. For structured state-space models, you can fix and free parameters by specifying structure matrices, such as `As` and `Bs` (see `idss`).

  Note By default, the property `'PName'` is empty. Use `setpname` to assign default parameter names. For example, `m = setpname(m)`.

- `Weighting`: Positive semi-definite symmetric matrix `W` to use as weighting for minimization of the trace criterion `trace(E'*E*W)`. `Weighting` can be used to specify relative importance of outputs in multiple-input multiple-output models (or reliability of corresponding data) by specifying `W` as a diagonal matrix of non-negative values. `Weighting` is not useful in single-input single-output models. By default, `Weighting` is the identity matrix of size equal to the number of model outputs, assigning equal importance to each output during estimation.
- `Trace`: Specifies what information displays in the MATLAB® Command Window about the iterative search during estimation.
  - `'Trace'='Off'`: Displays no information.
  - `'Trace'='On'`: Displays the loss-function values for each iteration.
  - `'Trace'='Full'`: Displays the same information as `'On'` and also include the current parameter values and the search direction

(except when the `Advanced SSParameterization` model property is set to `'Free'` for `idss` models and the list of parameters can change between iterations).

- `LimitError`: Specifies when to adjust the weight of large errors from quadratic to linear. Default value is `0`. Errors larger than `LimitError` times the estimated standard deviation have a linear weight in the criteria. The standard deviation is estimated robustly as the median of the absolute deviations from the median and divided by `0.7`. (See the section about choosing a robust norm in [2].) `LimitError = 0` disables the robustification and leads to a purely quadratic criterion. When estimating with frequency-domain data, `LimitError` is set to zero.

  Note You can estimate the model with the default value of `LimitError` (zero) and plot the prediction errors using `pe(data.model)`. If the resulting plot shows occasional large values, repeat the estimation with `model.Algorithm.LimitError` set to a value between `1` and `2`.

- `MaxIter`: Specifies the maximum number of iterations during loss-function minimization. The iterations stop when `MaxIter` is reached or another stopping criterion is satisfied, such as the `Tolerance`. The default value of `MaxIter` is `20`. Setting `MaxIter = 0` returns the result of the startup procedure. Use `EstimationInfo.Iterations` to get the actual number of iterations during an estimation.
- `Tolerance`: Specifies the minimum percentage difference (divided by 100) between the current value of the loss function and its expected improvement after the next iteration: When the percentage of expected improvement is less than `Tolerance`, the iterations are stopped. Default value is `0.01`. The estimate of the expected loss-function improvement at the next iteration is made based on the Gauss-Newton vector computed for the current parameter value.

- `SearchMethod`: The search method used for iterative parameter estimation. It can take one of the following values:
  - `'gn'`: The subspace Gauss-Newton direction. Singular values of the Jacobian matrix less than `GnPinvConst*eps*max(size(J))*norm(J)` are discarded when computing the search direction, where `J` is the Jacobian matrix. The Hessian matrix is approximated by J^TJ. If there is no improvement along this direction, the gradient direction is also tried.
  - `'gna'`: An adaptive version of subspace Gauss-Newton approach, suggested by Wills and Ninness (IFAC World congress, Prague 2005). Eigenvalues less than `gamma*max(sv)` of the Hessian are neglected , where `sv` are the singular values of the Hessian. The Gauss-Newton direction is computed in the remaining subspace. `gamma` has the initial value `InitGnaTol` (see below) and is increased by the factor `LmStep` each time the search fails to find a lower value of the criterion in less than 5 bisections. It is decreased by the factor `2*LmStep` each time a search is successful without any bisections.
  - `'lm'`: Uses the Levenberg-Marquardt method. This means that the next parameter value is `-pinv(H+d*I)*grad` from the previous one, where `H` is the Hessian, `I` is the identity matrix, and `grad` is the gradient. `d` is a number that is increased until a lower value of the criterion is found.
  - `'Auto'`: A choice among the above is made in the algorithm. This is the default choice.
  - `'lsqnonlin'`: Uses `lsqnonlin` optimizer from Optimization Toolbox software. You must have Optimization Toolbox installed to use this option. This search method can only handle the `Trace` criterion.
- `Advanced`: Structure that specifies advanced algorithm options and has the following fields:

- `Search`: Uses the following fields to specify options for the iterative search:

a `GnPinvConst`: Must be a positive real value. Specifies that singular values of the Jacobian that are smaller than `GnPinvConst*max(size(J)*norm(J)*eps` are discarded when computing the search direction using the `'gn'` method. Default value is `1e4`.

b `InitGnaTol`: The initial value of `gamma` in the `gna` search algorithm. See `SearchMethod` for description of `gna`. Default is `10^-4`.

c `LmStep`: The size of the Levenberg-Marquardt step. The next value of the search-direction length `d` in the Levenberg-Marquardt method is `LmStep` times the previous one. Default is `2`.

d `StepReduction`: For search directions other than the Levenberg-Marquardt direction, the step is reduced by the factor `StepReduction` after each iteration. Default is `2`.

e `MaxBisection`: The maximum number of bisections used by the line search along the search direction. Default is `25`.

f `LmStartValue`: The starting value of search-direction length `d` in the Levenberg-Marquardt method. Default is `0.001`.

g `RelImprovement`: The iterations are stopped if the relative improvement of the criterion is less than `RelImprovement`. Default is `RelImprovement = 0`. This property is different from `Tolerance` in that it uses the actual improvement of the loss function, as opposed to the expected improvement.

- `Threshold`: Contains fields with thresholds for several tests:

a `Sstability`: Specifies the location of the rightmost pole to test the stability of continuous-time models. A model is considered stable when its rightmost pole is to the left of `Sstability`. Default is `0`.

b `Zstability`: Specifies the maximum distance of all poles from the origin to test stability of discrete-time models. A model is considered stable if all poles are within the distance `Zstability` from the origin. Default is `1+sqrt(eps)`.

- `AutoInitialState`: Specifies when to automatically estimate the initial state. When `InitialState = 'Auto'`, the initial state is estimated when the ratio of the prediction-error norm with a zero initial state to the norm with an estimated initial state exceeds `AutoInitialState`. Default is `1.05`.

Properties Relevant to Estimation of n4sid, State-Space (idss) Models

---

Note These properties apply to `n4sid`. Since `pem` commonly uses `n4sid` to initialize a model for iterative estimation, these properties affect the results of `pem` too.

- `N4Weight`: Calculates the weighting matrices used in the singular-value decomposition step of the algorithm and has three possible values:
  - `'Auto'`: (Default) Automatically chooses between `'MOESP'` and `'CVA'` .
  - `'MOESP'`: Uses the MOESP algorithm by Verhaegen.
  - `'CVA'`: Uses the canonical variable algorithm by Larimore.

  For more information about setting this property, see the `n4sid` reference page.
- `N4Horizon`: Determines the forward and backward prediction horizons used by the algorithm. Enter a row vector with three elements: `N4Horizon=[r sy su]`, where `r` is the maximum forward prediction horizon; that is, the algorithm uses up to `r` step-ahead predictors. `sy` is the number of past outputs, and `su` is the number of past inputs used for predictions. For an exact definition of these integers, see the section about subspace methods in [2], where they

are called *r*, *s1*, and *s2*. These numbers can have a substantial influence on the quality of the resulting model and there are no simple rules for choosing them. Making `'N4Horizon'` a `k`-by-`3` matrix means that the algorithm tries each row of `'N4Horizon'` and selects the value that gives the best (prediction) fit to the data. Choosing the best row is not available when you also specify to select the best model order. When you specify one column in `'N4Horizon'`, the interpretation is `r=sy=su`. The default choice is `'N4Horizon' = 'Auto'`, which uses an Akaike Information Criterion (AIC) for the selection of `sy` and `su`.

Note For algorithm properties of nonlinear models, see the reference pages for `idnlarx`, `idnlhw`, and `idnlgrey`.

References

[1] Dennis, J.E., Jr., and R.B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice Hall, Englewood Cliffs, N.J., 1983. See the chapter about iterative minimization.

[2] Ljung, L. *System Identification: Theory for the User*, Upper Saddle River, NJ, Prentice-Hal PTR, 1999. See the chapter about computing the estimate.

See Also

`armax`

`bj`

`EstimationInfo`

`idpoly`

`idss`

`n4sid`

`oe`

`pem`

Purpose

Estimate parameters of AR model for scalar time series returning `idpoly` object

Syntax

```
m = ar(y,n)
[m,refl] = ar(y,n,approach,window)
[m,refl] = ar(y,n,approach,window,'P1',V1,...,'PN',VN)
```

Arguments

`y`

`iddata` object that contains the time-series data (one output channel).

`n`

Scalar that specifies the order of the model you want to estimate (the number of *A* parameters in the AR model).

`approach`

Lets you choose the algorithm for computing the least squares AR model from the following options:

- `'burg'`: Burg's lattice-based method. Solves the lattice filter equations using the harmonic mean of forward and backward squared prediction errors.
- `'fb'`: (Default) Forward-backward approach. Minimizes the sum of a least- squares criterion for a forward model, and the analogous criterion for a time-reversed model.
- `'gl'`: Geometric lattice approach. Similar to Burg's method, but uses the geometric mean instead of the harmonic mean during minimization.
- `'ls'`: Least-squares approach. Minimizes the standard sum of squared forward-prediction errors.
- `'yw'`: Yule-Walker approach. Solves the Yule-Walker equations, formed from sample covariances.

`window`

Lets you specify how to use information about the data outside the measured time interval (past and future values). The following windowing options are available:

- `'now'`: (Default) No windowing. This value is the default except when the `approach` argument is `'yw'`. Only measured data is used to form regression vectors. The summation in the criteria starts at the sample index equal to `n+1`.
- `'pow'`: Postwindowing. Missing end values are replaced with zeros and the summation is extended to time `N+n` (`N` is the number of observations).
- `'ppw'`: Pre- and postwindowing. Used in the Yule-Walker approach.
- `'prw'`: Prewindowing. Missing past values are replaced with zeros so that the summation in the criteria can start at time equal to zero.

`'P1',V1,...,'PN',VN`

Pairs of property names and property values can include any of the following.

Property Name	Property Value	Description
`'Covariance'`	• `'None'` Suppresses the calculation of the covariance matrix. • `[]` Empty. • Square matrix containing covariance values of size equal to the length of the parameter vector	Specifies calculation of uncertainties in parameter estimates.

Property Name	Property Value	Description
'MaxSize'	Integer	See Algorithm Properties for the description.
'Ts'	Real positive number	Sets the sampling time and overrides the sampling time of y.

Description

Note Use for scalar time series only. For multivariate data, use arx.

m = ar(y,n) returns an idpoly model m.

[m,refl] = ar(y,n,approach,window) returns an idpoly model m and the variable refl. For the two lattice-based approaches, 'burg' and 'gl', refl stores the reflection coefficients in the first row, and the corresponding loss function values in the second row. The first column of refl is the zeroth-order model, and the (2,1) element of refl is the norm of the time series itself.

[m,refl] = ar(y,n,approach,window,'P1',V1,...,'PN',VN) returns an idpoly model m and the variable refl using additional windowing criteria.

Remarks

The AR model structure is given by the following equation:

$$A(q)y(t) = e(t)$$

AR model parameters are estimated using variants of the least-squares method. The following table summarizes the common names for methods with a specific combination of approach and window argument values.

Method	Combination of approach and window values
Modified Covariance Method	(Default) Forward-backward approach and no windowing.
Correlation Method	Yule-Walker approach, which corresponding to least squares plus pre- and postwindowing.
Covariance Method	Least squares approach with no windowing. `arx` uses this routine.

Examples

Given a sinusoidal signal with noise, compare the spectral estimates of Burg's method with those found from the forward-backward approach and no-windowing method on a Bode plot.

```
y = sin([1:300]') + 0.5*randn(300,1);
y = iddata(y);
mb = ar(y,4,'burg');
mfb = ar(y,4);
bode(mb,mfb)
```

References

Marple, Jr., S.L., *Digital Spectral Analysis with Applications*, Prentice Hall, Englewood Cliffs, 1987, Chapter 8.

See Also

`Algorithm Properties`

`arx`

`EstimationInfo`

`etfe`

`idpoly`

`ivar`

`pem`

spa

step

Purpose

Estimate parameters of ARMAX or ARMA model returning `idpoly` object

Syntax

```
m = armax(data,orders)
m = armax(data,orders,'P1',V1,...,'PN',VN)
m = armax(data,'na',na,'nb',nb,'nc',nc,'nk',nk)
```

Arguments

`data`

`iddata` object that contains the input-output data.

`orders`

Vector of integers, specified using the format

```
orders = [na nb nc nk]
```

For multiple-input systems, `nb` and `nk` are row vectors where the `i`th element corresponds to the order and delay associated with the `i`th input.

When `data` is a time series, which has no input and one output, then

```
orders = [na nc]
```

Note When refining an estimated model `mi`, set the model orders as follows:

```
orders = mi
```

`'na',na,'nb',nb,'nc',nc,'nk',nk`

`'na'`, `'nb'`, and `'nc'` are orders of the ARMAX model. `nk` is the delay. `na`, `nb`, `nc`, and `nk` are the corresponding integer values.

`'P1',V1,...,'PN',VN`
: Pairs of property names and property values can include any of the following `idmodel` properties:

  `'Focus'`, `'InitialState'`, `'Trace'`, `'MaxIter'`, `'Tolerance'`, `'LimitError'`, and `'FixedParameter'`.

  See `Algorithm Properties`, `idpoly`, and `idmodel` for more information.

Description

Note `armax` only supports time-domain data with single or multiple inputs and single output. For frequency-domain data, use `oe`. For the multiple-output case, use ARX or a state-space model (see `n4sid` and `pem`).

`m = armax(data,orders)` returns an `idpoly` model `m` with estimated parameters and covariances (parameter uncertainties). Estimates the parameters using the prediction-error method and specified `orders`.

`m = armax(data,orders,'P1',V1,...,'PN',VN)` returns an `idpoly` model `m`. Use additional property-value pairs to specify the estimation algorithm properties.

`m = armax(data,'na',na,'nb',nb,'nc',nc,'nk',nk)` returns an `idpoly` model `m` with orders and delays specified as parameter-value pairs.

Remarks

The ARMAX model structure is

$$y(t) + a_1 y(t-1) + \ldots + a_{n_a} y(t-n_a) =$$
$$b_1 u(t-n_k) + \ldots + b_{n_b} u(t-n_k-n_b+1) +$$
$$c_1 e(t-1) + \ldots + c_{n_c} e(t-n_c) + e(t)$$

A more compact way to write the difference equation is

$$A(q)y(t) = B(q)u(t - n_k) + C(q)e(t)$$

where

- $y(t)$ — Output at time t.
- n_a — Number of poles.
- n_b — Number of zeroes plus 1.
- n_c — Number of C coefficients.
- n_k — Number of input samples that occur before the input affects the output, also called the *dead time* in the system. For discrete systems with no dead time, there is a minimum 1–sample delay because the output depends on the previous input and $n_k = 1$.
- $y(t-1)\ldots y(t-n_a)$ — Previous outputs on which the current output depends.
- $u(t-n_k)\ldots u(t-n_k-n_b+1)$ — Previous and delayed inputs on which the current output depends.
- $e(t-1)\ldots e(t-n_c)$ — White-noise disturbance value.

The parameters `na`, `nb`, and `nc` are the orders of the ARMAX model, and `nk` is the delay. q is the delay operator. Specifically,

$$A(q) = 1 + a_1 q^{-1} + \ldots + a_{n_a} q^{-n_a}$$

$$B(q) = b_1 + b_2 q^{-1} + \ldots + b_{n_b} q^{-n_b+1}$$

$$C(q) = 1 + c_1 q^{-1} + \ldots + c_{n_c} q^{-n_c}$$

If `data` is a time series, which has no input channels and one output channel, then `armax` calculates an ARMA model for the time series

$$A(q)y(t) = e(t)$$

In this case

```
orders = [na nc]
```

Algorithm

An iterative search algorithm with the properties `'SearchMethod'`, `'MaxIter'`, `'Tolerance'`, and `'Advanced'` minimizes a robustified quadratic prediction error criterion is minimized. The iterations are terminated either when `MaxIter` is reached, or when the expected improvement is less than `Tolerance`, or when a lower value of the criterion cannot be found. You can get information about the search criteria using `m.EstimationInfo`.

When you do not specify initial parameter values for the iterative search in `orders`, they are constructed in a special four-stage LS-IV algorithm.

The cutoff value for the robustification is based on the property `LimitError` and on the estimated standard deviation of the residuals from the initial parameter estimate. It is not recalculated during the minimization.

To ensure that only models corresponding to stable predictors are tested, the algorithm performs a stability test of the predictor. Generally, both

$C(q)$ and $F(q)$ (if applicable) must have all zeros inside the unit circle.

Minimization information is displayed on the screen when the property `'Trace'` is `'On'` or `'Full'`. With `'Trace' ='Full'`, both the current and the previous parameter estimates are displayed in column-vector form, listing parameters in alphabetical order. Also, the values of the criterion function are given and the Gauss-Newton vector and its norm are also displayed. With `'Trace' = 'On'` only the criterion values are displayed.

References

Ljung, L. *System Identification: Theory for the User*, Upper Saddle River, NJ, Prentice-Hal PTR, 1999. See chapter about computing the estimate.

See Also

`Algorithm Properties`

`EstimationInfo`

`idpoly`

`pem`

Purpose

Estimate parameters of ARX or AR model using least squares returning `idpoly` or `idarx` object

Syntax

```
m = arx(data,orders)
m = arx(data,orders,'P1',V1,...,'PN',VN)
m = arx(data,'na',na,'nb',nb,'nc',nc,'nk',nk)
```

Arguments

`data`

An `iddata` object, an `frd` object, or an `idfrd` frequency-response-data object.

`orders`

Vector of integers, specified using the format

```
orders = [na nb nk]
```

For multiple-input systems, `nb` and `nk` are row vectors where the `i`th element corresponds to the order and delay associated with the `i`th input.

When `data` is a time series, which has no input and one output, then

```
orders = [na]
```

Note When refining an estimated model `mi`, set the model orders as follows:

```
orders = mi
```

`'na',na,'nb',nb,'nc',nc,'nk',nk`

`'na'`, `'nb'`, and `'nc'` are orders of the ARMAX model. `nk` is the delay. `na`, `nb`, `nc`, and `nk` are the corresponding integer values.

`'P1',V1,...,'PN',VN`

Pairs of property names and property values can include any of the following `idmodel` properties:

`'Focus'`, `'InitialState'`, `'Trace'`, `'MaxIter'`, `'Tolerance'`, `'LimitError'`, and `'FixedParameter'`.

See `Algorithm Properties`, `idpoly`, and `idmodel` for more information.

Description

Note `arx` does not support multiple-output continuous-time models. Use state-space model structure instead. When the true noise term $e(t-1)\ldots e(t-n_c)$ in the ARX model structure is not white noise and `na` is nonzero, the model estimate is incorrect. In this case, use `armax`, `bj`, `iv4`, or `oe`.

`m = arx(data,orders)` returns a model `m` with estimated parameters and covariances (parameter uncertainties). For single-output data, the model is an `idpoly` object. For multiple-output data, the model is an `idarx` object. Uses the least-squares method and specified `orders`.

`m = arx(data,orders,'P1',V1,...,'PN',VN)` returns a model `m`. Use additional property-value pairs to specify the estimation algorithm properties.

`m = arx(data,'na',na,'nb',nb,'nc',nc,'nk',nk)` returns a model `m` with orders and delays specified as parameter-value pairs.

Remarks

`arx` the parameters of the ARX model structure:

$$y(t)+a_1y(t-1)+\ldots+a_{n_a}y(t-n_a)=$$
$$b_1u(t-1)+\ldots+b_{n_b}u(t-n_k-n_b+1)+e(t)$$

The parameters `na` and `nb` are the orders of the ARX model, and `nk` is the delay.

- $y(t)$ — Output at time t.
- n_a — Number of poles.
- n_b — Number of zeroes plus 1.
- n_k — Number of input samples that occur before the input affects the output, also called the *dead time* in the system. For discrete systems with no dead time, there is a minimum 1–sample delay because the output depends on the previous input and $n_k = 1$.
- $y(t-1)\ldots y(t-n_a)$ — Previous outputs on which the current output depends.
- $u(t-n_k)\ldots u(t-n_k-n_b+1)$ — Previous and delayed inputs on which the current output depends.
- $e(t-1)\ldots e(t-n_c)$ — White-noise disturbance value.

A more compact way to write the difference equation is

$$A(q)y(t) = B(q)u(t-n_k) + e(t)$$

q is the delay operator. Specifically,

$$A(q) = 1 + a_1 q^{-1} + \ldots + a_{n_a} q^{-n_a}$$

$$B(q) = b_1 + b_2 q^{-1} + \ldots + b_{n_b} q^{-n_b+1}$$

Time Series Models

For time-series data that contains no inputs, one output and `orders = na`, the model has AR structure of order `na`.

The AR model structure is

$$A(q)y(t) = e(t)$$

Multiple Inputs and Single-Output Models

For multiple-input systems, `nb` and `nk` are row vectors where the `i`th element corresponds to the order and delay associated with the `i`th input.

$$A(q)y(t) = B_1(q)u_1(t - n_{k1}) + \ldots + B_{nu}(q)u_{nu}(t - n_{k_{nu}}) + e(t)$$

Multioutput Models

For models with multiple inputs and multiple outputs, `na`, `nb`, and `nk` contain one row for each output signal.

In the multiple-output case, `arx` minimizes the trace of the prediction error covariance matrix, or the norm

$$\sum_{t=1}^{N} e^T(t)e(t)$$

To transform this to an arbitrary quadratic norm using a weighting matrix `Lambda`

$$\sum_{t=1}^{N} e^T(t)\Lambda^{-1}e(t)$$

use the syntax

```
m = arx(data,orders,'NoiseVariance', Lambda)
```

You can use `arx` to refine an existing model `m_initial` as an argument.

```
m = arx(data,m_initial)
```

The new model m uses the orders and the weighting matrix for the prediction errors from `m_initial`. You can further modify `m_initial` by adding a list of property name and value pairs as arguments.

This is especially useful when some parameters must be fixed using `'FixedParameter'` property.

Continuous-Time Models

For models with one output, continuous-time models can be estimated from continuous-time frequency-domain data. In this case, `na` is the number of estimated denominator coefficients and `nb` is number of estimated numerator coefficients.

Note For continuous-time models, omit the delay parameter `nk` because it has no meaning in this case. Because estimating continuous-time ARX models often produces bias, you might get better results by using the `oe` method.

For example, when `na = 4`, `nb = 2`, the model structure is:

$$G(s) = \frac{b_1 s + b_2}{s^4 + a_1 s^3 + a_2 s^2 + a_3 s + a_4}$$

Tip When using continuous-time data, limit the fit to a smaller frequency range using the `'Focus'` `idmodel` property:

```
m = arx(datac,[na nb],'focus',[0 wh])
```

Estimating Initial Conditions

For time-domain data, the signals are shifted such that unmeasured signals are never required in the predictors. Therefore, there is no need to estimate initial conditions.

For frequency-domain data, it might be necessary to adjust the data by initial conditions that support circular convolution.

You can set the property 'InitialState' to one of the following values:

- 'zero' — No adjustment.
- 'estimate' — Perform adjustment to the data by initial conditions that support circular convolution.
- 'auto' — Automatically choose between 'zero' and 'estimate' based on the data.

See Algorithm Properties for more information on model properties.

Algorithm

QR factorization solves the overdetermined set of linear equations that constitutes the least-squares estimation problem.

The regression matrix is formed so that only measured quantities are used (no fill-out with zeros). When the regression matrix is larger than MaxSize, data is segmented and QR factorization is performed iteratively on these data segments.

Examples

This example generates input data based on a specified ARX model, and then uses this data to estimate an ARX model.

```
A = [1  -1.5  0.7]; B = [0 1 0.5];
m0 = idpoly(A,B);
u = iddata([],idinput(300,'rbs'));
e = iddata([],randn(300,1));
y = sim(m0, [u e]);
z = [y,u];
m = arx(z,[2 2 1]);
```

See Also

Algorithm Properties

EstimationInfo

ar

idarx

idpoly

iv4

ivar

ivx

pem

Purpose

ARX parameters from multiple-output `idarx` or single-output `idpoly` objects with variance information

Syntax

```
[A,B] = arxdata(m)
[A,B,dA,dB] = arxdata(m)
```

Arguments

`model`

Name of the `idarx` model object, which belongs to the `idmodel` abstract class.

Note Also accepts `idpoly` models with an underlying ARX structure with orders `nc=nd=nf=0`.

Description

`[A,B] = arxdata(m)` returns `A` and `B` as 3–D arrays.

Suppose *ny* is the number of outputs (the dimension of the vector *y(t)*) and *nu* is the number of inputs.

`A` is an `ny-by-ny-by-(na+1)` array such that

```
A(:,:,k+1) = Ak
A(:,:,1) = eye(ny)
```

where `k=0,1,...,na`.

`B` is an `ny-by-nu-by-(nb+1)` array with

```
B(:,:,k+1) = Bk
```

`A(0)` is always the identity matrix. The leading entries in `B` equal to zero, which means there are no delays in the model.

Note For a time series, `B = []`.

`[A,B,dA,dB] = arxdata(m)` returns `A` and `B` matrices, and `dA` and `dB` as the estimated standard deviations of `A` and `B`, respectively.

Remarks

A and B are 2–D or 3–D arrays and are returned in the standard multivariable ARX format (see `idarx`), describing the model.

$$A(q)y(t) = B_1(q)u_1(t-n_{k1}) + \ldots + B_{nu}(q)u_{nu}(t-n_{k_{nu}}) + e(t)$$

$$y(t) + A_1 y(t-1) + A_2 y(t-2) + \ldots + A_{na} y(t-na) =$$

$$B_0 u(t) + B_1 u(t-1) + \ldots + B_{nb} u(t-nb) + e(t)$$

Here A_k and B_k

`A` and `B` matrices, which have dimensions *ny*-by-*ny* and *ny*-by-*nu*, respectively. Here, *ny* is the number of outputs (the dimension of the vector *y(t)*) and *nu* is the number of inputs.

See Also

`idarx`

`idpoly`

Purpose

Compute and compare loss functions for single-output ARX models

Syntax

```
V = arxstruc(ze,zv,NN)
V = arxstruc(ze,zv,NN,maxsize)
```

Arguments

`ze`
: Estimation data set can be `iddata` or `idfrd` object.

`zv`
: Validation data set can be `iddata` or `idfrd` object.

`NN`
: Matrix defines the number of different ARX-model structures. Each row of `NN` is of the form:

```
nn = [na nb nk]
```

`maxsize`
: Specified maximum data size. See `Algorithm Properties` for an explanation.

Description

Note Use `arxstruc` for single-output systems only. `arxstruc` supports both single-input and multiple-input systems.

`V = arxstruc(ze,zv,NN)` returns `V`, which contains the loss functions in its first row. The remaining rows of `V` contain the transpose of `NN`, so that the orders and delays are given just below the corresponding loss functions. The last column of `V` contains the number of data points in `ze`.

`V = arxstruc(ze,zv,NN,maxsize)` uses the additional specification of the maximum data size.

with the same interpretation as described for `arx`. See `struc` for easy generation of typical `NN` matrices for single-input systems.

The output argument V is best analyzed using selstruc. The selection of a suitable model structure based on the information in v is normally done using selstruc.

Remarks

Each of ze and zv is an iddata object containing output-input data. Frequency-domain data and idfrd objects are also supported. Models for each of the model structures defined by NN are estimated using the data set ze. The loss functions (normalized sum of squared prediction errors) are then computed for these models when applied to the validation data set zv. The data sets ze and zv need not be of equal size. They could, however, be the same sets, in which case the computation is faster.

Examples

Compare first- and fifth-order models with one delay using cross-validation on the second half of the data set. Then, select the order that gives the best fit to the validation data set.

```
NN = struc(1:5,1:5,1);
V = arxstruc(z(1:200),z(201:400),NN);
nn = selstruc(V,0);
m = arx(z,nn);
```

See Also

Algorithm Properties

arx

idpoly

ivstruc

selstruc

struc

Purpose

Reduce model order (requires Control System Toolbox™ product)

Syntax

```
MRED = balred(M)
MRED = balred(M,ORDER,'DisturbanceModel','None')
```

Description

This function reduces the order of any model `M` given as an `idmodel` object. The resulting reduced-order model, `MRED`, is an `idss` model.

The function requires routines from the Control System Toolbox product.

`ORDER`: The desired order (dimension of the state-space representation). If `ORDER = []`, which is the default, a plot shows how the diagonal elements of the observability and controllability Gramians of a balanced realization decay with the order of the representation. You are then prompted to select an order based on this plot. The idea is that such a small element has a negligible influence on the input-output behavior of the model. We recommend that you choose an order such that only large elements in these matrices are retained.

`'DisturbanceModel'`: If the property `DisturbanceModel` is set to `'None'`, then an output-error model `MRED` is produced: that is, one with the Kalman gain K equal to zero. Otherwise (default), the disturbance model is also reduced.

The function recognizes whether `M` is a continuous- or discrete-time model and performs the reduction accordingly. The resulting model, `MRED`, is similar to `M` in this respect.

There are several options for how the reduction is performed: `AbsTol, RelTol, Offset, Elimination`.

Algorithm

The function uses the `balred` algorithm in Control System Toolbox. The plot, in case `ORDER = []`, shows the vector `g` returned by `balreal`.

Examples

Build a high-order multivariable ARX model, reduce its order to 3, and compare the frequency responses of the original and reduced models:

```
M = arx(data,[4*ones(3,3),4*ones(3,2),ones(3,2)]);
MRED = balred(M,3);
```

```
bode(M,MRED)
```

Use the reduced-order model as an initial condition for a third-order state-space model.

```
M2 = pem(data,MRED);
```

See Also

balreal

Purpose

Estimate parameters of Box-Jenkins model returning `idpoly` object

Syntax

```
m = bj(data,orders)
m = bj(data,'nb',nb,'nc',nc,'nd',nd,'nf',nf,'nk',nk)
m = bj(data,orders,'Property1',Value1,'Property2',Value2,...)
```

Description

`bj` returns `m` as an `idpoly` object with the resulting parameter estimates, together with estimated covariances. The `bj` function estimates parameters of the Box-Jenkins model structure

$$y(t) = \frac{B(q)}{F(q)}u(t-nk) + \frac{C(q)}{D(q)}e(t)$$

using a prediction error method.

`data` is an `iddata` object containing the output-input data. Frequency-domain signals are not supported by `bj`. Use `oe` instead.

The model orders can be specified by setting the argument `orders` to

```
orders = [ nb nc nd nf nk]
```

The parameters `nb`, `nc`, `nd`, and `nf` are the orders of the Box-Jenkins model and `nk` is the delay. Specifically,

$$nf: \quad F(q) = 1 + f_1 q^{-1} + \ldots + f_{nf} q^{-nf}$$

$$nb: \quad B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1}$$

$$nc: \quad C(q) = 1 + c_1 q^{-1} + \ldots + c_{nc} q^{-nc}$$

$$nd: \quad D(q) = 1 + d_1 q^{-1} + \ldots + d_{nd} q^{-nd}$$

The orders can also be defined as property name/property value pairs `(...,'nb',nb,...)`. Alternatively, you can specify the vector as

```
orders = mi
```

where mi is an initial guess at the Box-Jenkins model given in idpoly format.

For multiple-input systems, nb, nf, and nk are row vectors with as many entries as there are input channels. Entry number i then describes the orders and delays associated with the ith input.

The structure and the estimation algorithm are affected by any property name/property value pairs that are set in the input argument list. Useful properties are 'Focus', 'InitialState', 'Trace', 'MaxIter', 'Tolerance', 'LimitError', and 'FixedParameter'.

See Algorithm Properties and the reference pages for idmodel and idpoly for details of these properties and their possible values.

bj does not support multiple-output models. Use a state-space model for this case (see n4sid and pem).

Examples

Here is an example that generates data and stores the results of the startup procedure separately.

```
B = [0 1 0.5];
C = [1 -1 0.2];
D = [1 1.5 0.7];
F = [1 -1.5 0.7];
m0 = idpoly(1,B,C,D,F,0.1);
e = iddata([],randn(200,1));
u = iddata([],idinput(200));
y = sim(m0,[u e]);
z = [y u];
mi = bj(z,[2 2 2 2 1],'MaxIter',0)
m = bj(z,mi,'Maxi',10)
m.EstimationInfo
m = bj(z,m); % Continue if m.es.WhyStop shows that
             % maxiter is reached.
compare(z,m,mi)
```

Algorithm

`bj` uses essentially the same algorithm as `armax` with modifications to the computation of prediction errors and gradients.

See Also

`Algorithm Properties`

`EstimationInfo`

`idpoly`

`pem`

Purpose

Plot Bode diagram of frequency response with confidence interval

Syntax

```
bode(m)
[mag,phase,w] = bode(m)
[mag,phase,w,sdmag,sdphase] = bode(m)
bode(m1,m2,m3,...,w)
bode(m1,'PlotStyle1',m2,'PlotStyle2',...)
bode(m1,m2,m3,..'sd',sd,'mode',mode,'ap',ap)

bode(m1,m2,m3,'sd',sd,'mode',mode,'ap',ap,'fill')
```

Description

`bode` computes the magnitude and phase of the frequency response of `idmodel` and `idfrd` models. When invoked without left-hand arguments, `bode` produces a Bode plot on the screen.

`bode(m)` plots the Bode response of an arbitrary `idmodel` or `idfrd` model `m`. This model can be continuous or discrete, and SISO or MIMO. The `InputNames` and `OuputNames` properties of the models are used to plot the responses for different I/O channels in separate plots. Pressing the **Enter** key advances the plot from one input-output pair to the next one. Typing **Ctrl+C** aborts the plotting in an orderly fashion

If `m` contains information about both I/O channels and output noise spectra, only the I/O channels are shown. To show the output noise spectra, enter `m('n')` (`'n'` for `'noise'`) in the model list. Analogously, you can select specific I/O channels with normal subreferencing `m(ky,ku)`.

Argument w

`bode(m,w)` explicitly specifies the frequency range or frequency points to be used for the plot or for computing the response.

To focus on a particular frequency interval `[wmin,wmax]`, set `w = {wmin,wmax}` (notice the curly brackets). This plots the response for 100 frequency points logarithmically spaced from `wmin` to `wmax`. You can change this to `NP` points by using `w = {wmin,wmax,NP}`.

To use particular frequency points, set `w` to the vector of desired frequencies. Use `logspace` to generate logarithmically spaced frequency vectors. All frequencies should be specified in rad/s.

Note that the frequencies cannot be specified for `idfrd` objects. For those the plot and response are calculated for the internally stored frequencies. However, the plot is restricted to the range `{wmin,wmax}` if this is specified.

If no frequency range is specified, a default choice is made based on the dynamics of the model.

Property Name/Property Value Pairs 'sd'/sd, 'ap'/ap, and 'mode'/mode

The pairs can appear in any order or be omitted.

- `sd`: If `sd` is specified as a number larger than zero, confidence intervals for the functions are added to the graph as dash-dotted curves (of the same color as the estimate curve). They indicate the confidence regions corresponding to `sd` standard deviations. If an argument `'fill'` is included in the argument list, the confidence region is marked as a filled band instead.
- `ap`: By default, amplitude and phase plots are shown simultaneously for each I/O channel present in `m`. For spectra, phase plots are omitted. To show amplitude plots only, use `ap = 'A'`. For phase plots only, use `ap = 'P'`. The default is `ap = 'B'` for both plots.
- `mode`: To obtain all input/output plots in the same diagram use `mode = 'same'`.

Several Models

`bode(m1,m2,...,mN)` or `bode(m1,m2,...mN,w)` plots the Bode response of several `idmodel` or `idfrd` models on a single figure. The models can be mixes of different sizes and continuous/discrete. The sorting of the plots is based on the `InputNames` and `OutputNames`. If the frequencies `w` are specified, these will apply to all non-`idfrd` models in the list. If you want different frequencies for different models, you should thus first convert them to `idfrd` objects using the `idfrd` command.

`bode(m1,'PlotStyle1',...,mN,'PlotStyleN')` further specifies which color, line style, and/or marker should be used to plot each system, as in

```
bode(m1,'r--',m2,'gx')
```

Arguments

The output argument `w` contains the frequencies for which the response is given, whether specified among the input arguments or not. The output arguments `mag` and `phase` are 3-D arrays with dimensions

```
(number of outputs)x(number of inputs)x(length of w)
```

For SISO systems, `mag(1,1,k)` and `phase(1,1,k)` give the magnitude and phase (in degrees) at the frequency ω_k = `w(k)`. To obtain the result as a normal vector of responses, use `mag = mag(:)` and `phase = phase(:)`.

For MIMO systems, `mag(i,j,k)` is the magnitude of the frequency response at frequency `w(k)` from input `j` to output `i`, and similarly for `phase(i,j,k)`.

If `sdmag` and `sdphase` are specified, the standard deviations of the magnitude and phase are also computed. Then `sdmag` is an array of the same size as `mag`, containing the estimated standard deviations of the response, and analogously for `sdphase`.

See Also

`etfe`

`ffplot`

`freqresp`

`idfrd`

`nyquist`

`spa`

`spafdr`

Purpose

Compare model output and measured output

Syntax

```
compare(data,m);
compare(data,m,k)
compare(data,m,k,'Samples',sampnr,'InitialState',init,'OutputPlots
',Yplots)
compare(data,m1,m2,...,mN)
compare(data,m1,'PlotStyle1',...,mN,'PlotStyleN')
[yh,fit,x0] = compare(data,m1,'PlotStyle1',...,mN,'PlotStyleN',k)
```

Description

`data` is the output-input data in the usual `iddata` object format. `data` can also be an `idfrd` object with frequency-response data.

`compare` computes the output `yh` that results when the model `m` is simulated with the input `u`. m can be any `idmodel` or `idnlmodel` model object. The result is plotted together with the corresponding measured output `y`. The percentage of the output variation that is explained by the model

```
fit = 100*(1 - norm(yh - y)/norm(y-mean(y)))
```

is also computed and displayed. For multiple-output systems, this is done separately for each output. For frequency-domain data (or in general for complex valued data) the `fit` is still calculated as above, but only the absolute values of `y` and `yh` are plotted.

When the argument `k` is specified, the k step-ahead prediction of `y` according to the model `m` are computed instead of the simulated output. In the calculation of $yh(t)$, the model can use outputs up to time $t-k$: $y(s), s = t-k, t-k-1, \ldots$ (and inputs up to the current time t). The default value of `k` is `inf`, which gives a pure simulation from the input only. Note that for frequency-domain data, only simulation (`k = inf`) is allowed, and for time-series data (no input) only prediction (`k` not `inf`) is possible.

Property Name/Property Value Pairs

The optional property name/property value pairs `'Samples'/sampnr`, `'InitialState'/init`, and `'OutputPlots'/Yplots` can be given in any order.

The argument `Yplots` can be a cell array of strings. Only the outputs with `OutputName` in this array are plotted, while all are used for the necessary computations. If `Yplots` is not specified, all outputs are plotted.

The argument `sampnr` indicates that only the sample numbers in this row vector are plotted and used for the calculation of the fit. The whole data record is used for the simulation/prediction.

The argument `init` determines how to handle initial conditions in the models:

- `init = 'e'` (for `'estimate'`) estimates the initial conditions for best fit.
- `init = 'm'` (for `'model'`) used the model's internally stored initial state.
- `init = 'z'` (for `'zero'`) uses zero initial conditions.
- `init = x0`, where `x0` is a column vector of the same size as the state vector of the models, uses `x0` as the initial state.
- `init = 'e'` is the default.

Several Models

When several models are specified, as in `compare(data,m1,m2,...,mN)`, the plots show responses and fits for all models. In that case `data` should contain all inputs and outputs that are required for the different models. However, some models might correspond to subselections of channels and might not need all channels in `data`. In that case the proper handling of signals is based on the `InputNames` and `OutputNames` of `data` and the models.

With `compare(data,m1,'PlotStyle1',...mN,'PlotStyle2')`, the color, line style, and/or marker can be specified for the curves associated with the different models. The markers are the same as for the regular `plot` command. For example,

```
compare(data,m1,'g_*',m2,'r:')
```

If `data` contains several experiments, separate plots are given for the different experiments. In this case `sampnr`, if specified, must be a cell array with as many entries as there are experiments.

Arguments

When output arguments `[yh,fit,x0] = compare(data,m1,..,mN)` are specified, no plots are produced.

`yh` is a cell array of length equal to the number of models. Each cell contains the corresponding model output as an `iddata` object.

`fit` is, in the general case, a 3-D array with `fit(kexp,kmod,ky)` containing the fit (computed as above) for output `ky`, model `kmod`, and experiment `kexp`.

`x0` is a cell array, such that `x0{kmod}` is the estimated initial state for model number `kmod`. If data is multiexperiment, `X0{kmod}` is a matrix whose column number `kexp` is the initial state vector for experiment number `kexp`.

Examples

Split the data record into two parts. Use the first one for estimating a model and the second one to check the model's ability to predict six steps ahead.

```
ze = z(1:250);
zv = z(251:500);
m= armax(ze,[2 3 1 0]);
compare(zv,m,6);
compare(zv,m,6,'Init','z') % No estimation of
                           % the initial state.
```

See Also

pe

predict

sim

Purpose

Estimate covariance functions for time-domain `iddata` object

Syntax

```
R = covf(data,M)
R = covf(data,M,maxsize)
```

Description

`data` is an `iddata` object and `M` is the maximum delay -1 for which the covariance function is estimated. The routine is intended for time-domain data only.

Let `z` contain the output and input channels

$$z(t) = \begin{bmatrix} y(t) \\ u(t) \end{bmatrix}$$

where y and u are the rows of `data.OutputData` and `data.InputData`, respectively, with a total of `nz` channels.

`R` is returned as an nz^2 -by- M matrix with entries

$$R(i+(j-1)nz,k+1) = \frac{1}{N}\sum_{t=1}^{N} z_i(t)z_j(t+k) = \hat{R}_{ij}(k)$$

where z_j is the `j`th row of z, and missing values in the sum are replaced by zero.

The optional argument `maxsize` controls the memory size as explained under `Algorithm Properties`.

The easiest way to describe and unpack the result is to use

```
reshape(R(:,k+1),nz,nz) = E z(t)*z'(t+k)
```

Here `'` is complex conjugate transpose, which also explains how complex data is handled. The expectation symbol `E` corresponds to the sample means.

Algorithm

When `nz` is at most two, and when permitted by `maxsize`, a fast Fourier transform technique is applied. Otherwise, straightforward summing is used.

See Also

`iddata`

`spa`

Purpose

Estimate impulse response using prewhitened-based correlation analysis

Syntax

```
cra(data);
[ir,R,cl] = cra(data,M,na,plot);
cra(R);
```

Description

`data` is the output-input data given as an `iddata` object. The routine is intended for time-domain data only.

The routine only handles single-input-single-output data pairs. (For the multivariate case, apply `cra` to two signals at a time, or use `impulse`.) `cra` prewhitens the input sequence; that is, `cra` filters `u` through a filter chosen so that the result is as uncorrelated (white) as possible. The output `y` is subjected to the same filter, and then the covariance functions of the filtered `y` and `u` are computed and graphed. The cross correlation function between (prewhitened) input and output is also computed and graphed. Positive values of the lag variable then correspond to an influence from `u` to later values of `y`. In other words, significant correlation for negative lags is an indication of feedback from `y` to `u` in the data.

A properly scaled version of this correlation function is also an estimate of the system's impulse response `ir`. This is also graphed along with 99% confidence levels. The output argument `ir` is this impulse response estimate, so that its first entry corresponds to lag zero. (Negative lags are excluded in `ir`.) In the plot, the impulse response is scaled so that it corresponds to an impulse of height $1/T$ and duration T, where T is the sampling interval of the data.

The output argument `R` contains the covariance/correlation information as follows:

- The first column of `R` contains the lag indices.
- The second column contains the covariance function of the (possibly filtered) output.

- The third column contains the covariance function of the (possibly prewhitened) input.
- The fourth column contains the correlation function. The plots can be redisplayed by `cra(R)`.

The output argument `cl` is the 99% confidence level for the impulse response estimate.

The optional argument `M` defines the number of lags for which the covariance/correlation functions are computed. These are from `-M` to `M`, so that the length of `R` is `2M+1`. The impulse response is computed from `0` to `M`. The default value of `M` is `20`.

For the prewhitening, the input is fitted to an AR model of order `na`. The third argument of `cra` can change this order from its default value `na = 10`. With `na = 0` the covariance and correlation functions of the original data sequences are obtained.

`plot:` `plot = 0` gives no plots. `plot = 1` (the default) gives a plot of the estimated impulse response together with a 99% confidence region. `plot = 2` gives a plot of all the covariance functions.

An often better alternative to `cra` is the functions `impulse` and `step`, which use a high-order FIR model to estimate the impulse response.

Examples

Compare a second-order ARX model's impulse response with the one obtained by correlation analysis.

```
ir = cra(z);
m = arx(z,[2 2 1]);
imp = [1;zeros(19,1)];
irth = sim(m,imp);
subplot(211)
plot([ir irth])
title('impulse responses')
subplot(212)
plot([cumsum(ir),cumsum(irth)])
title('step responses')
```

See Also

impulse

step

Purpose

Class representing nonlinearity estimator with user-defined unit function for nonlinear ARX and Hammerstein-Wiener models

Syntax

```
C=customnet(H)
C=customnet(H,Property1,Value1,...PropertyN,ValueN)
```

Arguments

`H`

User-defined function handle of the unit function of the custom net.

`H` must point to a function of the form `[f,g,a] = function_name(x)`, where `f` is the value of the function, `g=df/dx` and indicates the unit function active range. `g` is significantly nonzero in the interval `[-a a]`.

Description

`customnet` is an object that stores a custom nonlinear estimator with a user-defined unit function.

You can use the constructor to create the nonlinearity object, as follows:

`C=customnet(H)` creates a nonlinearity estimator object with a user-defined unit function using the function handle `H`.

`C=customnet(H,Property1,Value1,...PropertyN,ValueN)` creates a nonlinearity estimator using property-value pairs defined in "customnet Properties" on page 2-59.

Remarks

Use `customnet` to define a nonlinear function $y = F(x)$, where y is scalar and x is an `m`-dimensional row vector. The unit function is based on the following function expansion with a possible linear term L:

$$F(x) = (x - r)PL + a_1 f\left((x - r)Qb_1 + c_1\right) + \ldots + a_n f\left((x - r)Qb_n + c_n\right) + d$$

where f is a unit function that you define using the function handle H.

P and Q are `m`-*by*-`p` and `m`-*by*-`q` projection matrices, respectively. The projection matrices P and Q are determined by principal component

analysis of estimation data. Usually, `p=m`. If the components of x in the estimation data are linearly dependent, then `p<m`. The number of columns of Q, `q`, corresponds to the number of components of `x` used in the unit function.

When used to estimate nonlinear ARX models, `q` is equal to the size of the `NonlinearRegressors` property of the `idnlarx` object. When used to estimate Hammerstein-Wiener models, `m=q=1` and Q is a scalar.

r is a 1-*by*-`m` vector and represents the mean value of the regressor vector computed from estimation data.

d, a, and c are scalars.

L is a `p`-*by*-1 vector.

b are `q`-*by*-1 vectors.

The function handle of the unit function of the custom net must have the form `[f,g,a] = function_name(x)`. This function must be vectorized, which means that for a vector or matrix `x`, the output arguments `f` and `g` must have the same size as `x` and be computed element-by-element.

customnet Properties

You can include property-value pairs in the constructor to specify the object.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% List all property values
get(C)
% Get value of NumberOfUnits property
C.NumberOfUnits
```

You can also use the `set` function to set the value of particular properties. For example:

```
set(C, 'LinearTerm', 'on')
```

The first argument to `set` must be the name of a MATLAB® variable.

Property Name	Description
`NumberOfUnits`	Integer specifies the number of nonlinearity units in the expansion. Default=`10`. For example: `customnet(H,'NumberOfUnits',5)`
`LinearTerm`	Can have the following values: • `'on'`—Estimates the vector L in the expansion. • `'off'`—Fixes the vector L to zero. For example: `customnet(H,'LinearTerm','on')`
`Parameters`	A structure containing the parameters in the nonlinear expansion, as follows: • `RegressorMean`: 1-*by*-`m` vector containing the means of `x` in estimation data, `r`. • `NonLinearSubspace`: `m`-*by*-`q` matrix containing Q. • `LinearSubspace`: `m`-*by*-`p` matrix containing P. • `LinearCoef`: `p`-*by*-1 vector L. • `Dilation`: `q`-*by*-1 matrix containing the values `b_k`. • `Translation`: 1-*by*-`n` vector containing the values `c_k`. • `OutputCoef`: `n`-*by*-1 vector containing the values `a_k`. • `OutputOffset`: scalar `d`. Typically, the values of this structure are set by estimating a model with a `customnet` nonlinearity.
`UnitFcn`	Stores the function handle that points to the unit function.

Examples

The following code, contained in `gaussunit.m`, defines a sample unit function.

```
[f, g, a] = GAUSSUNIT(x)
% x: unit function variable
% f: unit function value
% g: df/dx
% a: unit active range (g(x) is significantly
% nonzero in the interval [-a a])

% The unit function must be "vectorized": for
% a vector or matrix x, the output arguments f and g
% must have the same size as x,
% computed element-by-element.

% GAUSSUNIT customnet unit function example
[f, g, a] = gaussunit(x)
f =  exp(-x.*x);
if nargout>1
  g = - 2*x.*f;
  a = 0.2;
end
```

You typically use custom networks in `nlarx` and `nlhw` model estimation commands. For example:

```
% Define handle to example unit function.
H = @gaussunit;
% Estimate nonlinear ARX model using
% Gauss unit function with 5 units.
m = nlarx(Data,Orders,customnet(H,'NumberOfUnits',5));
```

See Also

`evaluate`

`nlarx`

`nlhw`

Purpose

Class representing custom regressor for nonlinear ARX models

Syntax

```
C=customreg(Function,Arguments)
C=customreg(Function,Arguments,Delays,Vectorized)
```

Arguments

`Function`
: Function handle or string representing a function of input and output variables.

`Arguments`
: Cell array of strings that represent the names of model inputs and outputs used in the function `Function`. Each input and output name must coincide with the strings in the `InputName` and `OutputName` properties of the corresponding `idnlarx` object. The size of `Arguments` must match the number of `Function` inputs.

`Delays`
: Vector of positive integers representing the delays of `Arguments` variables. The size of `Delays` must match the size of `Arguments`. Default: `1` for each vector element.

`Vectorized`
: Flag that indicates whether `Function` is in a format that supports vectorized computations when applied to vector arguments. Can have values `1` (true) and `0` (false). Default: `0`.

Description

`C=customreg(Function,Arguments)` creates an object that stores custom regressors for nonlinear ARX models. Custom regressor is defined using the function of input and output variables, `Function`.

`C=customreg(Function,Arguments,Delays,Vectorized)` create a custom regressor that includes the delays corresponding to inputs or outputs in `Arguments`.

For multiple-output models with `p` outputs, the custom regressor is `p`-by-1 cell array or an array of `customreg` object, where the `ky`th entry defines the custom regressor for output `ky`.

Remarks

Use the `customreg` object to define a custom regressor entering the nonlinearity estimator of the nonlinear ARX models (`idnlarx` object). The custom regressor function is usually a nonlinear function of inputs and outputs.

To list custom regressors of a multiple-output model, type the following command:

```
model.custom
```

To retrieve `r`th custom regressor for output `ky`, type the following command:

```
model.custom{ky}(r)
```

For more information about regressors, type `help regressors`.

For more information on creating custom regressors that are polynomial combinations of delayed inputs and outputs, type `help polyreg`.

Use the `Vectorized` property to specify whether to compute custom regressors using vectorized form. If you know that your regressor formulas can be vectorized, set `Vectorized` to `1` after creating the `customreg` object to achieve better performance. To better understand vectorization, consider that custom regressors are defined by a formula, such as `z=@(x,y)x^2*y`. `x` and `y` are vectors and each variable is evaluated over a time grid. Therefore, `z` must be evaluated for each `(xi,yi)` pair, and the results are concatenated to produce a `z` vector, as follows:

```
for k = 1:length(x)
   z(k) = x(k)^2*y(k)
end
```

The above expression is a nonvectorized computation and tends to be slow. Specifying a `Vectorized` computation uses MATLAB® vectorization rules to evaluate the regressor expression using matrices instead of the `FOR`-loop and results in faster computation, as follows:

```
% ".*" indicates element-wise operation
```

```
z=(x.^2).*y
```

customreg Properties

You can include property-value pairs in the constructor to specify the object.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% List all property values
get(C)
% Get value of Arguments property
C.Arguments
```

You can also use the `set` function to set the value of particular properties. For example:

```
set(C, 'Vectorized', 0)
```

The first argument to `set` must be the name of a MATLAB variable.

Property Name	Description
`Function`	Function handle or string representing a function of standards regressors. For example: `cr = @(x,y) x*y`
`Arguments`	Cell array of strings that represent the names of model inputs and outputs used in the function `Function`. Each input and output name must coincide with the strings in the `InputName` and `OutputName` properties of the `idnlarx` object with the custom regressor. The size of `Arguments` must match the number of `Function` inputs. For example, `Arguments` correspond to `{'y1','u1'}` in the following syntax: `C = customreg(cr,{'y1','u1'},[2 3])`

Property Name	Description
`Delays`	Vector of positive integers representing the delays of `Arguments` variables. The size of `Delays` must match the size of `Arguments`. Default: `1` for each vector element. For example, `Delays` correspond to `[2 3]` in the following syntax: `C = customreg(cr,{'y1','u1'},[2 3])`
`Vectorized`	Can have the following values: • `1`—`Function` is computed in vectorized form when called with vector arguments. • `0`—`Function` is not computed in vectorized form when called with vector arguments.

Examples

Consider a system that has input `u` and output `y`. Suppose that the following transformations are useful for predicting future outputs, based on physical insight into the relationship between measured variables:

- `u(t-1)sin(y(t-3))`
- `u(t-2)^3`

You can define custom regressors in any of the two ways:

- Cell array of strings. For example:

```
C={'u(t-1)*sin(y(t-3)','u(t-2)^3'}
```

- Object array of `customreg` objects. For example:

```
cr1=@(x,y) x*sin(y)
cr2=@(x) x^3
C=[customreg(cr1,{'u' 'y'},[1 3]),...
   customreg(cr2,{'u'},2)]
```

After you define the custom regressors, estimate the nonlinear ARX model using the following syntax:

```
m = nlarx(Data,Orders,linear,'CustomRegressors',C)
```

In this case, `u` and `y` are `Data` channel names, and the nonlinearity estimator `linear` specifies that the prediction is a linear function of custom regressors, where the custom regressor might be nonlinear. You can introduce additional nonlinearities using nonlinearity estimators. For a complete list of available nonlinearities, type `idprops idnlestimators`.

To create two custom regressors as an object array, use the following commands:

```
cr1=@(x,y) x*sin(y);
cr2=@(x) x^3;
C=[customreg(cr1,{'u1' 'y1'},[1 3]),...
   customreg(cr2,{'u1'},2)]
% Use this customreg object array for estimating
% nonlinear ARX model
m=nlarx(Data,Orders,'wavenet','CustomRegressor',C);
```

The following command is equivalent to the previous code snippet and incorporates custom regressor definitions directly in the estimator command:

```
m = nlarx(Data,Orders,'wavenet',...
          'CustomRegressors',...
         {'u1(t-1)*sin(y1(t-3))','u1(t-2)^3'});
```

See Also

`customreg`

`nlarx`

Purpose

Convert model from continuous to discrete time

Syntax

```
md = c2d(mc,T)
md = c2d(mc,T,method)
[md,G] = c2d(mc,T,method)
```

Description

`mc` is a continuous-time model such as any `idmodel` object (`idgrey`, `idproc`, `idpoly`, or `idss`). `md` is the model that is obtained when it is sampled with sampling interval `T`.

`method = 'zoh'` (default) makes the translation to discrete time under the assumption that the input is piecewise constant (zero-order hold).

`method = 'foh'` assumes the input to be piecewise linear between the sampling instants (first-order hold).

When you have the Control System Toolbox™ product installed, the following methods are also supported: `'tustin'`, `'prewarp'`, and `'matched'`. In these cases the covariance matrix is not transformed.

Note that the innovations variance λ of the continuous-time model is interpreted as the intensity of the spectral density of the noise spectrum. The noise variance in `md` is thus given as λ/T.

`idpoly` and `idss` models are returned in the same format. `idgrey` structures are preserved if their `CDMfile` property is equal to `'cd'`. Otherwise they are transformed to `idss` objects. `idproc` models are returned as `idgrey` objects.

For `idpoly` models, the covariance matrix is translated by the use of numerical derivatives. The step sizes used for the differentiation are given by the function `nuderst`. For `idss, idproc,` and `idgrey` models, the covariance matrix is not translated, but covariance information about the input-output properties is included in `md`. To inhibit the translation of covariance information (which may take some time), use `c2d(mc,T,'covariance','none')`.

The output argument `G` is a matrix that transforms the initial state `x0` of `mc` to the initial state of `md` as

```
X0d=G * [X0; u(0)],
```

where `u(0)` is the input at time 0. For `idproc` models, the state variables correspond to those of `idgrey(mc)`. For `idpoly` models, `G` is returned as the empty matrix.

Examples

Define a continuous-time system and study the poles and zeros of the sampled counterpart.

```
mc = idpoly(1,1,1,1,[1 1 0],'Ts',0);
md = c2d(mc,0.5);
pzmap(md)
```

See Also

`d2c`

Purpose

Map past input/output data to current states of `idnlarx` model

Syntax

```
X = data2state(MODEL,IOSTRUCT)
X = data2state(MODEL,DATA)
```

Description

`X = data2state(MODEL,IOSTRUCT)` maps the input and output samples in `IOSTRUCT` to the current states of `MODEL`, `X`. For a definition of the states of `idnlarx` models, see "Definition of idnlarx States" on page 2-184. The data in `IOSTRUCT` is interpreted as past samples of data, so that the returned state values must be interpreted as values at the time immediately after the time corresponding to the last (most recent) sample in the data.

`X = data2state(MODEL,DATA)` maps the input and output samples from `DATA` to the current states, `X`, of the model.

Input

- `MODEL`: `idnlarx` model.
- `IOSTRUCT`: Structure with fields `Input` and `Output`. Samples in `IOSTRUCT` must be in the order of increasing time (the last row of values corresponds to the most recent time). Each field contains data samples corresponding to the past input and output of `MODEL` respectively.
  - `Input`: Matrix of `NU` columns, where `NU` is the number of inputs. The number of rows can be equal to either of the following:
    - Maximum input delay in `MODEL` (maximum across all input variables).
    - 1 to specify of steady-state (constant) input values.
  - `Output`: Matrix of `NY` columns, where `NY` is the number of outputs. The number of rows can be equal to either of the following:
    - Maximum input delay in `MODEL` (maximum across all output variables).
    - 1 to specify of steady-state (constant) output values.

- DATA: iddata object containing data samples. Samples in DATA must be in the order of increasing time (the last row of values corresponds to the most recent time). The number of samples in DATA must be greater than or equal to the maximum delay in the model across all input and output variables.

Note To determine maximum delay in each input and output channel of MODEL, use the getDelayInfo(idnlarx) command. For more information, see the getDelayInfo(idnlarx) reference page.

Input

X is the state vector of MODEL corresponding to the time after the most recent sample in the input data (IOSTRUCT or DATA).

Examples

In this example you determine the current state of an idnlarx model.

1 Load your data and create a data object.

```
load motorizedcamera;
z = iddata(y,u,0.02,'Name','Motorized Camera', ...
                        'TimeUnit','s');
```

2 Estimate the idnlarx model from the data. The model has 6 inputs and 3 outputs.

```
mw1 = nlarx(z,[ones(2,2),ones(2,6),ones(2,6)],wavenet);
```

3 Compute the maximum delays across all output variables in mw1.

```
MaxDelays =  getDelayInfo(mw1); % delays arranged channelwise
```

4 Determine the maximum delays across all channels in mw1.

```
IOData = struct('Input', ...
                 rand(max(MaxDelays(3+1:end)),6),...
                'Output', ...
```

```
                    rand(max(MaxDelays(1:3)),2));
```

5 Compute the current states of `mw1` based on the past data in `IOSTRUCT`.

```
X = data2state(mw1,IOData)
```

The previous command computes the state vector.

Note You can specify constant input levels with values `(10,20,30)` for the three input variables by setting `IOSTRUCT.Input = [10,20,30]` in place of a matrix of values.

See Also

`findop(idnlarx)`

`findstates(idnlarx)`

`getDelayInfo(idnlarx)`

deadzone

Purpose

Class representing dead-zone nonlinearity estimator for Hammerstein-Wiener models

Syntax

```
s=deadzone(ZeroInterval,I)
```

Description

`deadzone` is an object that stores the dead-zone nonlinearity estimator for estimating Hammerstein-Wiener models.

You can use the constructor to create the nonlinearity object, as follows:

`s=deadzone(ZeroInterval,I)` creates a dead-zone nonlinearity estimator object, initialized with the zero interval `I`.

Use `evaluate(d,x)` to compute the value of the function defined by the `deadzone` object `d` at `x`.

Remarks

Use `deadzone` to define a nonlinear function $y = F(x)$, where F is a function of x and has the following characteristics:

$$\begin{aligned} a \le x < b \quad & F(x) = 0 \\ x < a \quad & F(x) = x - a \\ x \ge b \quad & F(x) = x - b \end{aligned}$$

y and x are scalars.

Properties

You can specify the property value as an argument in the constructor to specify the object.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% List ZeroInterval property value
get(d)
d.ZeroInterval
```

You can also use the `set` function to set the value of particular properties. For example:

```
set(d, 'ZeroInterval', [-1.5 1.5])
```

The first argument to `set` must be the name of a MATLAB® variable.

Property Name	Description
`ZeroInterval`	1-*by*-2 row vector that specifies the initial zero interval of the nonlinearity. Default=`[NaN NaN]`. For example: `deadzone('ZeroInterval',[-1.5 1.5])`

Examples

Use `deadzone` to specify the dead-zone nonlinearity estimator in Hammerstein-Wiener models. For example:

```
m=nlhw(Data,Orders,deadzone([-1 1]),[]);
```

The dead-zone nonlinearity is initialized at the interval `[-1 1]`. The interval values are adjusted to the estimation data by `nlhw`.

See Also

`nlhw`

delayest

Purpose

Estimate time delay (dead time) from data

Syntax

```
nk = delayest(Data)
nk = delayest(Data,na,nb,nkmin,nkmax,maxtest)
```

Description

`Data` is an `iddata` object containing the input-output data. It can also be an `idfrd` object defining frequency-response data. Only single-output data can be handled.

`nk` is returned as an integer or a row vector of integers, containing the estimated time delay in samples from the input(s) to the output in `Data`.

The estimate is based on a comparison of ARX models with different delays:

$$y(t) + a_1 y(t-1) + \ldots + a_{na} y(t-na) = \ldots$$
$$b_1 u(t-nk) + \ldots + b_{nb} u(t-nb-nk+1)$$

The integer `na` is the order of the A polynomial (default 2). `nb` is a row vector of length equal to the number of inputs, containing the order(s) of the B polynomial(s) (default all 2).

`nkmin` and `nkmax` are row vectors of the same length as the number of inputs, containing the smallest and largest delays to be tested. Defaults are `nkmin = 0` and `nkmax = nkmin+20`.

If `nb`, `nkmax`, and/or `nkmin` are entered as scalars in the multiple-input case, all inputs will be assigned the same values.

`maxtest` is the largest number of tests allowed (default 10,000).

Purpose

Subtract trend from time-domain, frequency-domain, or time-series data signal

Syntax

```
zd = detrend(z)
zd = detrend(z,o,brkp)
```

Input

`z`

Name of `iddata` object in the MATLAB® workspace containing time-domain, frequency-domain, or time-series data.

Note For frequency-domain data, you can only remove a zeroth-order trend.

`o`

Order of the trend, specified as one of the following values:

- `0`: (Default) Specifies to subtract a mean value (zeroth-order trend) from each signal.
- `1`: Specifies to subtract a linear trend (first-order trend) from each signal after performing a least-squares fit on the data.

`brkp`

Data sample index that defines the discontinuities between successive linear trends, specified as one of the following types of values:

- Scalar: Specifies one discontinuity.
- Vector: Specifies several discontinuities, indicating to subtract multiple linear trends.

Description

`zd = detrend(z)` subtracts the mean value (zeroth-order trend) from each signal in `z` and returns an `iddata` object `zd`. If `z` is frequency-domain data, the response at frequency `0` is zero.

`zd = detrend(z,o,brkp)` subtracts the mean value (`o = 0`) or linear trend (`o = 1`). You can subtract piecewise linear trends by specifying one or more breakpoints `brkp`, where a *breakpoint* is a data sample index (time value) that defines the discontinuities between successive linear trends. When `brkp` contains breakpoints that match the time vector, `detrend` subtracts a continuous piecewise linear trend.

Note `detrend` for `iddata` objects differs from `detrend` in the Signal Processing Toolbox™ product.

Examples

Example 1

In this example, you subtract several linear trends that connect at three breakpoints `[30 60 90]`, which correspond to the data sample indexes data where breakpoints occur.

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

Example 2

In this example, you subtract a mean value from the input signal and a V-shaped trend from the output signal, such that the V peak occurs at the breakpoint value of `119`.

```
zd1(:,1,[]) = detrend(z(:,1,[]),1,119);
zd2(:,[],1) = detrend(z(:,[],1));
zd = [zd1,zd2];
```

See Also

"Subtracting Trends from Signals (Detrending)"

Purpose

Difference signals in `iddata` objects

Syntax

```
zdi = diff(z)
zdi = diff(z,n)
```

Description

`z` is a time-domain `iddata` object. `diff(z)` and `diff(z,n)` apply this command to each of the input/output signals in `z`.

Purpose

Convert model from discrete to continuous time

Syntax

```
mc = d2c(md)
mc = d2c(md,method)
mc = d2c(md,'CovarianceMatrix',cov,'InputDelay',inpd)
```

Description

The discrete-time model `md`, given as any `idmodel` object, is converted to a continuous-time counterpart `mc`. The covariance matrix of the parameters in the model is also translated using the Gauss approximation formula and numerical derivatives of the transformation. The step sizes in the numerical derivatives are determined by the function `nuderst`. To inhibit the translation of the covariance matrix and save time, enter among the input arguments `(...,'CovarianceMatrix,'None,....))` (any abbreviations will do).

`method` is one of the input intersample behaviors `'zoh'` (zero-order hold) or `'foh'` (first-order hold). If `method` is not specified, the `InterSample` behavior of the data from which `md` was estimated is used.

When you have Control System Toolbox™ installed, the following methods are also supported: `'tustin'`, `'prewarp'`, and `'matched'`. In these cases no translation of the covariance matrix takes place.

If the discrete-time model contains pure time delays, that is, $nk > 1$, then these are first removed before the transformation is made. These delays are appended as pure time delay (dead time) to the continuous-time model as the property `InputDelay`. To have the time delay approximated by a finite-dimensional continuous system, enter among the input arguments `(...,'InputDelay',0,...)`.

If the noise variance is λ in `md`, and its sampling interval is T, then the continuous-time model has an indicated level of noise spectral density equal to $T\ \lambda$.

While `idpoly` and `idss` models are returned in the same format, `idarx` models are returned as `idss` models `mc`. The reason is that the transformation does not preserve the special structure of `idarx`. The `idgrey` structures are preserved if their `CDMfile` property is equal to `cd`. Otherwise they are transformed to `idss` objects.

Note The transformation from discrete to continuous time is not unique. `d2c` selects the continuous-time counterpart with the slowest time constants consistent with the discrete-time model. The lack of uniqueness also means that the transformation can be ill-conditioned or even singular. In particular, poles on the negative real axis, in the origin, or in the point 1, are likely to cause problems. Interpret the results with care.

Examples

Transform an identified model to continuous time and compare the frequency responses of the two models.

```
m = n4sid(data,3)
mc = d2c(m);
bode(m.mc,'sd',3)
```

Note that you can include the transformation to continuous time in the `n4sid` command by specifying the model to be continuous time.

```
mc = n4sid(data,3,'Ts',0)
```

See Also

`c2d`

`nuderst`

EstimationInfo

Purpose

Information about estimation process results

Syntax

```
m.EstimationInfo
m.es
m.es.DataLength, etc
```

Description

Any estimated model has the property `EstimationInfo`, which is a structure whose fields give information about the results of the estimation. Depending on whether it is an estimated parametric `idmodel` or an estimated frequency response `idfrd`, `EstimationInfo` will contain different fields.

idmodel Case

The model structure will contain the properties `ParameterVector`, `CovarianceMatrix`, and `NoiseVariance`, which are all calculated in the estimation process (see the reference page for `idmodel`). In addition, `EstimationInfo` contains the following fields:

- `Status`: Information whether the model has been estimated, or modified after being estimated.
- `Method`: Name of the estimation command that produced the model.
- `LossFcn`: Value of the identification criterion at the estimate. Normally equal to the determinant of the covariance matrix of the prediction errors, that is, the determinant of `NoiseVariance`. Note that the loss function for the minimization might be different due to `LimitError`. The value of the nonrobustified loss function is always stored in `LossFcn`.
- `FPE`: Akaike's Final Prediction Error, defined as `LossFcn *(1+d/N}/(1-d/N)`, where `d` is the number of estimated parameters and `N` is the length of the data record.
- `DataName`: Name of the data set from which the model was estimated. This is equal to the property `name` of the `iddata` object. If this was not defined, the name of the `iddata` variable is used.
- `DataLength`: Length of the data record.

- `DataTs`: Sampling interval of the data.
- `DataDomain`: `'Time'` or `'Frequency'`, depending on the data domain.
- `DataInterSample`: Intersample behavior of the data from which the model was estimated. This equals the property `InterSample` of the `iddata` object. (See `iddata`.)
- `WhyStop`: For models that have been estimated by iterative search. The stopping rule that caused the iterations to terminate. Assumes values such as `'MaxIter reached'`, `'No improvement possible along the search vector'`, or `'Near (local) minimum'`. The latter means that the expected improvement is less than `Tolerance` (see `Algorithm Properties`).
- `UpdateNorm`: Norm of the Gauss-Newton vector at the last iteration.
- `LastImprovement`: Relative improvement of the criterion value at the last iteration.
- `Iterations`: Number of iterations used in the search.
- `InitialState`: Option actually used when `Model.InitialState = 'auto'`.
- `N4Weight`: For `n4sid` estimates, or estimates that have been initialized by `n4sid`: the actual value of `N4Weight` used.
- `N4Horizon`: For `n4sid` estimates, or estimates that have been initialized by `n4sid`: the actual value of `N4Horizon` used. See `n4sid` and `Algorithm Properties`.

idfrd Case

If the `idfrd` model is obtained from an estimated parametric model,

```
g = idfrd(Model)
```

`g.EstimationInfo` is the same as `Model.EstimationInfo` as described above.

For an `idfrd` model that has been estimated from `etfe`, `spa`, or `spafdr`, `EstimationInfo` contains the following fields:

- `Status`: Whether the model is estimated or directly constructed.
- `Method`: `etfe`, `spa`, or `spafdr`
- `WindowSize`: Resolution parameter (or vector) used for the estimation
- `DataName`, `DataLength`, `DataTs`, `DataDomain`, `DataInterSample`: Properties of the estimation data as above.

See Also

`Algorithm Properties`

`idpoly`

`idss`

Purpose

Estimate empirical transfer functions and periodograms returning `idfrd` object

Syntax

```
g = etfe(data)
g = etfe(data,M,N)
```

Description

`etfe` estimates the transfer function `g` of the general linear model

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

`data` contains the output-input data and is an `iddata` object (time or frequency domain).

`g` is given as an `idfrd` object with the estimate of $G(e^{i\omega})$ at the frequencies

```
w = [1:N]/N*pi/T
```

The default value of `N` is `128`.

In case `data` contains a time series (no input channels), `g` is returned as the periodogram of `y`.

When `M` is specified other than the default value `M = []`, a smoothing operation is performed on the raw spectral estimates. The effect of `M` is then similar to the effect of `M` in `spa`. This can be a useful alternative to `spa` for narrowband spectra and systems, which require large values of `M`.

When `etfe` is applied to time series, the corresponding spectral estimate is normalized in the way that is defined in "Understanding Spectrum Normalization". `etfe` normalization differs from the `spectrum` normalization in the Signal Processing Toolbox product.

If the (input) data is marked as periodic (`data.Period = integer`) and contains an even number of periods, the response is computed at the frequencies `k*2*pi/period` for `k = 0` up to the Nyquist frequency.

Examples

Compare an empirical transfer function estimate to a smoothed spectral estimate.

```
ge = etfe(z);
gs = spa(z);
bode(ge,gs)
```

Generate a periodic input, simulate a system with it, and compare the frequency response of the estimated model with the true system at the excited frequency points.

```
m = idpoly([1 -1.5 0.7],[0 1 0.5]);
u = iddata([],idinput([50,1,10],'sine'));
u.Period = 50;
y = sim(m,u);
me = etfe([y u])
bode(me,'b*',m)
```

Algorithm

The empirical transfer function estimate is computed as the ratio of the output Fourier transform to the input Fourier transform, using `fft`. The periodogram is computed as the normalized absolute square of the Fourier transform of the time series.

You obtain the smoothed versions (`M` less than the length of `z`) by applying a Hamming window to the output fast Fourier transform (FFT) times the conjugate of the input FFT, and to the absolute square of the input FFT, respectively, and subsequently forming the ratio of the results. The length of this Hamming window is equal to the number of data points in `z` divided by `M`, plus one.

See Also

`bode`

`ffplot`

`freqresp`

`idfrd`

```
nyquist
spa
spafdr
```

evaluate

Purpose

Value of nonlinearity estimator at given input

Syntax

```
value = evaluate(nl,x)
```

Arguments

`nl`

Nonlinearity estimator object.

`x`

Value at which to evaluate the nonlinearity.

If `nl` is a single nonlinearity estimator, then `x` is a `1`-by-`nx` row vector or an `nv`-by-`nx` matrix, where `nx` is the dimension of the regression vector input to `nl` (`size(nl)`) and `nv` is the number of points where `nl` is evaluated.

If `nl` is an array of `ny` nonlinearity estimators, then `x` is a `1`-by-`ny` cell array of `nv`-by-`nx` matrices.

Description

`value = evaluate(nl,x)` computes the value of a nonlinear estimator object of type `customnet`, `deadzone`, `linear`, `neuralnet`, `pwlinear`, `saturation`, `sigmoidnet`, `treepartition`, or `wavenet`.

Example

The following syntax evaluates the nonlinearity of an estimated nonlinear ARX model `m`:

```
value = evaluate(m.Nonlinearity,x)
```

where `m.Nonlinearity` accesses the nonlinearity estimator of the nonlinear ARX model.

See Also

`idnlarx`

`idnlhw`

Purpose

Concatenate frequency-domain signals in `idfrd` and `iddata` objects

Syntax

```
Mc = fcat(M1,M2,...Mn)
```

Description

`M1`, `M2`, etc., are all `idfrd` objects or `iddata` frequency-domain objects.

`Mc` is the corresponding object obtained by concatenation of the responses at all the frequencies in `Mk`.

Note that for `iddata` objects, this is the same as vertical concatenation (`vertcat`).

```
Mc = [M1;M2;..;Mn].
```

See Also

`fselect`

`iddata`

`idfrd`

feedback

Purpose

Identify possible feedback in `iddata` data

Syntax

```
[fbck,fbck0,nudir] = feedback(Data)
```

Description

`Data` is an `iddata` set with *Ny* outputs and *Nu* inputs.

`fbck` is an *Ny*-by-*Nu* matrix indicating the feedback. The *ky,ku* entry is a measure of feedback from output *ky* to input *ku*. The value is a probability *P* in percent. Its interpretation is that if the hypothesis that there is no feedback from output *ky* to input *ku* were tested at the level *P*, it would have been rejected. An intuitive but technically incorrect way of thinking about this is to see *P* as "the probability of feedback." Often only values above 90% are taken as indications of feedback. When `fbck` is calculated, direct dependence at lag zero between $u(t)$ and $y(t)$ is not regarded as a feedback effect.

`fbck0`: Same as `fbck`, but direct dependence at lag 0 between $u(t)$ and $y(t)$ is viewed as feedback effect.

`nudir`: A vector containing those input numbers that appear to have a direct effect on some outputs, that is, no delay from input to output.

See Also

`advice`

`iddata`

`idmodel`

Purpose

Plot frequency response and spectra

Syntax

```
ffplot(m)
[mag,phase,w] = ffplot(m)
[mag,phase,w,sdmag,sdphase] = ffplot(m)
ffplot(m1,m2,m3,...,w)
ffplot(m1,'PlotStyle1',m2,'PlotStyle2',...)
ffplot(m1,m2,m3,..'sd',sd,'mode',mode,'ap',ap)
```

Description

This function has exactly the same syntax as `bode`. The only difference is that it gives graphs with linear frequency scales and Hz as the frequency unit.

See Also

`bode`

`etfe`

`freqresp`

`idfrd`

`nyquist`

`spa`

`spafdr`

Purpose

Transform `iddata` object to frequency domain

Syntax

```
Datf = fft(Data)
Datf = fft(Data,N)
Datf = fft(Data,N,'complex')
```

Description

If `Data` is a time-domain `iddata` object with real-valued signals and with constant sampling interval `Ts`, `Datf` is returned as a frequency-domain `iddata` object with the frequency values equally distributed from frequency 0 to the Nyquist frequency. Whether the Nyquist frequency actually is included or not depends on the signal length (even or odd). Note that the FFTs are normalized by dividing each transform by the square root of the signal length. That is in order to preserve the signal power and noise level.

In the default case, the length of the transformation is determined by the signal length. A second argument `N` will force FFT transformations of length `N`, padding with zeros if the signals in `Data` are shorter and truncating otherwise. Thus the number of frequencies in the real signal case will be `N/2` or `(N+1)/2`. If `Data` contains several experiments, `N` can be a row vector of corresponding length.

For real signals, the default is that `Datf` only contains nonnegative frequencies. For complex-valued signals, negative frequencies are also included. To enforce negative frequencies in the real case, add a last argument, `'Complex'`.

See Also

`iddata`

`ifft`

Purpose

Compute operating point for nonlinear ARX model

Syntax

```
[X,U] = findop(SYS,'steady',InputLevel,OutputLevel)
[X,U] = findop(SYS,SPEC)
[X,U] = findop(SYS,'snapshot',T,UIN,XO)
[X,U,REPORT] = findop(...)
findop(SYS,...,PVPairs)
```

Description

`[X,U] = findop(SYS,'steady',InputLevel,OutputLevel)` computes operating-point state values, `X`, and input values, `U`, from steady-state specifications for an `idnlarx` model. For more information about the states of an `idnlarx` model, see "Definition of idnlarx States" on page 2-184.

`[X,U] = findop(SYS,SPEC)` computes the equilibrium operating point using the specifications in the object `SPEC`. Whereas the previous command only lets you specify the input and output level, `SPEC` provides additional specification for computing the steady-state operating point.

`[X,U] = findop(SYS,'snapshot',T,UIN,XO)` computes the operating point at a simulation snapshot of time `T` using the specified input and initial state values.

`[X,U,REPORT] = findop(...)` creates a structure, `REPORT`, containing information about the algorithm for computing an operating point.

`findop(SYS,...,PVPairs)` specifies property-value pairs for setting algorithm options.

Input

- `SYS`: `idnlarx` (nonlinear ARX) model.
- `'steady'`: Computes operating point using steady-state input and output levels.
- `'snapshot'`: Computes operating point at simulating snapshot of model `SYS` at time `T`.
- `InputLevel`: Steady-state input level for computing operating point. Use `NaN` when the value is unknown.

- `OutputLevel`: Steady-state output level for computing the operating point. Use `NaN` when the value is unknown.
- `SPEC`: Operating point specifications object. Use `SPEC = OPERSPEC(SYS)` to construct the `SPEC` object for model `SYS`. Then, configure `SPEC` options, such as signal bounds, known values, and initial guesses. See `operspec(idnlarx)` for more information.
- `T`: Simulation snapshot time at which to compute the operating point.
- `UIN`: Input for simulating the model. `UIN` is a double matrix or an `iddata` object. The number of input channels in `UIN` must match the number of `SYS` inputs.
- `X0`: Initial states for model simulation.

  Default: Zero.
- `PVPairs`: Property-value pairs for customizing the model `Algorithm` property fields, such as `SearchMethod`, `MaxSize`, and `Tolerance`.

Input

- `X`: Operating point state values.
- `U`: Operating point input value.
- `REPORT`: Structure containing the following fields:
  - `SearchMethod`: String indicating the value of the `SearchMethod` property of `MODEL.Algorithm`.
  - `WhyStop`: String describing why the estimation stopped.
  - `Iterations`: Number of estimation iterations.
  - `FinalCost`: Final value of the sum of squared errors that the algorithm minimizes.
  - `FirstOrderOptimality`: Measure of the gradient of the search direction at the final parameter values when the search algorithm terminates. It is equal to the ∞-norm of the gradient vector.
  - `SignalLevels`: Structure containing fields `Input` and `Output`, which are the input and output signal levels of the operating point.

Algorithm

`findop` computes the operating point from steady-state operating point specifications or at a simulation snapshot.

Computing the Operating Point from Steady-State Specifications

You specify to compute the steady-state operating point by calling `findop` in either of the following ways:

```
[X,U] = findop(SYS,'steady',InputLevel,OutputLevel)
[X,U] = findop(SYS,SPEC)
```

When you use the syntax `[X,U] = findop(SYS,'steady',InputLevel,OutputLevel)`, the algorithm assumes the following operating-point specifications:

- All finite input values are fixed values. Any `NaN` values specify an unknown input signal with the initial guess of `0`.
- All finite output values are initial guess values. Any `NaN` values specify an unknown output signal with the initial guess of `0`.
- The minimum and maximum bounds have default values (`-/+ Inf`) for both `Input` and `Output` properties in the specification object.

Using the syntax `[X,U] = findop(SYS,SPEC)`, you can specify additional information, such as the minimum and maximum constraints on the input/output signals and whether certain inputs are known (fixed).

To compute the states, `X`, and the input, `U`, of the steady-state operating point, `findop` uses the algorithm specified in the `SearchMethod` property of `MODEL.Algorithm` to minimize the norm of the error $e(t) = y(t)\text{-}f(x(t), u(t))$, where f is the nonlinearity estimator, $x(t)$ are the model states, and $u(t)$ is the input.

The algorithm uses the following independent variables for minimization:

- Unknown (unspecified) inputs

- Output signals

Because the states of a nonlinear ARX (`idnlarx`) model are delayed samples of the input and output variables, the values of all the states are the constant values of the corresponding steady-state inputs and outputs. For more information about the definition of nonlinear ARX model states, see "Definition of idnlarx States" on page 2-184.

Computing the Operating Point at a Simulation Snapshot

When you use the syntax `[X,U] = findop(SYS,'snapshot',T,UIN,X0)`, the algorithm simulates the model output until the snapshot time, `T`. At the snapshot time, the algorithm passes the input and output samples to the `data2state` command to map these values to the current state vector.

Note For snapshot-based computations, `findop` does not perform numerical optimization.

Examples

In this example, you compute the operating point of an `idnlarx` model for a steady-state input level of 1.

1 Estimate an `idnlarx` model from sample data `iddata2`.

```
load iddata2;
M = nlarx(z2,[4 3 2],'wavenet');
```

2 Compute the steady-state operating point for an input level of 1.

```
x0 = findop(M,'steady',1,NaN)
```

The resulting operating-point state values are, as follows:

```
x0 =
```

```
    7.323167529215261
    7.323167529215261
    7.323167529215261
    7.323167529215261
    1.000000000000000
    1.000000000000000
    1.000000000000000
    1.000000000000000
```

See Also

data2state(idnlarx)

operspec(idnlarx)

sim(idnlarx)

findop(idnlhw)

Purpose

Compute operating point for Hammerstein-Wiener model

Syntax

```
[X,U] = findop(SYS,'steady',InputLevel,OutputLevel)
[X,U] = findop(SYS,SPEC)
[X,U] = findop(SYS,'snapshot',T,UIN,XO)
[X,U,REPORT] = findop(...)
findop(SYS,...,PVPairs)
```

Description

`[X,U] = findop(SYS,'steady',InputLevel,OutputLevel)` computes operating-point state values, `X`, and input values, `U`, from steady-state specifications for an `idnlhw` model. For more information about the states of an `idnlhw` model, see "Definition of idnlhw States" on page 2-219.

`[X,U] = findop(SYS,SPEC)` computes the equilibrium operating point using the specifications in the object `SPEC`. Whereas the previous command only lets you specify the input and output level, `SPEC` provides additional specification for computing the steady-state operating point.

`[X,U] = findop(SYS,'snapshot',T,UIN,XO)` computes the operating point at a simulation snapshot of time `T` using the specified input and initial state values.

`[X,U,REPORT] = findop(...)` creates a structure, `REPORT`, containing information about the algorithm for computing an operating point.

`findop(SYS,...,PVPairs)` specifies property-value pairs for setting algorithm options.

Input

- `SYS`: `idnlhw` (Hammerstein-Wiener) model.
- `'steady'`: Computes operating point using steady-state input and output levels.
- `'snapshot'`: Computes operating point at simulating snapshot of model `SYS` at time `T`.
- `InputLevel`: Steady-state input level for computing operating point. Use `NaN` when the value is unknown. Do not enter `OutputLevel` when `InputLevel` does not contain any `NaN` values.

- `OutputLevel`: Steady-state output level for computing the operating point. Use `NaN` when the value is unknown.
- `SPEC`: Operating point specifications object. Use `SPEC = OPERSPEC(SYS)` to construct the `SPEC` object for model `SYS`. Then, configure `SPEC` options, such as signal bounds, known values, and initial guesses. See `operspec(idnlhw)` for more information.
- `T`: Simulation snapshot time at which to compute the operating point.
- `UIN`: Input for simulating the model. `UIN` is a double matrix or an `iddata` object. The number of input channels in `UIN` must match the number of `SYS` inputs.
- `X0`: Initial states for model simulation.

  Default: Zero.
- `PVPairs`: Property-value pairs for customizing the model `Algorithm` property fields, such as `SearchMethod`, `MaxSize`, and `Tolerance`.

Input

- `X`: Operating point state values.
- `U`: Operating point input value.
- `REPORT`: Structure containing the following fields:
  - `SearchMethod`: String indicating the value of the `SearchMethod` property of `MODEL.Algorithm`.
  - `WhyStop`: String describing why the estimation stopped.
  - `Iterations`: Number of estimation iterations.
  - `FinalCost`: Final value of the sum of squared errors that the algorithm minimizes.
  - `FirstOrderOptimality`: Measure of the gradient of the search direction at the final parameter values when the search algorithm terminates. It is equal to the ∞-norm of the gradient vector.
  - `SignalLevels`: Structure containing fields `Input` and `Output`, which are the input and output signal levels of the operating point.

Algorithm

`findop` computes the operating point from steady-state operating point specifications or at a simulation snapshot.

Computing the Operating Point from Steady-State Specifications

You specify to compute the steady-state operating point by calling `findop` in either of the following ways:

```
[X,U] = findop(SYS,'steady',InputLevel,OutputLevel)
[X,U] = findop(SYS,SPEC)
```

When you use the syntax `[X,U] = findop(SYS,'steady',InputLevel,OutputLevel)`, the algorithm assumes the following operating-point specifications:

- All finite input values are fixed values. Any `NaN` values specify an unknown input signal with the initial guess of `0`.
- All finite output values are initial guess values. Any `NaN` values specify an unknown output signal with the initial guess of `0`.
- The minimum and maximum bounds have default values (`-/+ Inf`) for both `Input` and `Output` properties in the specification object.

Using the syntax `[X,U] = findop(SYS,SPEC)`, you can specify additional information, such as the minimum and maximum constraints on the input/output signals and whether certain inputs are known (fixed).

`findop` uses a different approach to compute the steady-state operating point depending on how much information you provide for this computation:

- When you specify values for all input levels (no `NaN` values). For a given input level, U, the equilibrium state values are X = inv(I-`A`)*`B`*$f(U)$, where `[A,B,C,D] = ssdata(model.LinearModel)`, and $f()$ is the input nonlinearity.

- When you specify known and unknown input levels. `findop` uses numerical optimization to minimize the norm of the error and compute the operating point. The total error is the union of contributions from e_1 and e_2 , $e(t) = (e_1(t)e_2(t))$, such that:
  - e_1 applies for known outputs and the algorithm minimizes $e_1 = y\text{-}g(L(x,f(u)))$, where f is the input nonlinearity, $L(x,u)$ is the linear model with states x, and g is the output nonlinearity.
  - e_2 applies for unknown outputs and the error is a measure of whether these outputs are within the specified minimum and maximum bounds. If a variable is within its specified bounds, the corresponding error is zero. Otherwise, the error is equal to the distance from the nearest bound. For example, if a free output variable has a value z and its minimum and maximum bounds are L and U, respectively, then the error is e_2= max[z-U, L-z, 0].

  The independent variables for the minimization problem are the unknown inputs. In the error definition e, both the input u and he states x are free variables. To get an error expression that contains only unknown inputs as free variables, the algorithm `findop` specifies the states as a function of inputs by imposing steady-state conditions: x = inv(I-A)*B*f(U), where [A,B,C,D] are state-space parameters corresponding to the linear model $L(x,u)$. Thus, substituting x = inv(I-A)*B*f(U) into the error function results in an error expression that contains only unknown inputs as free variables computed by the optimization algorithm.

Computing the Operating Point at a Simulation Snapshot

When you use the syntax `[X,U] = findop(SYS,'snapshot',T,UIN,XO)`, the algorithm simulates the model output until the snapshot time, `T`. At the snapshot time, the algorithm computes the inputs for the linear model block of the Hammerstein-Wiener model (`LinearModel` property of `the idnlhw` object) by transforming the given inputs using the input nonlinearity: $w = f(u)$. `findop` uses the resulting w to compute `x` until the snapshot time using the following equation: $x(t+1) = Ax(t) + Bw(t)$, where `[A,B,C,D] = ssdata(model.LinearModel)`.

Note For snapshot-based computations, `findop` does not perform numerical optimization.

Examples

In this example, you compute the operating point of an `idnlhw` model for a steady-state input level of 1.

1 Estimate an `idnlhw` model from sample data `iddata2`.

```
load iddata2;
M = nlhw(z2,[4 3 2],'wavenet','pwl');
```

2 Compute the steady-state operating point for an input level of 1.

```
x0 =  findop(M,'steady',1,NaN)
```

The resulting operating-point state values are, as follows:

```
x0 =

  4.309851305460448
 -4.468881971945637
  1.242120308173632
 -0.085288756364965
  0.613281249999998
```

See Also

`findstates(idnlhw)`

`operspec(idnlhw)`

`sim(idnlhw)`

Purpose

Estimate initial states of linear model from data

Syntax

```
X0 = findstates(MODEL,DATA)
X0 = findstates(MODEL,DATA,INIT)
```

Description

`X0 = findstates(MODEL,DATA)` estimates the initial states of `MODEL` that provide the best fit to output signal in `DATA`.

`X0 = findstates(MODEL,DATA,INIT)` specifies how the initial states should be estimated using the flag `INIT`.

Input

- `MODEL`: `idmodel` object. If `MODEL` is not in state-space form, initial states must be interpreted as state values corresponding to `idss(MODEL)`.
- `DATA`: `iddata` object with matching input/output dimensions.
- `INIT`: Flag indicating how the initial states should be estimated. This flag can have the following values:
  - `'e'`: (Default) Estimate initial state so that the norm of prediction error is minimized.
  - `'d'`: (Only available for discrete-time models) Same as `'e'`, but if `MODEL.InputDelay` is non-zero, these delays are first converted to explicit model delays, and the extra initial states (those corresponding to the delays) are also estimated and returned.

Input

- `X0`: Estimated initial state vector corresponding to time `DATA.TStart`. For multi-experiment data, `X0` is a matrix with as many columns as there are experiments.

Examples

In this example you estimate an `idpoly` model and simulate it such that the response of the estimated model matches the estimation data's output signal as closely as possible.

1 Load sample data.

```
load iddata1 % estimation data z1;
```

2 Estimate the linear model from data.

```
model = arx(z1, [2 2 1]); % idpoly model
```

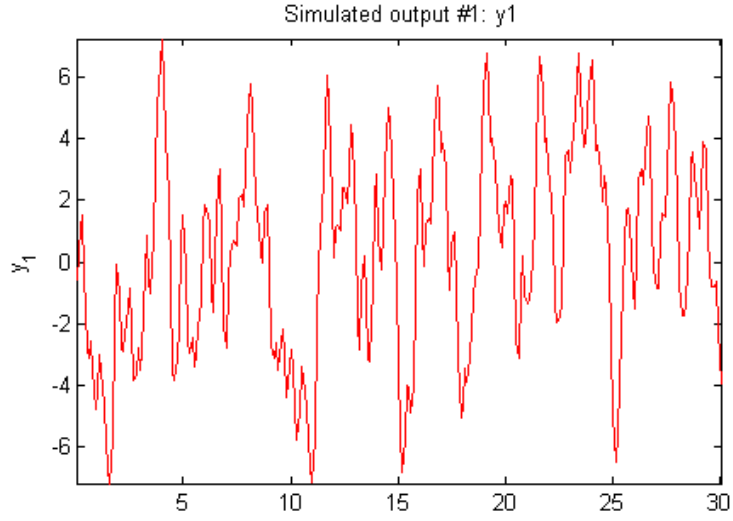
3 Estimate the value of the initial states to best fit the data.

```
x0est = findstates(model, z1);
```

4 Simulate the model.

```
sim(model, z1.u, `init', x0est)
```

MATLAB® software responds with the following plot of the simulated response.



See Also

`compare`

`pe`

`sim`

findstates(idnlarx)

Purpose

Estimate initial states of nonlinear ARX model from data

Syntax

```
XO = findstates(MODEL,DATA)
XO = findstates(MODEL,DATA,XOINIT)
XO = findstates(MODEL,DATA,XOINIT,PRED_OR_SIM)
XO = findstates(MODEL,DATA,XOINIT,PRED_OR_SIM,PVPairs)
[XO, REPORT] = findstates(...)
```

Description

`XO = findstates(MODEL,DATA)` estimates the initial states of an `idnlarx` model that minimize the error between the output measurements in `DATA` and the predicted output of the model. The states of an `idnlarx` model are defined as the delayed samples of input and output variables. For more information about the definition of states for `idnlarx` models, see "Definition of idnlarx States" on page 2-184.

`XO = findstates(MODEL,DATA,XOINIT)` specifies an initial guess for estimating the initial states.

`XO = findstates(MODEL,DATA,XOINIT,PRED_OR_SIM)` allows switching between prediction-error (default) and simulation-error minimization.

`XO = findstates(MODEL,DATA,XOINIT,PRED_OR_SIM,PVPairs)` lets you specify the algorithm properties that control the numerical optimization process as property-value pairs.

`[XO, REPORT] = findstates(...)` creates a report to summarize results of numerical optimization that is performed to search for the model states.

Input

- `MODEL`: `idnlarx` model.
- `DATA`: `iddata` object from which to estimate the initial states of `MODEL`.
- `XOINIT`: Initial guess for value of `XO`. Must be a vector of length equal to the number of the states of `MODEL` (`sum(getDelayInfo(MODEL))`).
- `PRED_OR_SIM`: Specifies minimization criteria using one of the following values:

  - `'prediction'`: (Default) Estimation of initial states by minimizing the difference between the measured output data and 1-step-ahead predicted response of the model.
  - `'simulation'`: Estimation of initial states by minimizing the difference between the measured output and the simulated response of the model. This estimation algorithm can be slower than `'prediction'`.

- `PVPairs`: Property-value pairs that specify the algorithm properties that control numerical optimization process. By default, algorithm properties are read from the `Algorithm` property of `MODEL`. You can override `MODEL.Algorithm` properties using property-value pairs. For example you might set `SearchMethod`, `MaxSize`, `Tolerance`, and `Trace`.

Input

- `X0`: Estimated initial state vector corresponding to time `DATA.TStart`. For multi-experiment data, `X0` is a matrix with as many columns as there are experiments.
- `REPORT`: Structure containing the following fields:
  - `'EstimationCriterion'`: String containing the minimization method used.
  - `'SearchMethod'`: String indicating the value of the `SearchMethod` property of `MODEL.Algorithm`.
  - `'WhyStop'`: String describing why the estimation was stopped.
  - `'Iterations'`: Number of iterations carried out during estimation.
  - `'FinalCost'`: The final value of the sum of squared errors that the search method attempts to minimize
  - `'FirstOrderOptimality'`: Measure of the gradient of the search direction at the final value of the parameter set when the search algorithm terminates. It is equal to the ∞-norm of the gradient vector.

Examples

Estimating Initial States

In this example, you use sample data `z1` to create a nonlinear ARX model. You use `findstates` to compute the initial states of the model such that the difference between the predicted output of the model and the output data in `z2` is minimized.

1 Load the sample data and create two data objects `z1` and `z2`.

```
load twotankdata
% Create data objects z1 and z2.
z = iddata(y,u,0.2,'Name','Two tank system');
z1 = z(1:1000); z2 = z(1001:2000);
```

2 Estimate the `idnlarx` model.

```
% Estimate a nonlinear ARX model from data in z1.
mw1 = nlarx(z1,[5 1 3],wavenet('NumberOfUnits',8));
```

3 Estimate the initial states of the model.

```
% Find the initial states X0 of mw1 that minimize
% the error between the output data of z2 and the
% simulated output of mw1.
X0 = findstates(mw1,z2,[],'sim')
```

This command computes:

```
X0 =
   0.446974971029569
   0.477047330267726
   0.345177563976721
   0.499168535130508
   0.275499832553790
   0.459405075998746
   0.453389709178942
   0.457206483874631
```

Estimating Initial States for Multiple-Experiment Data

In this example, you estimate the initial states for each data set in a multiple-experiment data object.

1 Create a multi-experiment data set from z1 and z2:

```
% Create a multi-experiment data set.
zm = merge(z1,z2);
```

2 Estimate the initial states for each experiment in the data set.

```
% Estimate initial states for each data set in zm.
X0 = findstates(mw1,zm)
```

This command computes the following initial states:

```
X0 =

   0.094132178358474   0.481501876919437
   0.176819996131956   0.421290990844905
   0.147547321715597   0.409770929782377
   0.166554849587273   0.408028392383016
   0.279390454813027   0.311475382142399
   0.075395497688180   0.452622735314260
   0.079227662461156   0.451653009299017
   0.079071040382789   0.451726254742998
```

See Also

`data2state(idnlarx)`

`getDelayInfo(idnlarx)`

`findop(idnlarx)`

`findstates(idmodel)`

`findstates(idnlhw)`

findstates(idnlgrey)

Purpose

Estimate initial states of nonlinear Grey-Box model from data

Syntax

```
X0 = findstates(NLSYS,DATA);
[X0,ESTINFO] = findstates(NLSYS,DATA);
[X0,ESTINFO] = findstates(NLSYS,DATA,X0INIT);
```

Description

`X0 = findstates(NLSYS,DATA);` estimates the initial states of an `idnlgrey` model from given data. For more information about the states of `idnlgrey` models, see "Definition of idnlgrey States" on page 2-205.

`[X0,ESTINFO] = findstates(NLSYS,DATA);` returns basic information about the estimation.

`[X0,ESTINFO] = findstates(NLSYS,DATA,X0INIT);` specifies an initial guess for `X0`.

Input

- `NLSYS`: `idnlgrey` model whose output is to be predicted.
- `DATA`: Input/output data `DATA = [Y U]`, where `U` and `Y` are the following:
  - `U`: Input data that can be given either as an `iddata` object or as a matrix `U = [U1 U2 ...Um]`, where the k^{th} column vector is input `Uk`
  - `Y`: Either an `iddata` object or a matrix of outputs (with as many columns as there are outputs).

Note For time-continuous `idnlgrey` models, `DATA` passed as a matrix will cause the data sample interval *Ts* to be assumed to be equal to 1.

- `X0INIT`: Initial state strategy to use:
  - `'zero'`: Use zero initial state and estimate all states (`NLSYS.InitialStates.Fixed` is thus ignored). Notice that all states are estimated, whereas they are fixed in `predict`.

- `'estimate'`: `NLSYS.InitialStates` determines the values of the states, but all initial states are estimated (`NLSYS.InitialStates.Fixed` is thus ignored).
- `'model'`: (Default) `NLSYS.InitialStates` determines the values of the initial states, which initial states to estimate, as well as their maximum and minimum values.
- vector/matrix: Column vector of appropriate length to be used as an initial guess for initial states. For multiple experiment `DATA`, `XOINIT` may be a matrix whose columns give different initial states for each experiment. With this option, all initial states are estimated (and not fixed as in `predict`) (`NLSYS.InitialStates.Fixed` is thus ignored).
- struct array: `Nx`-by-1 structure array with fields:
  - `Name`: Name of the state (a string).
  - `Unit`: Unit of the state (a string).
  - `Value`: Value of the states (a finite real 1-by-`Ne` vector, where `Ne` is the number of experiments).
  - `Minimum`: Minimum values of the states (a real 1-by-`Ne` vector or a real scalar, in which case all initial states have the same minimum value).
  - `Maximum`: Maximum values of the states (a real 1-by-`Ne` vector or a real scalar, in which case all initial states have the same maximum value).
  - `Fixed`: Boolean 1-by-`Ne` vector, or a scalar Boolean (applicable for all states) specifying whether the initial state is fixed or not.

Input

- `XO`: Matrix containing the initial states. In the single experiment case it is a column vector of length `Nx`. For multi-experiment data, `XO` is a matrix with as many columns as there are experiments.
- `ESTINFO`: Structure or `Ne`-by-1 structure array containing basic information about the estimation result (some of the fields normally stored in `NLSYS.EstimationInfo`). For multi-experiment data,

ESTINFO is an Ne-by-1 structure array with elements providing initial state estimation information related to each experiment.

Examples

Estimating Individual Initial States Selectively

In this example you estimate the initial states of a model selectively, fixing the first state and allowing the second state of the model to be estimated. First you create a model from sample data and set the Fixed property of the model such that the second state is free and the first is fixed.

1 Specify the file, model orders and model parameters.

```
% File describing the model structure.
FileName      = 'dcmotor_m';
Order         = [2 1 2]; % Model orders [ny nu nx].

% Model's parameters (see idnlgreydemo1 for information)
Parameters    = [0.24365; 0.24964];
```

2 Estimate the idnlgrey model and set the model properties.

```
nlgr = idnlgrey(FileName, Order, Parameters);
set(nlgr, 'InputName', 'Voltage','OutputName', ...
  {'Angular position', 'Angular velocity'});
```

3 Free the second state while keeping the first one fixed.

```
setinit(nlgr,'Fixed',{1 0});
```

4 Load the estimation data.

```
load(fullfile(matlabroot, 'toolbox', 'ident', ...
     'iddemos', 'data', 'dcmotordata'));
z = iddata(y, u, 0.1, 'Name', 'DC-motor','InputName', 'Voltage',...
    'OutputName', {'Angular position', 'Angular velocity'});
```

5 Estimate the free states of the model.

```
% Default calling syntax. The same as with
% 'model' as third argument.
[X0, EstInfo] = findstates(nlgr, z)
```

This command computes:

```
X0 =
            0
      -0.0179
EstInfo =
        Method: 'Solver: ode45; Search: lsqnonlin'
      DataName: 'DC-motor, Exp1'
       WhyStop: 'Near (local) minimum, (norm(g) < tol)'
    Iterations: 1
```

Estimating Initial States Starting from States Stored in Model

This example shows how you can also estimate all initial states, starting from an initial state of `0`, then again from the initial states stored in `nlgr`.

1. Estimate all the initial states starting from 0.

   ```
   X0 = findstates(nlgr, z, 'zero');
   ```

2. Estimate the free initial states specified by `nlgr` starting from the initial state stored in `nlgr`. The fixed/free setting of initial states as specified in `nlgr` are honored.

   ```
   X0 = findstates(nlgr, z, 'mod');
   ```

3. Estimate all the initial states, starting from an initial state vector that you specify.

   ```
   nlgr.Algorithm.Trace = 'full';

   % Starting from an initial state vector [10;10]
   X0 = findstates(nlgr,z,[10;10])
   ```

Advanced Use of findstates(idnlgrey)

The following example shows advanced use of `findstates`. Here you estimate states for multi-experiment data, such that the states of model `nlgr` are estimated separately for each experiment. After creating a 3-experiment data set `z3`, you estimate individual initial states separately.

1 Create a three-experiment data set.

```
z3 = merge(z, z, z); % 3-experiment data
```

2 Fix some initial states and only estimate the free initial states starting of with the initial state in `nlgr`. This means that both elements of state vector 1 will be estimated, that no state of the second state vector will be estimated, and that only the first state of state vector 3 is estimated.

```
% prepare model for 3-experiment data
nlgr = pem(z3, nlgr, 'Trace', 'off');
```

3 Specify which initial states to fix, and set the `Trace` property of `Algorithm` to `'full'`.

```
nlgr.InitialStates(1).Fixed = [true false true];
nlgr.InitialStates(2).Fixed = [true false false];
nlgr.Algorithm.Trace = 'full';
```

4 Estimate the initial states and obtain information about the estimation.

```
[X0, EstInfo] = findstates(nlgr, z3);
```

See Also

`findstates(idnlarx)`

`findstates(idnlhw)`

```
predict
sim
```

findstates(idnlhw)

Purpose

Estimate initial states of nonlinear Hammerstein-Wiener model from data

Syntax

```
X0 = findstates(MODEL,DATA)
X0 = findstates(MODEL,DATA,XOINIT)
X0 = findstates(MODEL,DATA,XOINIT,PVPairs)
[X0, REPORT] = findstates(...)
```

Description

`X0 = findstates(MODEL,DATA)` estimates the initial states of an `idnlhw` model from given data. The states of an `idnlhw` model are defined as the states of its embedded linear model (`Model.LinearModel`). For more information about the states of `idnlhw` models, see "Definition of idnlhw States" on page 2-219.

`X0 = findstates(MODEL,DATA,XOINIT)` specifies an initial guess for value of `X0` using `XOINIT`.

`X0 = findstates(MODEL,DATA,XOINIT,PVPairs)` specifies property-value pairs representing the algorithm properties that control the numerical optimization process.

`[X0, REPORT] = findstates(...)` creates a report to summarize results of numerical optimization that is performed to search for the model states.

Input

- `MODEL`: `idnlhw` model.
- `DATA`: `iddata` object from which to estimate the initial states of `MODEL`.
- `XOINIT`: Initial guess for value of `X0`. Must be a vector of length equal to the number of the states of `MODEL`.
- `PVPairs`: Property-value pairs that specify the algorithm properties that control numerical optimization process. By default, algorithm properties are read from the `Algorithm` property of `MODEL`. You can override `MODEL.Algorithm` properties using property-value pairs. For example you might set `SearchMethod`, `MaxSize`, `Tolerance`, and `Trace`.

Input

- `X0`: Estimated initial state vector corresponding to time `DATA.TStart`. For multi-experiment data, `X0` is a matrix with as many columns as there are experiments.
- `REPORT`: Structure containing the following fields:
  - `'EstimationCriterion'`: String containing the minimization method used.
  - `'SearchMethod'`: String indicating the value of the `SearchMethod` property of `MODEL.Algorithm`.
  - `'WhyStop'`: String describing why the estimation was stopped.
  - `'Iterations'`: Number of iterations carried out during estimation.
  - `'FinalCost'`: The final value of the sum of squared errors that the search method attempts to minimize
  - `'FirstOrderOptimality'`: Measure of the gradient of the search direction at the final value of the parameter set when the search algorithm terminates. It is equal to the ∞-norm of the gradient vector.

Examples

In this example, you create an `idnlarx` model from sample data and estimate initial states using another data set. Next you jointly estimate the states for separate data sets contained in multi-experiment data.

1 Load the data and create `iddata` objects `z1` and `z2`.

```
load twotankdata

z = iddata(y, u, 0.2,'Name','Two tank system');
z1 = z(1:1000); z2 = z(1001:2000);
```

2 Estimate an `idnlhw` model from data.

```
m1=nlhw(z1,[4 2 1], 'unitgain' , 'pwlinear')
```

3 Estimate the initial states of `m1` using data `z2`.

```
% Estimate initial states. View estimation trace and use
% only 5 iterations in the search algorithm
X0 = findstates(m1, z2, [], 'MaxIter', 5, 'Trace', 'on')
```

4 Estimate states using multiple-experiment data. There are separate sets of initial states for each experiment. The states of all data experiments are jointly estimated, and X0 is returned as a matrix with as many columns as there are data experiments.

```
zm = merge(z1,z2);
X0 = findstates(m1, zm)
```

See Also

`findstates(idnlarx)`

`findstates(idmodel)`

`findop(idnlhw)`

Purpose

Convert `idfrd` objects to Control System Toolbox™ frequency-response LTI model

Syntax

```
sys = frd(mod)
```

Description

`mod` is an `idfrd` object. `sys` is returned as an `frd` object.

The fields `Frequency`, `ResponseData`, `Units`, `Ts`, `InputDelay`, `InputName`, `OutputName` and `Notes` in `mod` are transferred to `sys`. The remaining fields (`SpectrumData`, `CovarianceData` and `NoiseCovariance`) are ignored. The command, therefore, cannot be applied to a time-series `idfrd` model object.

See Also

`ss`

`tf`

`zpk`

Purpose

Frequency-response data from idmodel or idfrd object

Syntax

```
H = freqresp(m)
[H,w,covH] = freqresp(m,w)
```

Description

m is any idmodel or idfrd object.

H = freqresp(m,w) computes the frequency response H of the idmodel model m at the frequencies specified by the vector w. These frequencies should be real and in rad/s.

If m has ny outputs and nu inputs, and w contains Nw frequencies, the output H is an ny-by-nu-by-Nw array such that H(:,:,k) gives the complex-valued response at the frequency w(k).

For a SISO model, specify H(:) to obtain a vector of the frequency response.

If w is not specified, a default choice is made based on the dynamics of the model.

Input

```
[H,w,covH] = freqresp(M,w)
```

also returns the frequencies w and the covariance covH of the response. covH is a 5-D array where covH(ky,ku,k,:,:) is the 2-by-2 covariance matrix of the response from input ku to output ky at frequency w(k). The 1,1 element is the variance of the real part, the 2,2 element is the variance of the imaginary part, and the 1,2 and 2,1 elements are the covariance between the real and imaginary parts. squeeze(covH(ky,ku,k,:,:)) gives the covariance matrix of the corresponding response.

If m is a time series (no input channels), H is returned as the (power) spectrum of the outputs, an ny-by-ny-by-Nw array. Hence H(:,:,k) is the spectrum matrix at frequency w(k). The element H(k1,k2,k) is the cross spectrum between outputs k1 and k2 at frequency w(k). When k1 = k2, this is the real-valued power spectrum of output k1.

`covH` is then the covariance of the estimated spectrum `H`, so that `covH(k1,k1,k)` is the variance of the power spectrum estimate of output `k1` at frequency `W(k)`. No information about the variance of the cross spectra is normally given; that is, `covH(k1,k2,k) = 0` for `k1` is not equal to `k2`)

If the model `m` is not a time series, use `freqresp(m('n'))` to obtain the spectrum information of the noise (output disturbance) signals.

Note that `idfrd` computes the same information as `freqresp`, and stores it in the `idfrd` object.

See Also

`bode`

`etfe`

`ffplot`

`idfrd`

`nyquist`

`spa`

`spafdr`

Purpose

Akaike Final Prediction Error for estimated model

Syntax

```
fp = fpe(Model1,Model2,Model3,...)
```

Description

`Model` is any estimated `idmodel` (`idarx`, `idgrey`, `idpoly`, `idproc`, `idss`).

`fp` is returned as a row vector containing the values of the Akaike Final Prediction Error (FPE) for the different models. This is defined as

$$FPE = V\left(\frac{1+d/N}{1-d/N}\right)$$

where V is the loss function, d is the number of estimated parameters, and N is the number of estimation data.

The loss function V is

$$V = \det\left(\frac{1}{N}\sum_{1}^{N}\varepsilon\left(t,\theta_N\right)\left(\varepsilon\left(t,\theta_N\right)\right)^T\right)$$

where $\hat{\theta}_N$ represents the estimated parameters.

FPE can be negative when the number of estimated parameters exceeds the number of data samples, which can occur for models with multiple outputs. For models with multiple output, the assumption that d/N is small is not valid. In when this assumption is not valid, use AIC instead.

References

Sections 7.4 and 16.4 in Ljung (1999).

See Also

`EstimationInfo`

`aic`

Purpose

Frequencies from idfrd object

Syntax

```
idfm = fselect(idf,index)
idfm = fselect(idf,Fmin,Fmax)
```

Description

idf is any idfrd object. index is a row vector of frequency indices, so that idfm is the idfrd object that contains the response at frequencies idf.Frequency(Index).

If Fmin and Fmax are specified, idfm contains responses at frequencies between Fmin and Fmax.

Note that the operation is the same as dat(index) for an iddata object.

Examples

Select every fifth frequency:

```
idfm = fselect(idf,5:5:100)
```

Select the response in the third quadrant:

```
ph = angle(squeeze(idf.response));
idfm = fselect(idf,find(ph>-pi & ph <-pi/2))
```

See Also

fcat

iddata

idfrd

Purpose

Query properties of data and model objects

Syntax

```
Value = get(m,'PropertyName')
get(m)
Struct = get(m)
```

Description

`value = get(m,'PropertyName')` returns the current value of the property `PropertyName` of the `iddata` object or `idfrd` object, or `idmodel` object (`idgrey`, `idarx`, `idpoly`, `idss`), or `idnlgrey`, `idnlarx`, or `idnlhw` model object.

The string `'PropertyName'` can be the full property name (for example, `'SSParameterization'`) or any unambiguous case-insensitive abbreviation (for example, `'ss'`).

`Struct = get(m)` converts the object `m` into a standard MATLAB® structure with the property names as field names and the property values as field values.

Without a left-hand argument

```
get(m)
```

displays all properties of `m` and their values.

Remarks

An alternative to the syntax

```
Value = get(m,'PropertyName')
```

is the structure-like referencing

```
Value = m.PropertyName
```

See Also

`Algorithm Properties`

`idarx`

`idfrd`

idgrey

idnlarx

idnlgrey

idnlhw

idpoly

idproc

idss

getDelayInfo(idnlarx)

Purpose

Get input/output delay information for `idnlarx` model structure

Syntax

```
DELAYS = getDelayInfo(MODEL)
DELAYS = getDelayInfo(MODEL,TYPE)
```

Description

`DELAYS = getDelayInfo(MODEL)` obtains the maximum delay in each input and output variable of an `idnlarx` model.

`DELAYS = getDelayInfo(MODEL,TYPE)` lets you choose between obtaining maximum delays across all input and output variables or maximum delays for each output variable individually. When delays are obtained for each output variable individually a matrix is returned, where each row is a vector containing n_y+n_u maximum delays for each output variable, and:

- n_y is the number of outputs of `MODEL`.
- n_u is the number of inputs of `MODEL`.

Delay information is useful for determining the number of states in the model. For nonlinear ARX models, the states are related to the set of delayed input and output variables that define the model structure (regressors). For example, if an input or output variable p has a maximum delay of D samples, then it contributes D elements to the state vector:

$p(t\text{-}1)$, $p(t\text{-}2)$, ...$p(t\text{-}D)$

The number of states of a nonlinear ARX model equals the sum of the maximum delays of each input and output variable. For more information about the definition of states for `idnlarx` models, see "Definition of idnlarx States" on page 2-184

Input

`getDelayInfo` accepts the following arguments:

- `MODEL`: `idnlarx` model.

- `TYPE`: (Optional) Specifies whether to obtain channel delays `'channelwise'` or `'all'` as follows:
  - `'all'`: Default value. `DELAYS` contains the maximum delays across each output (vector of n_y+n_u entries, where `[ny, nu] = size(MODEL)`).
  - `'channelwise'`: `DELAYS` contains delay values separated for each output (n_y-by-(n_y+n_u) matrix).

Input

- `DELAYS`: Contains delay information in a vector of length n_y+n_u arranged with output channels preceding the input channels, i.e., `[y1, y2,.., u1, u2,..]`.

Examples

In the following example you create a 2-output, 3-input nonlinear ARX model, then verify the number of delays using `getDelayInfo(idnlarx)`.

1 Create an `idnlarx` model.

```
M = idnlarx([2 0 2 2 1 1 0 0; 1 0 1 5 0 1 1 0], 'linear');
```

2 Compute the maximum delays for each output variable individually.

```
Del = getDelayInfo(M,'channelwise')

Del =

     2     0     2     1     0
     1     0     1     5     0
```

The matrix `Del` contains the maximum delays for the first and second output of the model `M`. You can interpret the contents of matrix `Del` as follows:

- In the dynamics for the output 1 (y_1) of model `M`, the maximum delays for each input/output channel are as follows: y_1: 2, y_2: 0, u_1: 2, u_2: 1, u_3:0.

- Similarly, in the dynamics for the output 2 (y_2) of the model, the maximum delays in channels y_1, y_2, u_1, u_2, u_3 are 1, 0, 1, 5, and 0 respectively.

You can find the maximum delays for all the input and output variables in the order (y_1, y_2, u_1, u_2, u_3) by executing the command

```
Del=getDelayInfo(M, 'all')
```

which returns

```
Del =

     2     0     2     5     0
```

Note The maximum delay across all output equations can be obtained by executing `MaxDel = max(Del,[],1)`. Since input u_2 has 5 delays (the 4th entry in `Del`, there are 5 terms corresponding to u_5 in the state vector (($u_5(t\text{-}1)$, ...$u_5(t\text{-}5)$. Applying this definition to all I/O channels, the complete state vector for model `M` becomes:

$X(t)$ = [$y_1(t\text{-}1)$, $y_1(t\text{-}2)$, $u_1(t\text{-}1)$, $u_1(t\text{-}2)$, $u_2(t\text{-}1)$, $u_2(t\text{-}2)$, $u_2(t\text{-}3)$, $u_2(t\text{-}4)$, $u_2(t\text{-}5)$]

See Also

`data2state(idnlarx)`

`getreg`

`idnlarx`

Purpose

Specific experiment(s) from multiple-experiment `iddata` object

Syntax

```
d1 = getexp(data,ExperimentNumber)
d1 = getexp(data,ExperimentName)
```

Description

`data` is an `iddata` object that contains several experiments. `d1` is another `iddata` object containing the indicated experiment(s). The reference can either be by `ExperimentNumber`, as in `d1 = getexp(data,3)` or `d1 = getexp(data,[4 2])`; or by `ExperimentName`, as in `d1 = getexp(data,'Period1')` or `d1 = getexp(data,{'Day1','Day3'})`.

See `merge (iddata)` and `iddata` for how to create multiple-experiment data objects.

You can also retrieve the experiments using a fourth subscript, as in `d1 = data(:,:,:,ExperimentNumber)`. Type `help iddata/subsref` for details on this.

getinit

Purpose

Values of idnlgrey model initial states

Syntax

```
getinit(model)
getinit(model,prop)
```

Arguments

`model`
: Name of the idnlgrey model object.

`Property`
: Name of the InitialStates model property field, such as 'Name', 'Unit', 'Value', 'Minimum', 'Maximum', and 'Fixed'.

  Default: 'Value'.

Description

`getinit(model)` gets the initial-state values in the 'Value' field of the InitialStates model property.

`getinit(model,prop)` gets the initial-state values of the prop field of the InitialStates model property. prop can be 'Name', 'Unit', 'Value', 'Minimum', 'Maximum', and 'Fixed'.

The returned values are an Nx-by-1 cell array of values, where Nx is the number of states.

See Also

getpar

idnlgrey

setinit

setpar

Purpose

Parameter values and properties of idnlgrey model parameters

Syntax

```
getpar(model)
getpar(model,prop)
```

Arguments

`model`
: Name of the idnlgrey model object.

`Property`
: Name of the Parameters model property field, such as 'Name', 'Unit', 'Value', 'Minimum', or 'Maximum'.

  Default: 'Value'.

Description

getpar(model) gets the model parameter values in the 'Value' field of the Parameters model property.

getpar(model,prop) gets the model parameter values in the prop field of the Parameters model property. prop can be 'Name', 'Unit', 'Value', 'Minimum', and 'Maximum'.

The returned values are an Np-by-1 cell array of values, where Np is the number of parameters.

See Also

getinit

idnlgrey

setinit

setpar

Purpose

Names of standard or custom regressors in nonlinear ARX model

Syntax

```
getreg(model)
getreg(model,subset)
R = getreg(model,subset)
```

Arguments

`model`
: Name of the `idnlarx` model object.

`subset`
: Has one of the following values: `'all'`, `'input'`, `'output'`, `'standard'`, `'custom'`, `'linear'`, and `'nonlinear'`.

Description

`getreg(model)` returns the regressor expressions for all of the regressors in the nonlinear ARX model.

`getreg(model,subset)` returns the regressor expressions for a specified subset of the regressors, as follows:

- `'all'` — All regressors.
- `'input'` — Input regressors only.
- `'output'` — Output regressors only.
- `'standard'` — Standard regressors only.
- `'custom'` — Custom regressors only.
- `'linear'` — Regressors that are not used in the in the nonlinear block.
- `'nonlinear'` — Regressors used in the nonlinear block.

Note You can use `'nl'` as an abbreviation for `'nonlinear'`.

`R = getreg(model,subset)` returns a cell array of strings of the regressors for a specified subset of the regressors. For multiple-output, returns a cell array of cell arrays.

See Also

`addreg`

`customreg`

`nlarx`

`polyreg`

Purpose

Class for storing multiple-output ARX polynomials and estimated impulse- and step-response models

Syntax

```
m = idarx(A,B,Ts)
m = idarx(A,B,Ts,'Property1',Value1,...,,'PropertyN',ValueN)
```

Description

`idarx` creates an object containing parameters that describe the general multiple-input, multiple-output model structure of ARX type.

$$y(t) + A_1 y(t-1) + A_2 y(t-2) + \ldots + A_{na} y(t-na) =$$

$$B_0 u(t) + B_1 u(t-1) + \ldots + B_{nb} u(t-nb) + e(t)$$

Here A_k and B_k are matrices of dimensions *ny*-by-*ny* and *ny*-by-*nu*, respectively. (*ny* is the number of outputs, that is, the dimension of the vector $y(t)$, and *nu* is the number of inputs.)

The arguments `A` and `B` are 3-D arrays that contain the `A` matrices and the `B` matrices of the model in the following way.

`A` is an `ny`-by-`ny`-by-`(na+1)` array such that:

```
A(:,:,k+1) = Ak
A(:,:,1) = eye(ny)
```

Similarly `B` is an `ny`-by-`nu`-by-`(nb+1)` array with:

```
B(:,:,k+1) = Bk
```

Note that `A` always starts with the identity matrix, and that delays in the model are defined by setting the corresponding leading entries in `B` to zero. For a multivariate time series, take `B = []`.

The optional property `NoiseVariance` sets the covariance matrix of the driving noise source $e(t)$ in the model above. The default value is the identity matrix.

The argument `Ts` is the sampling interval. Note that continuous-time models (`Ts = 0`) are not supported.

The use of idarx is twofold. You can use it to create models that are simulated (using sim) or analyzed (using bode, pzmap, etc.). You can also use it to define initial value models that are further adjusted to data (using arx). The free parameters in the structure are consistent with the structure of A and B; that is, leading zeros in the rows of B are regarded as fixed delays, and trailing zeros in A and B are regarded as a definition of lower-order polynomials. These zeros are fixed, while all other parameters are free.

For a model with one output, ARX models can be described both as idarx and idpoly models. The internal representation is different, however.

idarx Properties

- A, B: The A and B polynomials as 3-D arrays, described above.
- dA, dB: The standard deviations of A and B. Same format as A and B. Cannot be set.
- na, nb, nk: The orders and delays of the model. na is an ny-by-ny matrix whose *i-j* entry is the order of the polynomial corresponding to the *i-j* entry of A. Similarly nb is an ny-by-nu matrix with the orders of B. nk is also an ny-by-nu matrix, whose *i-j* entry is the delay from input *j* to output *i*, that is, the number of leading zeros in the *i-j* entry of B.
- InitialState: This describes how the initial state (initial values in filtering, etc.) should be handled. For time-domain applications, this is typically handled by starting the filtering when all data are available. For frequency-domain data, you must estimate initial states. The possible values of InitialState are 'zero', 'estimate', and 'auto' (which makes a data-dependent choice between zero and estimate).

In addition to these properties, idarx objects also have all the properties of the idmodel object. See idmodel, Algorithm Properties, and EstimationInfo.

Note that you can set and retrieve all properties either with the `set` and `get` commands or by subscripts. Autofill applies to all properties and values, and they are case insensitive.

For a complete list of property values, use `get(m)`. To see possible value assignments, use `set(m)`. See also `idprops idarx`.

idarx Definition of States

The states of an `idarx` model are defined as those corresponding to the model obtained by converting them to the state-space format using the `idss` command. For example, if you have an `idarx` model defined by `m1 = idarx(A,B,1)`, then the initial states of this model correspond to those of `m2 = idss(m1)`. The concept of states is useful for functions such as `sim`, `predict`, `compare` and `findstates`.

Examples

Simulate a second-order ARX model with one input and two outputs, and then estimate a model using the simulated data.

```
A = zeros(2,2,3);
B = zeros(2,1,3)
A(:,:,1) =eye(2);
A(:,:,2) = [-1.5 0.1;-0.2 1.5];
A(:,:,3) = [0.7 -0.3;0.1 0.7];
B(:,:,2) = [1;-1];
B(:,:,3) = [0.5;1.2];
m0 = idarx(A,B,1);
u = iddata([],idinput(300));
e = iddata([],randn(300,2));
y = sim(m0,[u e]);
m = arx([y u],[[2 2;2 2],[2;2],[1;1]]);
```

See Also

`Algorithm Properties`

`arx`

`arxdata`

`EstimationInfo`

`idmodel`

`idpoly`

Purpose

Class for storing time-domain and frequency-domain data

Syntax

```
data = iddata(y,[],Ts)
data = iddata(y,u,Ts)
data = iddata(y,u,Ts,'Frequency',W)
data = iddata(y,u,Ts,'P1',V1,...,'PN',VN)
data = iddata(idfrd_object)
```

Arguments

`y`

Name of MATLAB® variable that represents the output signal from the dynamic system. Sets the `OutputData` `iddata` property. For a single-output system, this is a column vector. For a multiple-output system with N_y output channels and N_T time samples, this is an N_T-by-N_y matrix.

Note Output data must be in the same domain as input data.

`u`

Name of MATLAB variable that represents the input signal for the dynamic system. Sets the `InputData` `iddata` property. For a single-input system, this is a column vector. For a multiple-output system with N_u output channels and N_T time samples, this is an N_T-by-N_u matrix.

Note Input data must be in the same domain as output data.

`Ts`

Time interval between successive data samples in seconds. Default value is `1`. For continuous-time data in the frequency domain, set `Ts` to `0`.

`'P1',V1,...,'PN',VN`

Pairs of `iddata` property names and property values.

`idfrd_object`
Name of `idfrd` data object.

Description

`data = iddata(y,[],Ts)` creates an `iddata` object for time-series data, containing a time-domain output signal `y` and an empty input signal `[]`, respectively. `Ts` specifies the sampling interval of the experimental data.

`data = iddata(y,u,Ts)` creates an `iddata` object containing a time-domain output signal `y` and input signal `u`, respectively. `Ts` specifies the sampling interval of the experimental data.

`data = iddata(y,u,Ts,'Frequency',W)` creates an `iddata` object containing a frequency-domain output signaly and input signal `u`, respectively.`Ts` specifies the sampling interval of the experimental data. `W` specifies the `iddata` property `'frequency'` as a vector of frequencies.

`data = iddata(y,u,Ts,'P1',V1,...,'PN',VN)` creates an `iddata` object containing a time-domain or frequency-domain output signal `y` and input signal `u`, respectively. `Ts` specifies the sampling interval of the experimental data. `'P1',V1,...,'PN',VN` are property-value pairs, as described in "iddata Properties" on page 2-137.

`data = iddata(idfrd_object)` transforms an `idfrd` object to a frequency-domain `iddata` object.

iddata Properties

The following table describes `iddata` object properties and their values. These properties are specified as property-value arguments `'P1',V1,...,'PN',VN'` in the `iddata` constructor, or you can set them using the `set` command or dot notation. In the list below, `N` denotes the number of data samples in the input and output signals, `ny` is the number of output channels, `nu` is the number of input channels, and `Ne` is the number of experiments.

Tip Property names are not case sensitive. You do not need to type the entire property name. However, the portion you enter must by enough to uniquely identify the property.

Property Name	Description	Value
`Domain`	Specifies whether the data is in the time domain or frequency domain.	• `'Frequency'` — Frequency-domain data. • `'Time'` (Default) — Time-domain data.
`ExperimentName`	Name of each data set contained in the `iddata` object.	For `Ne` experiments, a `1-by-Ne` cell array of strings. Each cell contains the name of the corresponding experiment. Default names are `{'Exp1', 'Exp2',...}`.
`Frequency`	(Frequency-domain data only) Frequency values for defining the Fourier Transforms of the signals.	For a single experiment, this is an `N-by-1` vector. For `Ne` experiments, a `1-by-Ne` cell array and each cell contains the frequencies of the corresponding experiment.
`InputData`	Name of MATLAB variable that stores the input signal for the dynamic system.	For `nu` input channels and `N` data samples, this is an `N-by-nu` matrix.
`InputName`	Specifies the names of individual input channels.	Cell array of length `nu-by-1` contains the name string of each input channel. Default names are `{'u1';'u2';...}`.
`InputUnit`	Specifies the units of each input channel.	Cell array of length `nu-by-1`. Each cell contains a string that specifies the units of each input channel.

Property Name	Description	Value
`InterSample`	Specifies the behavior of the input signals between samples for transformations between discrete-time and continuous-time.	For a single experiment: • `zoh`— (Default) Zero-order hold maintains a piecewise-contant input signal between samples. • `foh`— First-order hold maintains a piecewise-linear input signal between samples. • `bl`— Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency. For `Ne` experiments, `InterSample` is an `nu`-by-`Ne` cell array. Each cell contains one of these values corresponding to each experiment.
`Name`	Name of the data set.	Text string.
`Notes`	Comments about the data set.	Text string.
`OutputData`	Name of MATLAB variable that stores the output signal from the dynamic system.	For `ny` output channels and `N` samples, this is an `N`-by-`ny` matrix.

Property Name	Description	Value
`OutputName`	For a multiple-output system, specifies the names of individual output channels.	Cell array of length `ny`-by-`1` contains the name string of each output channel. Default names are `{'y1';'y2';...}`.
`OutputUnit`	Specifies the units of each output channel.	For `ny` output channels, a cell array of length `ny`-by-`1`. Each cell contains a string that specifies the units of the corresponding output channel.
`Period`	Period of the input signal.	(Default) For a nonperiodic signal, set to `inf`. For a multiple-input signal, this is an `nu`-by-`1` vector and the `k`th entry contains the period of the `k`th input. For `Ne` experiments, this is a `1`-by-`Ne` cell array and each cell contains a scalar or vector of periods for the corresponding experiment.
`SamplingInstants`	(Time-domain data only) The time values in the time vector calculated from the properties `Tstart` and `Ts`.	For a single experiment, this is an `N`-by-`1` vector. For `Ne` experiments, this is a `1`-by-`Ne` cell array and each cell contains the sampling instants of the corresponding experiment.
`TimeUnit`	(Time-domain data only) Time unit.	A string that specifies the time unit for the time vector.

Property Name	Description	Value
`Ts`	Time interval between successive data samples in seconds. Must be specified for both time- and frequency-domain data. For frequency-domain, it is used to compute Fourier transforms of the signals as discrete-time Fourier transforms (DTFT) with the indicated sampling interval. **Note** Your data must be uniformly sampled.	Default value is `1`. For continuous-time data in the frequency domain, set to `0`; the inputs and outputs are interpreted as continuous-time Fourier transforms of the signals. Note that `Ts` is essential also for frequency-domain data, for proper interpretation of how the Fourier transforms were computed: They are interpreted as discrete-time Fourier transforms (DTFT) with the indicated sampling interval.. For multiple-experiment data, `Ts` is a `1-by-Ne` cell array and each cell contains the sampling interval of the corresponding experiment.
`Tstart`	(Time-domain data only) Specifies the start time of the time vector.	For a single experiment, this is a scalar. For `Ne` experiments, `Ts` is a `1-by-Ne` cell array and each cell contains the sampling interval of the corresponding experiment.

Property Name	Description	Value
`Units`	(Frequency-domain data only) Frequency unit.	Specified as `rad/s` or `Hz`.For multiexperiement data with `Ne` experiments, `Units` is a `1-by-Ne` cell array and each cell contains the frequency unit for each experiment.
`UserData`	Additional comments.	Text string.

See Also

`advice`
`detrend`
`fcat`
`getexp`
`idfilt`
`idfrd`
`plot`
`resample`
`size`

Purpose

Open System Identification Tool GUI

Syntax

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

Description

`ident` by itself opens the main interface window, or brings it forward if it is already open.

`session` is the name of a previous session with the graphical user interface, and typically has extension `.sid`. The `path` argument is the complete path for the location of this file. If the session file is on the `MATLABPATH`, `path` can be omitted.

When the session is specified, the interface will open with this session active. Typing `ident(session,path)` in the MATLAB® Command Window, when the interface is active, will load and open the session in question.

For more information about the graphical user interface, see "Customizing and Using the GUI".

Examples

```
ident('iddata1.sid')
ident('mydata.sid','\matlab\data\cdplayer\')
```

See Also

`midprefs`

Purpose

Filter data using user-defined passbands, general filters, or Butterworth filters

Syntax

```
Zf = idfilt(Z,filter)
Zf = idfilt(Z,filter,causality)
Zf = idfilt(Z,filter,'FilterOrder',NF)
```

Description

`Z` is the data, defined as an `iddata` object. `Zf` contains the filtered data as an `iddata` object. The filter can be defined in three ways:

- As an explicit system that defines the filter,

  ```
  filter = idm or filter = {num,den} or filter = {A,B,C,D}
  ```

  `idm` can be any SISO `idmodel` or LTI model object. Alternatively the filter can be defined as a cell array `{A,B,C,D}` of SISO state-space matrices or as a cell array `{num,den}` of numerator/denominator filter coefficients.

- As a vector or matrix that defines one or several passbands,

  ```
  filter=[[wp1l,wp1h];[ wp2l,wp2h]; ....;[wpnl,wpnh]]
  ```

  The matrix is `n`-by-2, where each row defines a passband in rad/s. A filter is constructed that gives the union of these passbands. For time-domain data, it is computed as cascaded Butterworth filters or order NF. The default value of NF is `5`.

  For example, to define a stopband between `ws1` and `ws2`, use

  ```
  filter = [0 ws1; ws2,Nyqf]
  ```

  where `Nyqf` is the Nyquist frequency.

- For frequency-domain data, only the frequency response of the filter can be specified:

  ```
  filter = Wf
  ```

Here `Wf` is a vector of possibly complex values that define the filter's frequency response, so that the inputs and outputs at frequency `Z.Frequency(kf)` are multiplied by `Wf(kf)`. `Wf` is a column vector of length = number of frequencies in `Z`. If the data object has several experiments, `Wf` is a cell array of length = # of experiments in `Z`.

For time-domain data, the filtering is carried out in the time domain as causal filtering as default. This corresponds to a last argument `causality = 'causal'`. With `causality = 'noncausal'`, a noncausal, zero-phase filter is used for the filtering (corresponding to `filtfilt` in the Signal Processing Toolbox™ product).

For frequency-domain data, the signals are multiplied by the frequency response of the filter. With the filters defined as passband, this gives ideal, zero-phase filtering ("brickwall filters"). Frequencies that have been assigned zero weight by the filter (outside the passband, or via the frequency response) are removed from the `iddata` object `Zf`.

It is common practice in identification to select a frequency band where the fit between model and data is concentrated. Often this corresponds to bandpass filtering with a passband over the interesting breakpoints in a Bode diagram. For identification where a disturbance model is also estimated, it is better to achieve the desired estimation result by using the property `'Focus'` (see `Algorithm Properties`) than just to prefilter the data. The proper values for `'Focus'` are the same as the argument `filter` in `idfilt`.

Algorithm

The Butterworth filter is the same as `butter` in the Signal Processing Toolbox product. Also, the zero-phase filter is equivalent to `filtfilt` in that toolbox.

References

Ljung (1999), Chapter 14.

See Also

`Algorithm Properties`

`iddata`

Purpose

Class for storing frequency-response or spectral-analysis data or frequency-response models

Syntax

```
h = idfrd(Response,Freq,Ts)
h = idfrd(Response,Freq,Ts,
   'CovarianceData',Covariance, ...
'SpectrumData',Spec,
   'NoiseCovariance',Speccov,'P1', ...
 V1,'PN',
   VN)
h = idfrd(mod)
h = idfrd(mod,Freqs)
```

Description

```
h = idfrd(Response,Freq,Ts)
h = idfrd(Response,Freq,Ts,'CovarianceData',Covariance, ...
'SpectrumData',Spec,'NoiseCovariance',Speccov,'P1', ...
 V1,'PN',VN)
h = idfrd(mod)
h = idfrd(mod,Freqs)
```

`idfrd` creates the `idfrd` model object.

For a model

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

stores the transfer function estimate G

$$G(e^{i\omega})$$

as well as the spectrum of the additive noise (Φ_v) at the output

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

where λ is the estimated variance of $e(t)$, and T is the sampling interval.

Creating idfrd from Given Responses

`Response` is a 3-D array of dimension `ny`-by-`nu`-by-`Nf`, with `ny` being the number of outputs, `nu` the number of inputs, and `Nf` the number of frequencies (that is, the length of `Freqs`). `Response(ky,ku,kf)` is thus the complex-valued frequency response from input `ku` to output `ky` at frequency `=Freqs(kf)`. When defining the response of a SISO system, `Response` can be given as a vector.

`Freqs` is a column vector of length `Nf` containing the frequencies of the response.

`Ts` is the sampling interval. `T = 0` means a continuous-time model.

`Covariance` is a 5-D array containing the covariance of the frequency response. It has dimension `ny`-by-`nu`-by-`Nf`-by-2-by-2. The structure is such that `Covariance(ky,ku,kf,:,:)` is the 2-by-2 covariance matrix of the response `Response(ky,ku,kf)`. The 1-1 element is the variance of the real part, the 2-2 element is the variance of the imaginary part, and the 1-2 and 2-1 elements are the covariance between the real and imaginary parts. `squeeze(Covariance(ky,ku,kf,:,:))` thus gives the covariance matrix of the corresponding response.

The information about spectrum is optional. The format is as follows:

`spec` is a 3-D array of dimension `ny`-by-`ny`-by-`Nf`, such that `spec(ky1,ky2,kf)` is the cross spectrum between the noise at output `ky1` and the noise at output `ky2`, at frequency `Freqs(kf)`. When `ky1 = ky2` the (power) spectrum of the noise at output `ky1` is thus obtained. For a single-output model, `spec` can be given as a vector.

`speccov` is a 3-D array of dimension `ny`-by-`ny`-by-`Nf`, such that `speccov(ky1,ky1,kf)` is the variance of the corresponding power spectrum. Normally, no information is included about the covariance of the nondiagonal spectrum elements.

If only `SpectrumData` is to be packaged in the `idfrd` object, set `Response = []`.

Creating idfrd from a Given Model

`idfrd` can also be computed from a given model `mod` (defined as any `idmodel` object).

If the frequencies `Freqs` are not specified, a default choice is made based on the dynamics of the model `mod`.

If `mod` has `InputDelay` different from zero, these are appended as phase lags, and `h` will then have an `InputDelay` of `0`.

The estimated covariances are computed using the Gauss approximation formula from the uncertainty information in `mod`. For models with complicated parameter dependencies, numerical differentiation is applied. The step sizes for the numerical derivatives are determined by `nuderst`.

Frequency responses for submodels can be obtained by the standard subreferencing, `h = idfrd(m(2,3))`. See `idmodel`. In particular, `h = idfrf(m('measured'))` gives an `h` that just contains the `ResponseData (G)` and no spectra. Also `h = idfrd(m('noise'))` gives an `h` that just contains `SpectrumData`.

The `idfrd` models can be graphed with `bode`, `ffplot`, and `nyquist`, which all accept mixtures of `idmodel` and `idfrd` models as arguments. Note that `spa`, `spafdr`, and `etfe` return their estimation results as `idfrd` objects.

idfrd Properties

- `ResponseData`: 3-D array of the complex-valued frequency response as described above. For SISO systems use `Response(1,1,:)` to obtain a vector of the response data.
- `Frequency`: Column vector containing the frequencies at which the responses are defined.
- `CovarianceData`: 5-D array of the covariance matrices of the response data as described abfove.
- `SpectrumData`: 3-D array containing power spectra and cross spectra of the output disturbances (noise) of the system.

- `NoiseCovariance`: 3-D array containing the variances of the power spectra, as explained above.
- `Units`: Unit of the frequency vector. Can assume the values `'rad/s'` and `'Hz'`.
- `Ts`: Scalar denoting the sampling interval of the model whose frequency response is stored. `'Ts' = 0` means a continuous-time model.
- `Name`: An optional name for the object.
- `InputName`: String or cell array containing the names of the input channels. It has as many entries as there are input channels.
- `OutputName`: Correspondingly for the output channels.
- `InputUnit`: Units in which the input channels are measured. It has the same format as `'InputName'`.
- `OutputUnit`: Correspondingly for the output channels.
- `InputDelay`: Row vector of length equal to the number of input channels. Contains the delays from the input channels. These should thus be appended as phase lags when the response is calculated. This is done automatically by `freqresp`, `bode`, `ffplot`, and `nyquist`. Note that if the `idfrd` is calculated from an `idmodel`, possible input delays in that model are converted to phase lags, and the `InputDelay` of the `idfrd` model is set to zero.
- `Notes`: An arbitrary field to store extra information and notes about the object.
- `UserData`: An arbitrary field for any possible use.
- `EstimationInfo`: Structure that contains information about the estimation process that is behind the frequency data. It contains the following fields (see also the reference page for `EstimationInfo`).
  - `Status`: Gives the status of the model, for example, `'Not estimated'`.
  - `Method`: The identification routine that created the model.

- `WindowSize`: If the model was estimated by `spa`, `spafdr`, or `etfe`, the size of window (input argument `M`, the resolution parameter) that was used. This is scalar or a vector.
- `DataName`: Name of the data set from which the model was estimated.
- `DataLength`: Length of this data set.

Note that you can set or retrieve all properties either with the `set` and `get` commands or by subscripts. Autofill applies to all properties and values, and these are case insensitive:

```
h.ts = 0
loglog(h.fre,squeeze(h.spe(2,2,:)))
```

For a complete list of property values, use `get(m)`. To see possible value assignments, use `set(m)`. See also `idprops idfrd`.

Subreferencing

The different channels of the `idfrd` are retrieved by subreferencing.

```
h(outputs,inputs)
```

`h(2,3)` thus contains the response data from input channel 3 to output channel 2, and, if applicable, the output spectrum data for output channel 2. The channels can also be referred to by their names, as in `h('power',{'voltage', 'speed'})`.

```
h('m')
```

contains the information for measured inputs only, that is, just `ResponseData`, while

```
h('n')
```

(`'n'` for `'noise'`) just contains `SpectrumData`.

Horizontal Concatenation

Adding input channels,

```
h = [h1,h2,...,hN]
```

creates an `idfrd` model `h`, with `ResponseData` containing all the input channels in `h1,...,hN`. The output channels of `hk` must be the same, as well as the frequency vectors. `SpectrumData` is ignored.

Vertical Concatenation

Adding output channels,

```
h = [h1;h2;... ;hN]
```

creates an `idfrd` model `h` with `ResponseData` containing all the output channels in `h1, h2,...,hN`. The input channels of `hk` must all be the same, as well as the frequency vectors. `SpectrumData` is also appended for the new outputs. The cross spectrum between output channels is then set to zero.

Converting to iddata

You can convert an `idfrd` object to a frequency-domain `iddata` object by

```
Data = iddata(Idfrdmodel)
```

See `iddata`.

Examples

Compare the results from spectral analysis and an ARMAX model.

```
m = armax(z,[2 2 2 1]);
g = spa(z)
g = spafdr(z,[],{0,10})
bode(g,m)
```

Compute separate `idfrd` models, one containing the frequency function and the other the noise spectrum.

```
g = idfrd(m('m'))
phi = idfrd(m('n'))
```

See Also

bode

etfe

ffplot

freqresp

nyquist

spa

spafdr

Purpose

Class for storing linear ODE models

Syntax

```
m = idgrey(MfileName,ParameterVector,CDmfile)
m = idgrey(MfileName,ParameterVector,CDmfile,FileArgument,Ts,...
'Property1',Value1,...,'PropertyN',ValueN)
```

Description

The function `idgrey` is used to create arbitrarily parameterized state-space models as `idgrey` objects.

`MfileName` is the name of an M-file that defines how the state-space matrices depend on the parameters to be estimated. The format of this M-file is given by

```
[A,B,C,D,K,X0] = mymfile(pars,Tsm,Auxarg)
```

and is further discussed below.

`ParameterVector` is a column vector of the nominal/initial parameters. Its length must be equal to the number of free parameters in the model (that is, the argument `pars` in the example below).

The argument `CDmfile` describes how the user-written M-file handles continuous and discrete-time models. It takes the following values:

- `CDmfile = 'cd'`: The M-file returns the continuous-time state-space matrices when called with the argument `Tsm = 0`. When called with a value `Tsm > 0`, the M-file returns the discrete-time state-space matrices, obtained by sampling the continuous-time system with sampling interval `Tsm`. The M-file must consequently in this case include the sampling procedure.
- `CDmfile = 'c'`. The M-file always returns the continuous-time state-space matrices, no matter the value of `Tsm`. In this case the toolbox's estimation routines will provide the sampling when you are fitting the model to discrete-time data.
- `CDmfile = 'd'`. The M-file always returns discrete-time state-space matrices that may or may not depend on `Tsm`.

The argument `FileArgument` corresponds to the auxiliary argument `Auxarg` in the user-written M-file. It can be used to handle several variants of the model structure, without your having to edit the M-file. If it is not used, enter `FileArgument = []`. (Default.)

`Ts` denotes the sampling interval of the model. Its default value is `Ts = 0`, that is, a continuous-time model.

The `idgrey` object is a child of `idmodel`. Therefore any `idmodel` properties can be set as property name/property value pairs in the `idgrey` command. They can also be set by the command `set`, or by subassignment, as in

```
m.InputName = {'speed','voltage'}
m.FileArgument = 0.23
```

There are also two properties, `DisturbanceModel` and `InitialState`, that can be used to affect the parameterizations of *K* and *X0*, thus overriding the outputs from the M-file.

idgrey Properties

- `MfileName`: Name of the user-written M-file.
- `CDmfile`: How this file handles continuous and discrete-time models depending on its second argument, `T`.
  - `CDmfile = 'cd'` means that the M-file returns the continuous-time state-space model matrices when the argument `T = 0`, and the discrete-time model, obtained by sampling with sampling interval `T`, when `T > 0`.
  - `CDmfile = 'c'` means that the M-file always returns continuous-time model matrices, no matter the value of T.
  - `CDmfile = 'd'` means that the M-file always returns discrete-time model matrices that may or may not depend on the value of `T`.
- `FileArgument`: Possible extra input arguments to the user-written M-file.
- `DisturbanceModel`: Affects the parameterization of the K matrix. It can assume the following values:

- `'Model'`: This is the default. It means that the K matrix obtained from the user-written M-file is used.
- `'Estimate'`: The K matrix is treated as unknown and all its elements are estimated as free parameters.
- `'Fixed'`: The K matrix is fixed to a given value.
- `'None'`: The K matrix is fixed to zero, thus producing an output-error model.

Note that in the three last cases the output `K` from the user-written M-file is ignored. The estimated/fixed value is stored internally and does not change when the model is sampled, resampled, or converted to continuous time. Note also that this estimated value is tailored only to the sampling interval of the data.

- `InitialState`: Affects the parameterization of the X0 vector. It can assume the following values:
  - `'Model'`: This is the default. It means that the X0 vector is obtained from the user-written M-file.
  - `'Estimate'`: The X0 matrix is treated as unknown and all its elements are estimated as free parameters.
  - `'Fixed'`: The X0 vector is fixed to a given value.
  - `'Backcast'`: The X0 vector is estimated using a backcast operation analogous to the `idss` case.
  - `'Auto'`: Makes a data-dependent choice among `'Estimate'`, `'Backcast'`, and `'Model'`.
- `A`, `B`, `C`, `D`, `K`, and `X0`: The state-space matrices. For `idgrey` models, only `'K'` and `'X0'` can be set; the others can only be retrieved. The set `'K'` and `'X0'` are relevant only when `DisturbanceModel/InitialState` are `Estimate` or `Fixed`.
- `dA`, `dB`, `dC`, `dD`, `dK`, and `dX0`: The estimated standard deviations of the state-space matrices. These cannot be set, only retrieved.

In addition, any idgrey object also has all the properties of idmodel. See Algorithm Properties and the reference page for idmodel.

Note that you can set or retrieve all properties using either the set and get commands or subscripts. Autofill applies to all properties and values, and they are case insensitive.

```
m.fi = 10;
set(m,'search','gn')
p = roots(m.a)
```

For a complete list of property values, use get(m). To see possible value assignments, use set(m). See also idprops and idgrey.

M-File Details

The model structure corresponds to the general linear state-space structure

$$\tilde{x}(t) = A(\theta)x(t) + B(\theta)u(t) + K(\theta)e(t)$$

$$x(0) = x_0(\theta)$$

$$y(t) = C(\theta)x(t) + D(\theta)u(t) + e(t)$$

Here $\tilde{x}(t)$ is the time derivative $\dot{x}(t)$ for a continuous-time model and $x(t+Ts)$ for a discrete-time model.

The matrices in this time-discrete model can be parameterized in an arbitrary way by the vector θ. Write the format for the M-file as follows:

```
[A,B,C,D,K,x0] = mymfile(pars,T,Auxarg)
```

Here the vector pars contains the parameters θ, and the output arguments A, B, C, D, K, and x0 are the matrices in the model description that correspond to this value of the parameters and this value of the sampling interval T.

T is the sampling interval, and Auxarg is any variable of auxiliary quantities with which you want to work. (In that way you can change certain constants and other aspects in the model structure without

having to edit the M-file.) Note that the two arguments `T` and `Auxarg` must be included in the function head of the M-file, even if they are not used within the M-file.

A comment about `CDmfile`: If a continuous-time model is sought, it is easiest to let the M-file deliver just the continuous-time model, that is, have `CDmfile = 'c'` and rely upon the toolbox's routines for the proper sampling. Similarly, if the underlying parameterization is indeed discrete time, it is natural to deliver the discrete-time model matrices and let `CDmfile = 'd'`. If the underlying parameterization is continuous, but you prefer for some reason to do your own sampling inside the M-file in accordance with the value of `T`, then let your M-file deliver the continuous-time model when called with `T = 0`, that is, the alternative `CMmfile = 'cd'`. This avoids sampling and then transforming back (using `d2c`) to find the continuous-time model.

idgrey Definition of States

The states of an `idgrey` model are defined explicitly by the user in the M-file or MEX-file (as specified in MfileName property) storing the model structure. The concept of states is useful for functions such as `sim`, `predict`, `compare` and `findstates`.

Examples

Use the M-file `mynoise` given in "Example – Estimating a Discrete-Time Grey-Box Model with Parameterized Disturbance" to obtain a physical parameterization of the Kalman gain.

```
mn = idgrey('mynoise',[0.1,-2,1,3,0.2]','d', 1)
m = pem(z,mn)
```

Purpose

Generate input signals

Syntax

```
u = idinput(N)
u = idinput(N,type,band,levels)
[u,freqs] = idinput(N,'sine',band,levels,sinedata)
```

Description

`idinput` generates input signals of different kinds, which are typically used for identification purposes. `u` is returned as a matrix or column vector.

For further use in the toolbox, we recommend that you create an `iddata` object from `u`, indicating sampling time, input names, periodicity, and so on:

```
u = iddata([],u);
```

`N` determines the number of generated input data. If `N` is a scalar, `u` is a column vector with this number of rows.

`N = [N nu]` gives an input with `nu` input channels each of length `N`.

`N = [P nu M]` gives a periodic input with `nu` channels, each of length `M*P` and periodic with period `P`.

Default is `nu = 1` and `M = 1`.

`type` defines the type of input signal to be generated. This argument takes one of the following values:

- `type = 'rgs'`: Gives a random, Gaussian signal.
- `type = 'rbs'`: Gives a random, binary signal. This is the default.
- `type = 'prbs'`: Gives a pseudorandom, binary signal.
- `type = 'sine'`: Gives a signal that is a sum of sinusoids.

The frequency contents of the signal is determined by the argument `band`. For the choices `type = 'rs'`, `'rbs'`, and `'sine'`, this argument is a row vector with two entries

```
band = [wlow, whigh]
```

that determine the lower and upper bound of the passband. The frequencies `wlow` and `whigh` are expressed in fractions of the Nyquist frequency. A white noise character input is thus obtained for `band = [0 1]`, which is also the default value.

For the choice `type = 'prbs'`,

```
band = [0, B]
```

where `B` is such that the signal is constant over intervals of length `1/B` (the clock period). In this case the default is `band = [0 1]`.

The argument `levels` defines the input level. It is a row vector

```
levels = [minu, maxu]
```

such that the signal `u` will always be between the values `minu` and `maxu` for the choices `type = 'rbs'`, `'prbs'`, and `'sine'`. For `type = 'rgs'`, the signal level is such that `minu` is the mean value of the signal, minus one standard deviation, while `maxu` is the mean value plus one standard deviation. Gaussian white noise with zero mean and variance one is thus obtained for `levels = [-1, 1]`, which is also the default value.

Some PRBS Aspects

If more than one period is demanded (that is, `M > 1`), the length of the data sequence and the period of the PRBS signal are adjusted so that an integer number of maximum length PRBS periods is always obtained. If `M = 1`, the period of the PRBS signal is chosen to that it is longer than `P = N`. In the multiple-input case, the signals are maximally shifted. This means `P/nu` is an upper bound for the model orders that can be estimated with such a signal.

Some Sine Aspects

In the `'sine'` case, the sinusoids are chosen from the frequency grid

```
freq = 2*pi*[1:Grid_Skip:fix(P/2)]/P
```

intersected with `pi*[band(1) band(2)]`. For `Grid_Skip`, see below. For multiple-input signals, the different inputs use different frequencies from this grid. An integer number of full periods is always delivered. The selected frequencies are obtained as the second output argument, `freqs`, where row `ku` of `freqs` contains the frequencies of input number `ku`. The resulting signal is affected by a fifth input argument, `sinedata`

```
sinedata = [No_of_Sinusoids, No_of_Trials, Grid_Skip]
```

meaning that `No_of_Sinusoids` is equally spread over the indicated `band`. `No_of_Trials` (different, random, relative phases) are tried until the lowest amplitude signal is found.

```
Default: sinedata = [10,10,1];
```

`Grid_Skip` can be useful for controlling odd and even frequency multiples, for example, to detect nonlinearities of various kinds.

Algorithm

Very simple algorithms are used. The frequency contents are achieved for `'rgs'` by an eighth-order Butterworth, noncausal filter, using `idfilt`. This is quite reliable. The same filter is used for the `'rbs'` case, before making the signal binary. This means that the frequency contents are not guaranteed to be precise in this case.

For the `'sine'` case, the frequencies are selected to be equally spread over the chosen grid, and each sinusoid is given a random phase. A number of trials are made, and the phases that give the smallest signal amplitude are selected. The amplitude is then scaled so as to satisfy the specifications of `levels`.

References

See Söderström and Stoica (1989), Chapter C5.3. For a general discussion of input signals, see Ljung (1999), Section 13.3.

Examples

Create an input consisting of five sinusoids spread over the whole frequency interval. Compare the spectrum of this signal with that of its square. The frequency splitting (the square having spectral support at other frequencies) reveals the nonlinearity involved:

```
u = idinput([100 1 20],'sine',[],[],[5 10 1]);
u = iddata([],u,1,'per',100);
u2 = u.u.^2;
u2 = iddata([],u2,1,'per',100);
ffplot(etfe(u),'r*',etfe(u2),'+')
```

idmdlsim

Purpose

Simulate `idmodel` objects using Simulink® software

Syntax

```
idmdlsim
```

Description

Typing `idmdlsim` launches the Idmodel Simulink block. By clicking the block you can specify the `idmodel` to simulate, whether to include initial state values, and whether to add noise to the simulation in accordance with the model's own noise description.

See Also

`compare`

`pe`

`predict`

`sim`

`simsd`

Purpose

Superclass for linear models

Description

`idmodel` is an object that you do not deal with directly. It contains all the common properties of the model objects `idarx`, `idgrey`, `idpoly`, `idproc`, and `idss`, which are returned by the different estimation routines.

Basic Use

If you just estimate models from data, the model objects should be transparent. All parametric estimation routines return `idmodel` results.

```
m = arx(Data,[2 2 1])
```

The model `m` contains all relevant information. Just typing `m` will give a brief account of the model. `present(m)` also gives information about the uncertainties of the estimated parameters. `get(m)` gives a complete list of model properties.

Most of the interesting properties can be directly accessed by subreferencing:

```
m.a
m.da
```

See the property list obtained by `get(m)`, as well as the property lists of `idgrey`, `idarx`, `idpoly`, and `idss` in Chapter 2, "Functions – Alphabetical List" for more details on this. See also `idprops`.

The characteristics of the model `m` can be directly examined and displayed by commands like `impulse`, `step`, `bode`, `nyquist`, and `pzmap`. The quality of the model is assessed by commands like `compare` and `resid`. When you have Control System Toolbox™ software installed, you can use `view(m)` to access various display functions.

To extract state-space matrices, transfer function polynomials, etc., use the commands `arxdata`, `polydata`, `tfdata`, `ssdata`, and `zpkdata`.

To compute the frequency response of the model, use the commands `idfrd` and `freqresp`.

Creating and Modifying Model Objects

If you want to define a model to use, for example, for simulating data, you need to use the model creator functions:

- `idarx`, for multivariable ARX models
- `idgrey`, for user-defined gray-box state-space models
- `idpoly`, for single-output polynomial models
- `idproc`, for simple, continuous-time process models
- `idss`, for state-space models

If you want to estimate a state-space model with a specific internal parameterization, you need to create an `idss` model or an `idgrey` model. See the reference pages for these functions.

Dealing with Input and Output Channels

For multivariable models, you construct submodels containing a subset of inputs and outputs by simple subreferencing. The outputs and input channels can be referenced according to

```
m(outputs,inputs)
```

Use a colon (:) to denote all channels and an empty matrix (`[]`) to denote no channels. The channels can be referenced by number or by name. For several names, you must use a cell array, such as

```
m3 = m('position',{'power','speed'})
```

or

```
m3 = m(3,[1 4])
```

Thus `m3` is the model obtained from `m` by looking at the transfer functions from input numbers 1 and 4 (with input names 'power' and 'speed') to output number 3 (with name `position`).

For a single-output model `m`,

```
m4 = m(inputs)
```

selects the corresponding input channels, and for a single-input model,

```
m5 = m(outputs)
```

selects the indicated output channels.

Subreferencing is quite useful, for example, when a plot of just some channels is desired.

Noise Channels

The estimated models have two kinds of input channels: the measured inputs u and the noise inputs e. For a general linear model `m`, we have

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

where `u` is the nu-dimensional vector of measured input channels and `e` is the ny-dimensional vector of noise channels. The covariance matrix of `e` is given by the property `'NoiseVariance'`. Occasionally this matrix Λ is written in factored form,

$$\Lambda = LL^T$$

This means that `e` can be written as

$$e = Lv$$

where v is white noise with identity covariance matrix (independent noise sources with unit variances).

If `m` is a time series (nu = `0`), G is empty and the model is given by

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

For the model `m`, the restriction to the transfer function matrix G is obtained by

```
m1 = m('measured') or just m1 = m('m')
```

Then `e` is set to `0` and `H` is removed.

Analogously,

```
m2 = m('noise') or just m2 = m('n')
```

creates a time-series model `m2` from `m` by ignoring the measured input. That is, `m2` describes the signal He.

For a system with measured inputs, `bode`, `step`, and other transformation and display functions deal with the transfer function matrix G. To obtain or graph the properties of the disturbance model H, it is therefore important to make the transformations `m('n')`. For example,

```
bode(m('n'))
```

plots the additive noise spectra according to the model `m`, while

```
bode(m)
```

just plots the frequency responses of G.

To study the noise contributions in more detail, it is useful to convert the noise channels to measured channels, using the command `noisecnv`.

```
m3 = noisecnv(m)
```

This creates a model `m3` with all input channels, both measured `u` and noise sources `e`, treated as measured signals,. That is, `m3` is a model from `u` and `e` to `y`, describing the transfer functions G and H. The information about the variance of the innovations `e` is lost. For example, studying the step response from the noise channels does not take into consideration how large the noise contributions actually are.

To include that information, `e` should first be normalized, $e = Lv$, so that v becomes white noise with an identity covariance matrix.

```
m4 = noisecnv(m,'Norm')
```

This creates a model `m4` with `u` and v treated as measured signals.

$$y(t) = G(q)u(t) + H(q)Lv(t) = \begin{bmatrix} G & HL \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

For example, the step responses from v to y will now reflect the typical size of the disturbance influence because of the scaling by L. In both cases, the previous noise sources that have become regular inputs will automatically get input names that are related to the corresponding output. The unnormalized noise sources `e` have names like `'e@ynam1'` (noise `e` at output channel `ynam1`), while the normalized sources `v` are called `'v@ynam1'`.

Retrieving Transfer Functions

The functions that retrieve transfer function properties, `ssdata`, `tfdata`, and `zpkdata`, will thus work as follows for a model () with measured inputs. (`fcn` is `ssdata`, `tfdata`, or `zpkdata`.)

`fcn(m)` returns the properties of G (ny outputs and nu inputs).

`fcn(m('n'))` returns the properties of the transfer function H (ny outputs and ny inputs).

`fcn(noisec nv(m,'Norm'))` returns the properties of the transfer function [G HL} (ny outputs and ny+nu inputs). Analogously,

```
m1 = m('n'). fcn(noisecnv(m1,'Norm'))
```

returns the properties of the transfer function HL (ny outputs and ny inputs).

If `m` is a time-series model, `fcn(m)` returns the properties of H, while

```
fcn(noisecnv(m,'Norm'))
```

returns the properties of HL.

Note that the estimated covariance matrix `NoiseVariance` itself is uncertain. This means that the uncertainty information about *H* is different from that of *HL*.

idmodel Properties

In the list below, `ny` is the number of output channels, and `nu` is the number of input channels:

- `Name`: An optional name for the data set. An arbitrary string.
- `OutputName`, `InputName`: Cell arrays of length `ny`-by-1 and `nu`-by-1 containing the names of the output and input channels. For estimated models, these are inherited from the data. If not specified, they are given default names `{'y1','y2',...}` and `{'u1','u2',...}`.
- `OutputUnit`, `InputUnit`: Cell arrays of length `ny`-by-1 and `nu`-by-1 containing the units of the output and input channels. Inherited from data for estimated models.
- `TimeUnit`: Unit for the sampling interval.
- `Ts`: Sampling interval. A nonnegative scalar. `Ts = 0` denotes a continuous-time model. Note that changing just `Ts` will not recompute the model parameters. Use `c2d` and `d2c` for recomputing the model to other sampling intervals.
- `ParameterVector`: Vector of adjustable parameters in the model structure. Initial/nominal values or estimated values, depending on the status of the model. A column vector.
- `PName`: The names of the parameters. A cell array of the length of the parameter vector. If not specified, it will contain empty strings. See also `setpname`.
- `CovarianceMatrix`: Estimated covariance matrix of the parameter vector. For a nonestimated model this is the empty matrix. For state-space models in the `'Free'` parameterization the covariance matrix is also the empty matrix, since the individual matrix elements are not identifiable then. Instead, in this case, the covariance information is hidden (in the hidden property `'Utility'`) and retrieved by the relevant functions when necessary. Setting

`CovarianceMatrix` to `'None'` inhibits calculation of covariance and uncertainty information. This can save substantial time for certain models.

- `NoiseVariance`: Covariance matrix of the noise source `e`. An `ny`-by-`ny` matrix.
- `InputDelay`: Vector of size `nu`-by-1, containing the input delay from each input channel. For a continuous-time model (`Ts = 0`) the delay is measured in `TimeUnit`, while for discrete-time models (`Ts > 0`) the delay is measured as the number of samples. Note the difference between `InputDelay` and `nk` (which is a property of `idarx`, `idss`, and `idpoly`). `'Nk'` is a model structure property that tells the model structure to include such an input delay. In that case, the corresponding state-space matrices and polynomials will explicitly contain `Nk` input delays. The property `InputDelay`, on the other hand, is an indication that in addition to the model as defined, the inputs should be shifted by the given amount. `InputDelay` is used by `sim` and the estimation routines to shift the input data. When computing frequency responses, the `InputDelay` is also respected. Note that `InputDelay` can be both positive and negative.
- `Algorithm`: See the reference page for `Algorithm Properties`.
- `EstimationInfo`: See the reference page for `EstimationInfo`.
- `Notes`: An arbitrary field to store extra information and notes about the object.
- `UserData`: An arbitrary field for any possible use.

Note All properties can be set or retrieved either by these commands or by subscripts. Autofill applies to all properties and values, and is case insensitive.

For a complete list of property values, use `get(m)`. To see possible value assignments, use `set(m)`.

Subreferencing

The outputs and input channels can be referenced according to

```
m(outputs,inputs)
```

Use a colon (:) to denote all channels and an empty matrix ([ ]) to denote no channels. The channels can be referenced by number or by name. For several names, you must use a cell array.

```
m2 = m('y3',{'u1','u4'})
m3 = m(3,[1 4])
```

For a single output model `m`,

```
m4 = m(inputs)
```

selects the corresponding input channels, and for a single input model,

```
m5 = m(outputs)
```

selects the indicated output channels.

The string `'measured'` (or any abbreviation like `'m'`) means the measured input channels.

```
m4 = m(3,'m')
m('m') is the same as m(:,'m')
```

Similarly, the string `'noise'` (or any abbreviation) refers to the noise input channels. See “Noise Channels” on page 2-165 for more details.

Horizontal Concatenation

Adding input channels,

```
m = [m1,m2,...,mN]
```

creates an `idmodel` object `m`, consisting of all the input channels in `m1,... mN`. The output channels of `mk` must be the same.

Vertical Concatenation

Adding output channels,

```
m = [m1;m2;... ;mN]
```

creates an `idmodel` object `m` consisting of all the output channels in `m1`, `m2`, `..mN`. The input channels of `mk` must all be the same.

Online Help Functions

Type `idhelp idmodel`, `idprops idmodel`, `Methods(idmodel)`, `idprops idmodel algorithm`.

See Also

`Algorithm Properties`

`EstimationInfo`

`compare`

`idarx`

`idgrey`

`idpoly`

`idproc`

`idss`

`noisecnv`

idnlarx

Purpose

Class representing nonlinear ARX models

Syntax

```
m=idnlarx([na nb nk])
m=idnlarx([na nb nk],Nonlinearity)
m=idnlarx([na nb nk],Nonlinearity,P1,V1,...,PN,VN)
```

Description

`idnlarx` is an object that stores nonlinear ARX model properties, including model parameters.

Typically, you use the `nlarx` command to both specify the nonlinear ARX model properties and estimate the model. You can specify the model properties directly in the `nlarx` syntax.

For information about the nonlinear ARX model structure, see "Definition of the Nonlinear ARX Model".

The information in these reference pages summarizes the `idnlarx` model constructor and properties. It discusses the following topics:

- "idnlarx Constructor" on page 2-172
- "idnlarx Properties" on page 2-174
- "idnlarx Algorithm Properties" on page 2-177
- "idnlarx Advanced Algorithm Properties" on page 2-181
- "idnlarx EstimationInfo Properties" on page 2-183

idnlarx Constructor

Typically, you use the `nlarx` estimator command to specify the model properties and estimate the nonlinear ARX model. However, you can also use the `idnlarx` constructor to create the nonlinear ARX model object in advance, and then estimate the parameters of this object using `pem`.

`m=idnlarx([na nb nk])` creates an `idnlarx` object with the specified number of output terms `na`, input terms `nb`, and input delays `nk`. It uses `sigmoidnet` as the nonlinearity estimator.

`m=idnlarx([na nb nk],Nonlinearity)` creates an `idnlarx` object with the specified nonlinearity estimator.

`m=idnlarx([na nb nk],Nonlinearity,P1,V1,...,PN,VN)` creates an `idnlarx` object and specifies `idnlarx` property-value pairs. For more information about `idnlarx` properties, see "idnlarx Properties" on page 2-174.

The constructor arguments have the following specifications:

`[na nb nk]`

`na` is the number of output terms, `nb` is the number of input terms, and `nk` is the input delays from each input to output.

For `ny` outputs and `nu` inputs, `[na nb nk]` has as many rows as there are outputs. In this case, `na` is an `ny`-by-`ny` matrix whose *i*-*j*th entry gives the number of delayed *j*th outputs used to compute the *i*th output. `nb` and `nk` are `ny`-by-`nu` matrices.

These orders specify the regressors and the predicted output is a nonlinear function of these regressors:

$$F\left(y(t-1),\ldots,y(t-na),u(t-nk),\ldots,u(t-nk-nb+1)\right)$$

`Nonlinearity`

Specifies the nonlinearity estimator object (formula for F above) as one of the following: `sigmoidnet` (default), `wavenet`, `treepartition`, `customnet`, `neuralnet`, and `linear`. The nonlinearity estimator objects have properties that you can set in the constructor, as follows:

```
m=idnlarx([2 3 1],sigmoidnet('Num',15))
```

For `ny` outputs, `Nonlinearity` is an `ny`-by-1 array, such as `[sigmoidnet;wavenet]`. However, if you specify a scalar object, this nonlinearity object applies to all outputs.

To use default nonlinearity properties, you may also specify the nonlinearity object name as a string. For example:

```
m=idnlarx([3 2 1],'sigmoidnet')
```

```
% Abbreviated.
m=idnlarx([3 2 1],'sig')

% The estimator 'linear' or [] denotes the
% absence of nonlinearity
m=idnlarx([2 2 1], []) % m contains no nonlinearity
```

For more information about nonlinearity properties, see the corresponding reference pages.

idnlarx Properties

You can include property-value pairs in the model estimator or constructor to specify the model structure and estimation algorithm properties.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% Get the model time unit
get(m,'TimeUnit')
% Get value of Nonlinearity property
m.Nonlinearity
```

The following table summarizes `idnlarx` model properties. The general `idnlmodel` properties also apply to this nonlinear model object (see the corresponding reference pages).

Property Name	Description
`Algorithm`	A structure that specifies the estimation algorithm options, as described in “idnlarx Algorithm Properties” on page 2-177.

Property Name	Description
`CustomRegressors`	Custom expression in terms of standard regressors. Assignable values: • Cell array of strings. For example: `{'y1(t-3)^3','y2(t-1)*u1(t-3)','sin(u3(t-2))'}`. • Object array of `customreg` objects. These objects can be created using commands such as `customreg` and `polyreg`. For more information about this object, see the corresponding reference pages.
`EstimationInfo`	A read-only structure that stores estimation settings and results, as described in “idnlarx EstimationInfo Properties” on page 2-183.
`Focus`	Specifies `'Prediction'` or `'Simulation'`. Assignable values: • `'Prediction'` — The estimation algorithm minimizes $\|y-\hat{y}\|$, where $\hat{y}$ is the 1–step ahead predicted output. • `'Simulation'` — The estimation algorithm minimizes the output error fit. That is, when computing $\hat{y}$, y in the regressors in F are replaced by values simulated from the input only. **Note** You cannot use `'Simulation'` when the model contains custom regressors that use output variables, or if the model contains `treepartition` or `neuralnet` as a nonlinearity estimator.

Property Name	Description
`NonlinearRegressors`	Specifies which standard or custom regressors enter the nonlinear block. For multiple-output models, use cell array of n_y elements (n_y = number of model outputs). For each output, assignable values are: • `'all'` — All regressors enter the nonlinear block. • `'input'` — Input regressors only. • `'output'` — Output regressors only. • `'standard'` — Standard regressors only. • `'custom'` — Custom regressors only. • `'search'` — Specifies that the estimation algorithm perform an exhaustive search of the best regressor combination. • `'[]'` — No regressors enter the nonlinear block. • A vector of indices: Specifies the indices of the regressors that should be used in the nonlinear estimator. To determine the order of regressors, use the `getreg` function.

Property Name	Description
`Nonlinearity`	Nonlinearity estimator object. Assignable values include `sigmoidnet` (default), `wavenet`, `treepartition`, `customnet`, `neuralnet`, and `linear`. If the model contains only one regressor, you can also use `saturation`, `deadzone`, `pwlinear` or `poly1d` as a nonlinearity estimator. For `ny` outputs, `Nonlinearity` is an `ny`-by-1 array, such as `[sigmoidnet;wavenet]` for a two-output model. However, if you specify a scalar object, this nonlinearity object applies to all outputs.

`na`
`nb`
`nk`
	Model orders and input delays, where `na` is the number of output terms, `nb` is the number of input terms, and `nk` is the delay from input to output in terms of the number of samples.
	
	For `ny` outputs and `nu` inputs, `na` is an `ny`-by-`ny` matrix whose *i*-*j*th entry gives the number of delayed *j*th outputs used to compute the *i*th output. `nb` and `nk` are `ny`-by-`nu` matrices.

idnlarx Algorithm Properties

The following table summarizes the fields of the `Algorithm` `idnlarx` model properties. `Algorithm` is a structure that specifies the estimation-algorithm options.

Property Name	Description
`Advanced`	A structure that specifies additional estimation algorithm options, as described in “idnlarx Advanced Algorithm Properties” on page 2-181.

Property Name	Description
`Criterion`	Specifies criterion used during minimization. `Criterion` can have the following values: • `'Det'`: Minimize `det(E'*E)`, where `E` represents the prediction error. This is the optimal choice in a statistical sense and leads to the maximum likelihood estimates in case nothing is known about the variance of the noise. It uses the inverse of the estimated noise variance as the weighting function. This is the default criterion used for all models, except `idnlgrey` which uses `'Trace'` by default. • `'Trace'`: Minimize the trace of the weighted prediction error matrix `trace(E'*E*W)`, where `E` is the matrix of prediction errors, with one column for each output, and `W` is a positive semi-definite symmetric matrix of size equal to the number of outputs. By default, `W` is an identity matrix of size equal to the number of model outputs (so the minimization criterion becomes `trace(E'*E)`, or the traditional least-squares criterion). You can specify the relative weighting of prediction errors for each output using the `Weighting` field of the `Algorithm` property. If the model contains `neuralnet` or `treepartition` as one of its nonlinearity estimators, weighting is not applied since the estimations are performed independently for each output.
`IterWavenet`	(For `wavenet` nonlinear estimator only) Toggles performing iterative or noniterative estimation. Default: `'auto'`. Assignable values: • `'auto'` — First estimation is noniterative and subsequent estimation are iterative. • `'On'` — Perform iterative estimation only. • `'Off'` — Perform noniterative estimation only.

Property Name	Description
`LimitError`	Robustification criterion that limits the influence of large residuals, specified as a positive real value. Residual values that are larger than `'LimitError'` times the estimated residual standard deviation have a linear cost instead of the usual quadratic cost. Default: `0` (no robustification).
`MaxIter`	Maximum number of iterations for the estimation algorithm, specified as a positive integer. Default: `20`.
`MaxSize`	The number of elements (size) of the largest matrix to be formed by the algorithm. Computational loops are used for larger matrices. Use this value for memory/speed trade-off. `MaxSize` can be any positive integer. Default: `250000`. **Note** The original data matrix of u and y must be smaller than `MaxSize`.
`SearchMethod`	Method used by the iterative search algorithm. Assignable values: • `'Auto'` — Automatically chooses from the following methods. • `'gn'` — Subspace Gauss-Newton method. • `'gna'` — Adaptive Gauss-Newton method. • `'grad'` — A gradient method. • `'lm'` — Levenberg-Marquardt method. • `'lsqnonlin'` — Nonlinear least-squares method (requires theOptimization Toolbox™ product). This method only handles the `'Trace'` criterion.

Property Name	Description
`Tolerance`	Specifies to terminate the iterative search when the expected improvement of the parameter values is less than `Tolerance`, specified as a positive real value in %. Default: `0.01`.
`Trace`	Toggles displaying or hiding estimation progress information in the MATLAB® Command Window. Default: `'Off'`. Assignable values: • `'Off'` — Hide estimation information. • `'On'` — Display estimation information.
`Weighting`	Positive semi-definite matrix `W` used for weighted trace minimization. When `Criterion = 'Trace'`, `trace(E'*E*W)` is minimized. `Weighting` can be used to specify relative importance of outputs in multiple-input multiple-output models (or reliability of corresponding data) when `W` is a diagonal matrix of nonnegative values. `Weighting` is not useful in single-output models. By default, `Weighting` is an identity matrix of size equal to the number of outputs.

Note The `Criterion` property setting is meaningful in multiple-output cases only. In single-output models, the two criteria are equivalent. Both the `Det` and `Trace` criteria are derived from a general requirement of minimizing a weighted sum of least squares of prediction errors. The `Det` criterion can be interpreted as estimating the covariance matrix of the noise source and using the inverse of that matrix as the weighting. You should specify the weighting when using the `Trace` criterion.

If you want to achieve better accuracy for a particular channel in multiple-input multiple-output models, you should use `Trace` with weighting that favors that channel. Otherwise it is natural to use `Det`. When using `Det` you can check `cond(model.NoiseVariance)` after estimation. If the matrix is ill-conditioned, it may be more robust to use the `Trace` criterion. You can also use `compare` on validation data to check whether the relative error for different channels corresponds you your needs or expectations. Use the `Trace` criterion if you need to modify the relative errors, and check `model.NoiseVariance` to determine what weighting modifications to specify.

The search method of `lsqnonlin` supports the `Trace` criterion only.

idnlarx Advanced Algorithm Properties

The following table summarizes the fields of the `Algorithm.Advanced` model properties. The fields in the `Algorithm.Advanced` structure specify additional estimation-algorithm options.

Property Name	Description
`GnPinvConst`	When the search direction is computed, the algorithm discards the singular values of the Jacobian that are smaller than `GnPinvConst*max(size(J))*norm(J)*eps`. Singular values that are closer to 0 are included when `GnPinvConst` is decreased. Default: `1e4`. Assign a positive, real value.

Property Name	Description
`LMStartValue`	(For Levenberg-Marquardt search algorithm) The starting level of *regularization* when using the Levenberg-Marquardt search method (`Algorithm.SearchMethod='lm'`). Default: `0.001`. Assign a positive real value.
`LMStep`	(For Levenberg-Marquardt search algorithm) Try this next level of *regularization* to get a lower value of the criterion function. The level of regularization is `LMStep` times the previous level. At the start of a new iteration, the level of regularization is computed as`1/LMStep` times the value from the previous iteration. Default: `10`. Assign a real value `>1`.
`MaxBisections`	Maximum number of bisections performed by the line search algorithm along the search direction (number of rotations of search vector for `'lm'`). Used by `'gn'`, `'lm'`, `'gna'` and `'grad'` search methods (`Algorithm.SearchMethod` property) Default: `10`. Assign a positive integer value.
`MaxFunEvals`	The iterations are stopped if the number of calls to the model file exceeds this value. Default: `Inf`. Assign a positive integer value.
`MinParChange`	The smallest parameter update allowed per iteration. Default: `1e-16`. Assign a positive, real value.

Property Name	Description
`RelImprovement`	The iterations are stopped if the relative improvement of the criterion function is less than `RelImprovement`. Default: `0`. Assign a positive real value. **Note** Does not apply to `Algorithm.SearchMethod='lsqnonlin'`
`StepReduction`	(For line search algorithm) The suggested parameter update is reduced by the factor `'StepReduction'` after each try until either `'MaxBisections'` tries are completed or a lower value of the criterion function is obtained. Default: `2`. Assign a positive, real value `>1`. **Note** Does not apply to `Algorithm.SearchMethod='lsqnonlin'`

idnlarx EstimationInfo Properties

The following table summarizes the fields of the `EstimationInfo` model properties. The read-only fields of the `EstimationInfo` structure store estimation settings and results.

Property Name	Description
`Status`	Shows whether the model parameters were estimated.
`Method`	Shows the estimation method.
`LossFcn`	Value of the loss function, equal to `det(E'*E/N)`, where `E` is the residual error matrix (one column for each output) and `N` is the total number of samples.

Property Name	Description
FPE	Value of Akaike's Final Prediction Error (see fpe).
DataName	Name of the data from which the model is estimated.
DataLength	Length of the estimation data.
DataTs	Sampling interval of the estimation data.
DataDomain	'Time' means time domain data. 'Frequency' is not supported.
DataInterSample	Intersample behavior of the input estimation data used for interpolation: • 'zoh' means zero-order-hold, or piecewise constant. • 'foh' means first-order-hold, or piecewise linear.
WhyStop	Reason for terminating parameter estimation iterations.
UpdateNorm	Norm of the Gauss-Newton in the last iteration. Empty when 'lsqnonlin' is the search method.
LastImprovement	Criterion improvement in the last iteration, shown in %. Empty when 'lsqnonlin' is the search method.
Iterations	Number of iterations performed by the estimation algorithm.
Warning	Any warnings encountered during parameter estimation.
InitRandState	The value of randn('state') at the last randomization of the initial parameter vector.
EstimationTime	Duration of the estimation.

Definition of idnlarx States

The states of an idnlarx object are defined by the set of delayed input and output variables that define the structure of the model. The concept of states is useful for functions such as sim, predict, compare, findstates, data2state, findop, and linearize. A nonlinear ARX model consists of two essential elements — regressors and nonlinearities. The regressors represent the dynamic element of the model while the nonlinearities are static. The regressors include

delayed samples of input and output variables (standard regressors) and user-defined transformations of delayed input and output variables (custom regressors). To determine the states of an `idnlarx` model, you must determine the maximum delay in each input and output variable used by the regressors. If a variable p has a maximum delay of D samples, then it contributes D elements to the state vector: $p(t\text{-}1)$, $p(t\text{-}2)$, ..., $p(t\text{-}D)$.

For example, if you have a single-input, single-output `idnlarx` model defined as:

```
m = idnlarx([2 3 0], 'wavenet', ...
                     'CustomRegressors',
                     {'y1(t-10)*u1(t-1)'});
```

You can use the function to show that `m` has the following 6 regressors:

```
>> getreg(m)
Regressors:
    y1(t-1)
    y1(t-2)
    u1(t)
    u1(t-1)
    u1(t-2)
    y1(t-10)*u1(t-1)
```

From the list of regressors, it can be seen that the maximum delay in output variable `y1` is 10 samples, while the maximum delay in input `u1` is 2 samples. Therefore there are 12 states in this model composed of 10 delayed sampled of output `y1` and 2 delayed samples of input `u1`:

X(t) = [y1(t-1), y2(t-2), …, y1(t-10), u1(t-1), u1(t-2)]

Note The states vector contains the output variables first, followed by the input variables.

As another example, consider the 2-output, 3-input model:

```
m = idnlarx([2 0 2 2 1 1 0 0; 1 0 1 5 0 1 1 0], ...
             [wavenet; linear])
```

You can use to find that the model has following regressors:

```
>> getreg(m)
Regressors:
  For output 1:
    y1(t-1)
    y1(t-2)
    u1(t-1)
    u1(t-2)
    u2(t)
    u2(t-1)
    u3(t)
  For output 2:
    y1(t-1)
    u1(t-1)
    u2(t-1)
    u2(t-2)
    u2(t-3)
    u2(t-4)
    u2(t-5)
```

The maximum delay in output variable `y1` is 2 samples, which occurs in regressor set for output 1. Variable `y2` does not appear in the regressor set since the maximum delay in `y2` is zero. The maximum delays in the three input variables are 2, 5 and 0 respectively. Summarizing this information as before, the state vector is:

X(t) = [y1(t-1), y1(t-2), u1(t-1), u1(t-2), u2(t-1), u2(t-2), u2(t-3), u2(t-4), u2(t-5)]

You can see that the variables `y2` and `u3` do not contribute to the state vector, because the maximum delay in these variables is zero. The

above analysis of determining states from an inspection of regressors can be automated using the `getDelayInfo(idnlarx)` command. This function returns the maximum delays in all I/O variables across all model outputs. For the multiple-input multiple-output model `m`, `getDelayInfo` returns:

```
maxDel = getDelayInfo(m)
maxDel =
     2     0     2     5     0
```

`maxDel` contains the maximum delays for all the input and output variables in the order (`y1`, `y2`, `u1`, `u2`, `u3`). The total number of model states is `sum(maxDel) = 9`.

Note The set of states for an `idnlarx` model is not minimal.

See Also

`getreg`

`nlarx`

`pem`

idnlgrey

Purpose

Class representing nonlinear ODE models

Syntax

```
m = idnlgrey('filename',Order,Parameters)
m = idnlgrey('filename',Order,Parameters,InitialStates)
m = idnlgrey('filename',Order,Parameters,InitialStates,Ts)
m =
idnlgrey('filename',Order,Parameters,InitialStates,Ts,P1,
   V1,...,PN,VN)
```

Description

`idnlgrey` is an object that represents the nonlinear grey-box model.

For information about the nonlinear grey-box models, see "Estimating Nonlinear Grey-Box Models".

The information in these reference pages summarizes the `idnlgrey` model constructor and properties. It discusses the following topics:

- "idnlgrey Constructor" on page 2-188
- "idnlgrey Properties" on page 2-189
- "idnlgrey Advanced Algorithm Properties" on page 2-196
- "idnlgrey Simulation Options" on page 2-198
- "idnlgrey Gradient Options" on page 2-202
- "idnlgrey EstimationInfo Properties" on page 2-203

idnlgrey Constructor

Use the following syntax to define the `idnlgrey` model object:

```
m = idnlgrey('filename',Order,Parameters)
```

```
m = idnlgrey('filename',Order,Parameters,InitialStates)
```

```
m = idnlgrey('filename',Order,Parameters,InitialStates,Ts)
```

```
m =
idnlgrey('filename',Order,Parameters,InitialStates,Ts,P1,V1,...,PN,VN)
```

The `idnlgrey` arguments are defined as follows:

- `'filename'` — Name of the M-file or MEX-file storing the model structure (ODE file).
- `Order` — Vector with three entries `[Ny Nu Nx]`, specifying the number of model outputs `Ny`, the number of inputs `Nu`, and the number of states `Nx`.
- `Parameters` — Parameters, specified as `struct` arrays, cell arrays, or double arrays.
- `InitialStates` — Specified in a same way as parameters. Must be fourth input to the `idnlgrey` constructor.
- The command

  ```
  m = idnlgrey('filename',Order,Parameters,...
                InitialStates,Ts,P1,V1,...,PN,VN)
  ```

  specifies `idnlgrey` property-value pairs. See information on properties of `idnlgrey` objects below.

Estimate the parameters of this object using `pem`.

idnlgrey Properties

You can include property-value pairs in the model estimator or constructor to specify the model structure and estimation algorithm properties.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% Get the model time unit
get(m,'TimeUnit')
m.TimeUnit
```

The following table summarizes `idnlgrey` model properties. The general `idnlmodel` properties also apply to this nonlinear model object (see the corresponding reference pages).

Property Name	Description
`Algorithm`	A structure that specifies the estimation algorithm options, as described in “idnlgrey Algorithm Properties” on page 2-193.
`CovarianceMatrix`	Covariance matrix of the estimated `Parameters`. Assignable values: • `'None'` to omit computing uncertainties and save time during parameter estimation. • `'Estimate'` to estimation covariance. Symmetric and positive N_p-by-N_p matrix (or `[]`) where N_p is the number of free model parameters.
`EstimationInfo`	A read-only structure that stores estimation settings and results, as described in “idnlgrey EstimationInfo Properties” on page 2-203.
`FileArgument`	Contains auxiliary variables passed to the ODE file (M-file or MEX-file) specified in `FileName`. These variables may be used as extra inputs for specifying the state and/or output equations.`FileArgument` should be specified as a cell array. Default: `{}`.
`FileName`	File name string (without extension) or a function handle for computing the states and the outputs. If `'FileName'` is a string, then it must point to an M-file or MEX-file. For more information about the file variables, see “Specifying the Nonlinear Grey-Box Model Structure”.

Property Name	Description

`InitialStates`	An `Nx`-by-1 structure array with fields as follows. Here, `Nx` is the number of states of the model.
	
	• `Name`: Name of the state (a string). Default value is `'x#i'`, where `#i` is an integer in `[1, Nx]`.
	• `Unit`: Unit of the state (a string). Default value is `''`.
	• `Value`: Initial value of the initial state(s). Assignable values are:
	  - A finite real scalar
	  - A finite real 1-by-Ne vector, where `Ne` is the number of experiments in the data set to be used for estimation
	• `Minimum`: Minimum value of the initial state(s). Must be a real scalar/1-by-`Ne` vector of the same size as Value and such that Minimum <= Value for all components. Default value: `-Inf(size(Value))`.
	• `Maximum`: Maximum value of the initial state(s). Must be a real scalar/1-by-`Ne` vector of the same size as `Value` and such that `Value` <= `Maximum` for all components. Default value: `Inf(size(Value))`.
	• `Fixed`: Specifies which component(s) of the initial state(s) are fixed to their known values. Must be a Boolean scalar/1-by-`Ne` vector of the same size as `Value`. Default value: `true(size(Value))` (that is, do not estimate the initial states).
	
	For an `idnlgrey` model `M`, the `i`:th initial state is accessed through `M.InitialStates(i)` and its sub-fields as `M.InitialStates(i).FIELDNAME`.

Property Name	Description
`Order`	Structure with following fields: • `ny` — Number of outputs of the model structure. • `nu` — Number of inputs of the model structure. • `nx` — Number of states of the model structure. For time series, `nu` is 0. For static model structures, `nx` is 0.

`Parameters`	`Np`-by-1 structure array with information about the model parameters containing the following fields:
	• `Name`: Name of the parameter (a string). Default value is `'p#i'`, where `#i` is an integer in `[1, Np]`.
	• `Unit`: Unit of the parameter (a string). Default value is `''`.
	• `Value`: Initial value of the parameter(s). Assignable values are:
	  - A finite real scalar
	  - A finite real column vector
	  - A 2–dimensional real matrix
	• `Minimum`: Minimum value of the parameter(s). Must be a real scalar/column vector/matrix of the same size as `Value` and such that `Minimum <= Value` for all components. Default value: `-Inf(size(Value))`.
	• `Maximum`: Maximum value of the parameter(s). Must be a real scalar/column vector/matrix of the same size as `Value` and such that `Value <= Maximum` for all components. Default value: `Inf(size(Value))`.
	• `Fixed`: Specifies which component(s) of the parameter(s) are fixed to their known values. Must be a boolean scalar/column vector/matrix of the same size as `Value`. Default value: `false(size(Value))`, (estimate all parameter components).
	For an `idnlgrey` model `M`, the `i`:th parameter is accessed through `M.Parameters(i)` and its sub-fields as `M.Parameters(i).FIELDNAME`.

idnlgrey Algorithm Properties

The following table summarizes the fields of the `Algorithm` `idnlgrey` model properties. `Algorithm` is a structure that specifies the estimation-algorithm options.

Property Name	Description
`Advanced`	A structure that specifies additional estimation algorithm options, as described in "idnlgrey Advanced Algorithm Properties" on page 2-196.
`Criterion`	Specifies criterion used during minimization. `Criterion` can have the following values: • `'Det'`: Minimize `det(E'*E)` where `E` represents the prediction error. This is the optimal choice in a statistical sense and leads to the maximum likelihood estimates in case nothing is known about the variance of the noise. It uses the inverse of the estimated noise variance as the weighting function. This is the default criterion used for all models, except `idnlgrey` which uses `'Trace'` by default. • `'Trace'`: Minimize the trace of the weighted prediction error matrix `trace(E'*E*W)`, where `E` is the matrix of prediction errors, with one column for each output, and `W` is a positive semi-definite symmetric matrix of size equal to the number of outputs. By default, `W` is an identity matrix of size equal to the number of model outputs (so the minimization criterion becomes `trace(E'*E)`, or the traditional least-sum-of-squared-errors criterion. You can specify the relative weighting of prediction errors for each output using the `Weighting` field of the `Algorithm` property.
`LimitError`	Robustification criterion that limits the influence of large residuals, specified as a positive real value. Residual values that are larger than `'LimitError'` times the estimated residual standard deviation have a linear cost instead of the usual quadratic cost. Default: `0` (no robustification).

Property Name	Description
`MaxIter`	Maximum number of iterations for the estimation algorithm, specified as a positive integer. Default: `20`.
`SearchMethod`	Method used by the iterative search algorithm. Assignable values: • `'Auto'` — Automatically chooses from the following methods. • `'gn'` — Gauss-Newton method. • `'gna'` — Adaptive Gauss-Newton method. • `'grad'` — A gradient method. • `'lm'` — Levenberg-Marquardt method. • `'lsqnonlin'` — Nonlinear least-squares method (requires the Optimization Toolbox™ product). This method handles only the `'Trace'` criterion.
`Tolerance`	Specifies to terminate the iterative search when the expected improvement of the parameter values is less than `Tolerance`, specified as a positive real value in %. Default: `0.01`.
`GradientOptions`	A structure that specifies the options related to calculation of gradient of the cost, “idnlgrey Gradient Options” on page 2-202.
`SimulationOptions`	A structure that specifies the simulation method and related options, as described in “idnlgrey Simulation Options” on page 2-198.

Property Name	Description
`Trace`	Toggles displaying or hiding estimation progress information in the MATLAB® Command Window. Default: `'Off'`. Assignable values: • `'Off'` — Hide estimation information. • `'On'` — Display estimation information.
`Weighting`	Positive semi-definite matrix `W` used for weighted trace minimization. When `Criterion = 'Trace'`, `trace(E'*E*W)` is minimized. `Weighting` can be used to specify relative importance of outputs in multiple-input multiple-output models (or reliability of corresponding data) when `W` is a diagonal matrix of nonnegative values. `Weighting` is not useful in single-output models. By default, `Weighting` is an identity matrix of size equal to the number of outputs.

Note The `Criterion` property setting is meaningful in multiple-output cases only. In single-output models, the two criteria are equivalent. Both the `Det` and `Trace` criteria are derived from a general requirement of minimizing a weighted sum of least squares of prediction errors. The `Det` criterion can be interpreted as estimating the covariance matrix of the noise source and using the inverse of that matrix as the weighting. You should specify the weighting when using the `Trace` criterion.

If you want to achieve better accuracy for a particular channel in multiple-input multiple-output models, you should use `Trace` with weighting that favors that channel. Otherwise it is natural to use `Det`. When using `Det` you can check `cond(model.NoiseVariance)` after estimation. If the matrix is ill-conditioned, it may be more robust to use the `Trace` criterion. You can also use `compare` on validation data to check whether the relative error for different channels corresponds you your needs or expectations. Use the `Trace` criterion if you need to modify the relative errors, and check `model.NoiseVariance` to determine what weighting modifications to specify.

The search method of `lsqnonlin` supports the `Trace` criterion only.

idnlgrey Advanced Algorithm Properties

The following table summarizes the fields of the `Algorithm.Advanced` model properties. The fields in the `Algorithm.Advanced` structure specify additional estimation-algorithm options.

Property Name	Description
`GnPinvConst`	When the search direction is computed, the algorithm discards the singular values of the Jacobian that are smaller than `GnPinvConst*max(size(J))*norm(J)*eps`. Singular values that are closer to 0 are included when `GnPinvConst` is decreased. Default: `1e4`. Assign a positive, real value.
`LMStartValue`	(For Levenberg-Marquardt search algorithm) The starting level of *regularization* when using the Levenberg-Marquardt search method (`Algorithm.SearchMethod='lm'`). Default: `0.001`. Assign a positive real value.
`LMStep`	(For Levenberg-Marquardt search algorithm) Try this next level of *regularization* to get a lower value of the criterion function. The level of regularization is `LMStep` times the previous level. At the start of a new iteration, the level of regularization is computed as`1/LMStep` times the value from the previous iteration. Default: `10`. Assign a real value `>1`.
`MaxBisections`	Maximum number of bisections performed by the line search algorithm along the search direction (number of rotations of search vector for `'lm'`). Used by `'gn'`, `'lm'`, `'gna'` and `'grad'` search methods (`Algorithm.SearchMethod` property) Default: `10`. Assign a positive integer value.
`MaxFunEvals`	The iterations are stopped if the number of calls to the model file exceeds this value. Default: `Inf`. Assign a positive integer value.

Property Name	Description
`MinParChange`	The smallest parameter update allowed per iteration. Default: `1e-16`. Assign a positive, real value.
`RelImprovement`	The iterations are stopped if the relative improvement of the criterion function is less than `RelImprovement`. Default: `0`. Assign a positive real value. **Note** Does not apply to `Algorithm.SearchMethod='lsqnonlin'`
`StepReduction`	(For line search algorithm) The suggested parameter update is reduced by the factor `'StepReduction'` after each try until either `'MaxBisections'` tries are completed or a lower value of the criterion function is obtained. Default: `2`. Assign a positive, real value `>1`. **Note** Does not apply to `Algorithm.SearchMethod='lsqnonlin'`

idnlgrey Simulation Options

The following table summarizes the fields of `Algorithm.SimulationOptions` model properties.

Property Name	Description
`AbsTol`	Absolute error tolerance. This scalar applies to all components of the state vector. `AbsTol` applies only to the variable step solvers. Default: `1e-6`. Assignable value: A positive real value.
`FixedStep`	(For fixed-step time-continuous solvers) Step size used by the solver. Default: `'Auto'`. Assignable values: • `'Auto'` — Automatically chooses the initial step. • A real value such that `0<FixedStep<=1`.
`InitialStep`	(For variable-step time-continuous solvers) Specifies the initial step at which the ODE solver starts. Default: `'Auto'`. Assignable values: • `'Auto'` — Automatically chooses the initial step. • A positive real value such that `MinStep<=InitialStep<=MaxStep`.
`MaxOrder`	(For ode15s) Specifies the order of the Numerical Differentiation Formulas (NDF). Default: `5`. Assignable values: 1, 2, 3, 4 or 5.
`MaxStep`	(For variable-step time-continuous solvers) Specifies the largest time step of the ODE solver. Default: `'Auto'` — 1/15 of the simulation interval. Assignable values: • `'Auto'` — Automatically chooses the time step. • A positive real value > `MinStep`.

Property Name	Description
`MinStep`	(For variable-step time-continuous solvers) Specifies the smallest time step of the ODE solver. Default: `'Auto'`. Assignable values: • `'Auto'` — Automatically chooses the time step. • A positive real value < `MaxStep`.

Property Name	Description
`RelTol`	(For variable-step time-continuous solvers) Relative error tolerance that applies to all components of the state vector. The estimated error in each integration step satisfies `|e(i)| <= max(RelTol*abs(x(i)), AbsTol(i))`. Default: `1e-3` (0.1% accuracy). Assignable value: A positive real value.
`Solver`	ODE (Ordinary Differential/Difference Equation) solver for solving state space equations. A. Variable-step solvers for time-continuous `idnlgrey` models: • `'ode45'` — Runge-Kutta (4,5) solver for nonstiff problems. • `'ode23'` — Runge-Kutta (2,3) solver for nonstiff problems. • `'ode113'` — Adams-Bashforth-Moulton solver for nonstiff problems. • `'ode15s'` — Numerical Differential Formula solver for stiff problems. • `'ode23s'` — Modified Rosenbrock solver for stiff problems. • `'ode23t'` — Trapezoidal solver for moderately stiff problems. • `'ode23tb'` — Implicit Runge-Kutta solver for stiff problems. B. Fixed-step solvers for time-continuous `idnlgrey` models: • `'ode5'` — Dormand-Prince solver. • `'ode4'` — Fourth-order Runge-Kutta solver. • `'ode3'` — Bogacki-Shampine solver. • `'ode2'` — Heun or improved Euler solver. • `'ode1'` — Euler solver. C. Fixed-step solvers for time-discrete `idnlgrey` models: `'FixedStepDiscrete'` D. General: `'Auto'` — Automatically chooses one of the previous solvers (default).

idnlgrey Gradient Options

The following table summarizes the fields of the `Algorithm.GradientOptions` model properties. `Algorithm` is a structure that specifies the estimation-algorithm options.

Property Name	Description
`DiffMaxChange`	Largest allowed parameter perturbation when computing numerical derivatives. Default: `Inf`. Assignable value: A positive real value `>'DiffMinChange'`.
`DiffMinChange`	Smallest allowed parameter perturbation when computing numerical derivatives. Default: `0.01*sqrt(eps)`. Assignable value: A positive real value `<'DiffMaxChange'`.

Property Name	Description
`DiffScheme`	Method for computing numerical derivatives with respect to the components of the parameters and/or the initial state(s) to form the Jacobian. Default: `'Auto'` Assignable values: • `'Auto'` - Automatically chooses from the following methods. • `'Central approximation'` • `'Forward approximation'` • `'Backward approximation'`
`GradientType`	Method used when computing derivatives (Jacobian) of the parameters or the initial states to be estimated. Default: `'Auto'`. Assignable values: • `'Auto'` — Automatically chooses from the following methods. • `'Basic'` — Individually computes all numerical derivatives required to form each column of the Jacobian. • `'Refined'` — Simultaneously computes all numerical derivatives required to form each column of the Jacobian.

idnlgrey EstimationInfo Properties

The following table summarizes the fields of the `EstimationInfo` model properties. The read-only fields of the `EstimationInfo` structure store estimation settings and results.

Property Name	Description
`Status`	Shows whether the model parameters were estimated.

Property Name	Description
`Method`	Names of the solver and the optimizer used during estimation.
`LossFcn`	Value of the loss function, equal to `det(E'*E/N)`, where `E` is the residual error matrix (one column for each output) and `N` is the total number of samples. Provides a quantitative description of the model quality.
`FPE`	Value of Akaike's Final Prediction Error (see `fpe`).
`DataName`	Name of the data from which the model is estimated.
`DataLength`	Length of the estimation data.
`DataTs`	Sampling interval of the estimation data.
`DataDomain`	`'Time'` means time domain data. `'Frequency'` is not supported.
`DataInterSample`	Intersample behavior of the input estimation data used for interpolation: • `'zoh'` means zero-order-hold, or piecewise constant. • `'foh'` means first-order-hold, or piecewise linear.
`WhyStop`	Reason for terminating parameter estimation iterations.
`UpdateNorm`	Norm of the search vector (Gauss-Newton vector) at the last iteration. Empty when `'lsqnonlin'` is the search method.
`LastImprovement`	Criterion improvement in the last iteration, shown in %. Empty when `SearchMethod='lsqnonlin'` is the search method.
`Iterations`	Number of iterations performed by the estimation algorithm.
`InitialGuess`	Structure with the fields `InitialStates` and `Parameters`, specifying the values of these quantities before the last estimation.
`Warning`	Any warnings encountered during parameter estimation.

Definition of idnlgrey States

The states of an `idnlgrey` model are defined explicitly by the user in the M-file or MEX-file (as specified in the `FileName` property of the model) storing the model structure . The concept of states is useful for functions such as `sim`, `predict`, `compare`, and `findstates`.

Note The initial values of the states are configured by the `InitialStates` property of the `idnlgrey` model.

See Also

`pem`

`getinit`

`setinit`

`getpar`

`setpar`

idnlhw

Purpose

Class representing Hammerstein-Wiener input-output models

Syntax

```
m=idnlhw([nb nf nk])
m=idnlhw([nb nf nk],InputNL,OutputNL)
m=idnlhw([nb nf nk],InputNL,OutputNL,P1,V1,...,PN,VN)
```

Description

`idnlhw` is an object that stores Hammerstein-Wiener model properties, including model parameters.

Typically, you use the `nlhw` command to both specify the Hammerstein-Wiener model properties and estimate the model. You can specify the model properties directly in the `nlhw` syntax.

For information about the Hammerstein-Wiener model structure, see "Definition of the Hammerstein-Wiener Model".

The information in these reference pages summarizes the `idnlhw` model constructor and properties. It discusses the following topics:

- "idnlhw Constructor" on page 2-208
- "idnlhw Properties" on page 2-209
- "idnlhw Algorithm Properties" on page 2-211
- "idnlhw Advanced Algorithm Properties" on page 2-215
- "idnlhw EstimationInfo Properties" on page 2-217

Hammerstein-Wiener models describe dynamic systems using one or two static nonlinear blocks in series with a linear block. Only the linear block contains dynamic elements.

The linear block is a discrete-time transfer function and the nonlinear blocks are implemented using nonlinearity estimators, such as `saturation`, `wavenet`, and `deadzone`.

The input signal passes through the first nonlinear block, a linear block, and a second nonlinear block to produce the output signal, as shown in the following figure.

The following general equation describes the Hammerstein-Wiener structure:

$$w(t) = f(u(t))$$
$$x(t) = \frac{B_{j,i}(q)}{F_{j,i}(q)} w(t)$$
$$y(t) = h(x(t))$$

which contains the following variables:

- *u(t)* and *y(t)* are the inputs and outputs for the system, respectively.
- *f* and *h* are nonlinear functions that corresponding to the input and output nonlinearities, respectively.

  For multiple inputs and multiple outputs, *f* and *h* are defined independently for each input and output channel.
- *w(t)* and *x(t)* are internal variables that define the input and output of the linear block, respectively.

  w(t) has the same dimension as *u(t)*. *x(t)* has the same dimension as *y(t)*.
- *B(q)* and *F(q)* in the linear dynamic block are linear block, which are similar to the polynomial in an Output-Error model, as described in "What Are Black-Box Polynomial Models?".

  For *ny* outputs and *nu* inputs, the linear block is a transfer function matrix containing entries in the following form:

  $$\frac{B_{j,i}(q)}{F_{j,i}(q)}$$

  where $j = 1, 2, \ldots, ny$ and $i = 1, 2, \ldots, nu$.

If only the input nonlinearity is present, the model is called a Hammerstein model. If only the output nonlinearity is present, the model is called a Wiener model.

idnlhw Constructor

Typically, you use the `nlhw` estimator command to specify the model properties and estimate the Hammerstein-Wiener model. However, you can also use the `idnlhw` constructor to create the Hammerstein-Wiener model object in advance, and then estimate the parameters of this object using `pem`.

`m=idnlhw([nb nf nk])` creates an `idnlhw` object with the specified orders `nb`, `nf`, and input delays `nk`. It uses `sigmoidnet` as the default input and output nonlinearity estimators.

`m=idnlhw([nb nf nk],InputNL,OutputNL)` creates an `idnlhw` object with the specified input and output nonlinearity estimator.

`m=idnlhw([nb nf nk],InputNL,OutputNL,P1,V1,...,PN,VN)` creates an `idnlhw` object and specifies `idnlhw` property-value pairs. For more information about `idnlhw` properties, see "idnlarx Properties" on page 2-174.

The constructor arguments have the following specifications:

`[nb nf nk]`
: Model orders and input delays, where `nb` is the number of zeros plus 1, `nf` is the number of poles, and `nk` is the delay from input to output in terms of the number of samples.

  For `nu` inputs and `ny` outputs, `nb`, `nf` and, `nk` are `ny`-by-`nu` matrices whose *i*-*j*th entry specifies the orders and delay of the transfer function from the *j*th input to the *i*th output.

`InputNL` and `OutputNL`
: Specify the input and output nonlinearity estimator objects as one of the following: `pwlinear`, `deadzone`, `wavenet`, `saturation`, `customnet`, `sigmoidnet`, `poly1d`, and `unitgain`. The nonlinearity estimator objects have properties that you can set in the constructor, as follows:

```
m=idnlhw([2 2 1],sigmoidnet('num',5),deadzone([-1,2]))
```

To use default nonlinearity properties, specify the nonlinearity object name as a string. For example:

```
m=idnlhw([2 2 1],'sigmoidnet','deadzone')
m=idnlhw([2 2 1],'sig','dead') % Abbreviated
```

The estimator `unitgain` (can also be entered as `[]`) denotes the absence of nonlinearity. Thus, `m=idnlhw([2 2 1],'saturation',[])` gives a Hammerstein model (no output nonlinearities). For more information about nonlinearity properties, see the corresponding reference pages.

idnlhw Properties

You can include property-value pairs in the model estimator or constructor to specify the model structure and estimation algorithm properties.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% Get the model B parameters
get(m,'b')
% Get value of InputNonlinearity property
m.InputNonlinearity
```

The following table summarizes `idnlhw` model properties. The general `idnlmodel` properties also apply to this nonlinear model object (see the corresponding reference pages).

Property Name	Description
`Algorithm`	A structure that specifies the estimation algorithm options, as described in “idnlhw Algorithm Properties” on page 2-211.

Property Name	Description
`b`	*B* polynomial as a cell array of `Ny`-by-`Nu` elements, where `Ny` is the number of outputs and `Nu` is the number of inputs. An element `b{i,j}` is a row vector representing the numerator polynomial for the `j`:th input to `i`:th output transfer function. It contains as many leading zeros as there are input delays.
`f`	*F* polynomial as a cell array of `Ny`-by-`Nu` elements, where `Ny` is the number of outputs and `Nu` is the number of inputs. An element `f{i,j}` is a row vector representing the denominator polynomial for the `j`:th input to `i`:th output transfer function.
`LinearModel`	(Read only) The linear model is an Output-Error (OE) model. For single output, represented as an `idpoly` object. For muliple output, represented as an `idss` object.
`EstimationInfo`	A read-only structure that stores estimation settings and results, as described in "idnlhw EstimationInfo Properties" on page 2-217.
`InputNonlinearity`	Nonlinearity estimator object. Assignable values include `pwlinear`, `deadzone`, `wavenet`, `saturation`, `customnet`, `sigmoidnet`, `poly1d`, and `unitgain`. For more information, see the corresponding reference pages. For `ny` outputs, `Nonlinearity` is an `ny`-by-1 array, such as `[sigmoidnet;wavenet]`. However, if you specify a scalar object, this nonlinearity object applies to all outputs.

Property Name	Description
`OutputNonlinearity`	Same as `InputNonlinearity`.

`nb`
`nf`
`nk`
	Model orders and input delays, where `nb` is the number of zeros plus 1, `nf` is the number of poles, and `nk` is the delay from input to output in terms of the number of samples.
	For `nu` inputs and `ny` outputs, `nb`, `nf` and, `nk` are `ny`-by-`nu` matrices whose *i*-*j*th entry specifies the orders and delay of the transfer function from the *j*th input to the *i*th output.

idnlhw Algorithm Properties

The following table summarizes the fields of the `Algorithm` `idnlhw` model properties. `Algorithm` is a structure that specifies the estimation-algorithm options.

Property Name	Description
`Advanced`	A structure that specifies additional estimation algorithm options, as described in “idnlhw Advanced Algorithm Properties” on page 2-215.

Property Name	Description
`Criterion`	Specifies criterion used during minimization. `Criterion` can have the following values: • `'Det'`: Minimize `det(E'*E)`, where `E` represents the prediction error. This is the optimal choice in a statistical sense and leads to the maximum likelihood estimates in case nothing is known about the variance of the noise. It uses the inverse of the estimated noise variance as the weighting function. This is the default criterion used for all models, except `idnlgrey` which uses `'Trace'` by default. • `'Trace'`: Minimize the trace of the weighted prediction error matrix `trace(E'*E*W)`, where `E` is the matrix of prediction errors, with one column for each output, and `W` is a positive semi-definite symmetric matrix of size equal to the number of outputs. By default, `W` is an identity matrix of size equal to the number of model outputs (so the minimization criterion becomes `trace(E'*E)`, or the traditional least-squares criterion. You can specify the relative weighting of prediction errors for each output using the `Weighting` field of the `Algorithm` property.
`IterWavenet`	(For `wavenet` nonlinear estimator only) Toggles performing iterative or noniterative estimation. Default: `'auto'`. Assignable values: • `'auto'` — First estimation is noniterative and subsequent estimation are iterative. • `'On'` — Perform iterative estimation only. • `'Off'` — Perform noniterative estimation only.

Property Name	Description
`LimitError`	Robustification criterion that limits the influence of large residuals, specified as a positive real value. Residual values that are larger than `'LimitError'` times the estimated residual standard deviation have a linear cost instead of the usual quadratic cost. Default: `0` (no robustification).
`MaxIter`	Maximum number of iterations for the estimation algorithm, specified as a positive integer. Default: `20`.
`MaxSize`	The number of elements (size) of the largest matrix to be formed by the algorithm. Computational loops are used for larger matrices. Use this value for memory/speed trade-off. `MaxSize` can be any positive integer. Default: `250000`. **Note** The original data matrix of u and y must be smaller than `MaxSize`.
`SearchMethod`	Method used by the iterative search algorithm. Assignable values: • `'Auto'` — Automatically chooses from the following methods. • `'gn'` — Gauss-Newton method. • `'gna'` — Adaptive Gauss-Newton method. • `'grad'` — A gradient method. • `'lm'` — Levenberg-Marquardt method. • `'lsqnonlin'` — Nonlinear least-squares method (requires the Optimization Toolbox™ product). This method handles only the `'Trace'` criterion.

Property Name	Description
`Tolerance`	Specifies to terminate the iterative search when the expected improvement of the parameter values is less than `Tolerance`, specified as a positive real value in %. Default: `0.01`.
`Trace`	Toggles displaying or hiding estimation progress information in the MATLAB® Command Window. Default: `'Off'`. Assignable values: • `'Off'` — Hide estimation information. • `'On'` — Display estimation information.
`Weighting`	Positive semi-definite matrix `W` used for weighted trace minimization. When `Criterion = 'Trace'`, `trace(E'*E*W)` is minimized. `Weighting` can be used to specify relative importance of outputs in multiple-input multiple-output models (or reliability of corresponding data) when `W` is a diagonal matrix of nonnegative values. `Weighting` is not useful in single-output models. By default, `Weighting` is an identity matrix of size equal to the number of outputs.

Note The `Criterion` property setting is meaningful in multiple-output cases only. In single-output models, the two criteria are equivalent. Both the `Det` and `Trace` criteria are derived from a general requirement of minimizing a weighted sum of least squares of prediction errors. The `Det` criterion can be interpreted as estimating the covariance matrix of the noise source and using the inverse of that matrix as the weighting. You should specify the weighting when using the `Trace` criterion.

If you want to achieve better accuracy for a particular channel in multiple-input multiple-output models, you should use `Trace` with weighting that favors that channel. Otherwise it is natural to use `Det`. When using `Det` you can check `cond(model.NoiseVariance)` after estimation. If the matrix is ill-conditioned, it may be more robust to use the `Trace` criterion. You can also use `compare` on validation data to check whether the relative error for different channels corresponds you your needs or expectations. Use the `Trace` criterion if you need to modify the relative errors, and check `model.NoiseVariance` to determine what weighting modifications to specify.

The search method of `lsqnonlin` supports the `Trace` criterion only.

idnlhw Advanced Algorithm Properties

The following table summarizes the fields of the `Algorithm.Advanced` model properties. The fields in the `Algorithm.Advanced` structure specify additional estimation-algorithm options.

Property Name	Description
`GnPinvConst`	When the search direction is computed, the algorithm discards the singular values of the Jacobian that are smaller than `GnPinvConst*max(size(J))*norm(J)*eps`. Singular values that are closer to 0 are included when `GnPinvConst` is decreased. Default: `1e4`. Assign a positive, real value.
`LMStartValue`	(For Levenberg-Marquardt search algorithm) The starting level of *regularization* when using the Levenberg-Marquardt search method (`Algorithm.SearchMethod='lm'`). Default: `0.001`. Assign a positive real value.
`LMStep`	(For Levenberg-Marquardt search algorithm) Try this next level of *regularization* to get a lower value of the criterion function. The level of regularization is `LMStep` times the previous level. At the start of a new iteration, the level of regularization is computed as`1/LMStep` times the value from the previous iteration. Default: `10`. Assign a real value `>1`.
`MaxBisections`	Maximum number of bisections performed by the line search algorithm along the search direction (number of rotations of search vector for `'lm'`). Used by `'gn'`, `'lm'`, `'gna'` and `'grad'` search methods (`Algorithm.SearchMethod` property). Default: `10`. Assign a positive integer value.
`MaxFunEvals`	The iterations are stopped if the number of calls to the model file exceeds this value. Default: `Inf`. Assign a positive integer value.

Property Name	Description
`MinParChange`	The smallest parameter update allowed per iteration. Default: `1e-16`. Assign a positive, real value.
`RelImprovement`	The iterations are stopped if the relative improvement of the criterion function is less than `RelImprovement`. Default: `0`. Assign a positive real value. **Note** This does not apply when `Algorithm.SearchMethod='lsqnonlin'`.
`StepReduction`	(For line search algorithm) The suggested parameter update is reduced by the factor `'StepReduction'` after each try until either `'MaxBisections'` tries are completed or a lower value of the criterion function is obtained. Default: `2`. Assign a positive, real value `>1`. **Note** This does not apply when `Algorithm.SearchMethod='lsqnonlin'`.

idnlhw EstimationInfo Properties

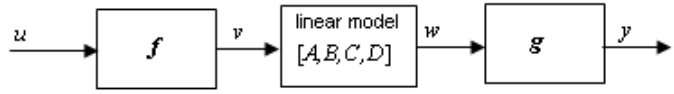
The following table summarizes the fields of the `EstimationInfo` model properties. The read-only fields of the `EstimationInfo` structure store estimation settings and results.

Property Name	Description
`Status`	Shows whether the model parameters were estimated.

Property Name	Description
`Method`	Shows the estimation method.
`LossFcn`	Value of the loss function, equal to `det(E'*E/N)`, where `E` is the residual error matrix (one column for each output) and `N` is the total number of samples.
`FPE`	Value of Akaike's Final Prediction Error (see `fpe`).
`DataName`	Name of the data from which the model is estimated.
`DataLength`	Length of the estimation data.
`DataTs`	Sampling interval of the estimation data.
`DataDomain`	`'Time'` means time domain data. `'Frequency'` is not supported.
`DataInterSample`	Intersample behavior of the input estimation data used for interpolation: • `'zoh'` means zero-order-hold, or piecewise constant. • `'foh'` means first-order-hold, or piecewise linear.
`WhyStop`	Reason for terminating parameter estimation iterations.
`UpdateNorm`	Norm of the search vector (gn-vector) in the last iteration. Empty when `'lsqnonlin'` is the search method.
`LastImprovement`	Criterion improvement in the last iteration, shown in %. Empty when `'lsqnonlin'` is the search method.
`Iterations`	Number of iterations performed by the estimation algorithm.
`Warning`	Any warnings encountered during parameter estimation.
`InitRandState`	The value of `randn('state')` at the last randomization of the initial parameter vector.
`EstimationTime`	Duration of the estimation.

Definition of idnlhw States

The Hammerstein-Weiner model structure consists of static nonlinearities around linear model as shown in the following figure.



The dynamic element of the model consists of the linear model alone. The states of the `idnlhw` model are the states of this linear component of the model. If this linear model is not a state-space structure (such as with single-input single-output cases), the states are defined as those of model `Mss` where `Mss = idss(Model.LinearModel)` and `Model` is the `idnlhw` object. The concept of states is useful for functions such as `sim`, `predict`, `compare`, `findstates`, `data2state`, `findop`, and `linearize`.

See Also

`pem`

`predict(idnlhw)`

idnlmodel

Purpose

Superclass for nonlinear models

Description

You do not use the `idnlmodel` class directly. Instead, `idnlmodel` defines the common properties and methods inherited by its subclasses, `idnlarx`, `idnlgrey`, and `idnlhw`.

idnlmodel Properties

The following table lists the properties shared by the `idnlarx`, `idnlgrey`, and `idnlhw`, defined in terms of `Ny` outputs and `Nu` inputs.

Property Name	Description
`InputName`	Specifies the names of individual input channels. Default: `{'u1';'u2';...;'uNu'}`. Assignable values: • For single-output models, a string. For example, `'torque'`. • For multiple-output models, an `nu`-by-`1` cell array. For example: `{'thrust'; 'aileron deflection'}`
`InputUnit`	Specifies the units of each input channel. Default: `''`. Assignable values: • For single-output models, a string. For example, `'m/s'`. • For multiple-output models, an `nu`-by-`1` cell array.
`Name`	Name of the model, specified as a string.
`NoiseVariance`	Noise variance (covariance matrix) of the model innovations *e*. Assignable value is an `Ny`-by-`Ny` matrix. Typically set automatically by the estimation algorithm.

Property Name	Description
`OutputName`	Specifies the names of individual output channels. Default: `{'y1';'y2';...;'yNy'}`. Assignable values: • For single-output models, a string. For example, `'torque'`. • For multiple-output models, an `nu`-by-`1` cell array. For example: `{'thrust'; 'aileron deflection'}`
`OutputUnit`	Specifies the units of each output channel. Default: `''`. Assignable values: • For single-output models, a string. For example, `'m/s'`. • For multiple-output models, an `nu`-by-`1` cell array.
`TimeUnit`	Unit of the sampling interval and time vector, specified as a string. Default: `''`.
`TimeVariable`	Independent variable for the inputs, outputs, and—when available—internal states, specified as a string. Default: `'t'` (time).
`Ts`	Sampling interval with the unit specified by `TimeUnit`. Default: `1`. Assignable values: • For discrete-time models, positive scalar value of the sampling interval. • For continuous-time models, `0`.

See Also

idnlarx

idnlgrey

idnlhw

Purpose

Class for storing linear polynomial input-output models

Syntax

```
m = idpoly(A,B)
m = idpoly(A,B,C,D,F,NoiseVariance,Ts)
m = idpoly(A,B,C,D,F,NoiseVariance,Ts,'Property1',Value1,...
        'PropertyN',ValueN)
m = idpoly(mi)
```

Description

`idpoly` creates a model object containing parameters that describe the general multiple-input single-output model structure.

$$A(q)y(t) = \frac{B_1(q)}{F_1(q)}u_1(t-nk_1) + \ldots + \frac{B_{nu}(q)}{F_{nu}(q)}u_{nu}(t-nk_{nu}) + \frac{C(q)}{D(q)}e(t)$$

`A`, `B`, `C`, `D`, and `F` specify the polynomial coefficients.

For single-input systems, these are all row vectors.

```
A = [1 a1 a2 ... ana]
```

consequently describes

$$A(q) = 1 + a_1 q^{-1} + \ldots + a_{na} q^{-na}$$

`A`, `C`, `D`, and `F` all start with 1, while `B` contains leading zeros to indicate the delays. See “What Are Black-Box Polynomial Models?”.

For multiple-input systems, `B` and `F` are matrices with one row for each input.

For time series, `B` and `F` are entered as empty matrices.

```
B = [];  F = [];
```

`NoiseVariance` is the variance of the white noise sequence $e(t)$, while `Ts` is the sampling interval.

Trailing arguments `C`, `D`, `F`, `NoiseVariance`, and `Ts` can be omitted, in which case they are taken as 1. (If `B = []`, then `F` is taken as `[]`.) The property name/property value pairs can start directly after `B`.

`Ts = 0` means that the model is a continuous-time one. Then the interpretation of the arguments is that

```
A = [1 2 3 4]
```

corresponds to the polynomial $s^3 + 2s^2 + 3s + 4$ in the Laplace variable s, and so on. For continuous-time systems, `NoiseVariance` indicates the level of the spectral density of the innovations. A sampled version of the model has the innovations variance `NoiseVariance/Ts`, where `Ts` is the sampling interval. The continuous-time model must have a white noise component in its disturbance description. See "Understanding Spectrum Normalization".

For discrete-time models (`Ts > 0`), note the following: `idpoly` strips any trailing zeros from the polynomials when determining the orders. It also strips leading zeros from the `B` polynomial to determine the delays. Keep this in mind when you use `idpoly` and `polydata` to modify earlier estimates to serve as initial conditions for estimating new structures.

`idpoly` can also take any single-output `idmodel` or LTI object `mi` as an input argument. If an LTI system has an input group with name `'Noise'`, these inputs are interpreted as white noise with unit variance, and the noise model of the `idpoly` model is computed accordingly.

Properties

- `na`, `nb`, `nc`, `nd`, `nf`, `nk`: The orders and delays of the polynomials. Integers or row vectors of integers.
- `a`, `b`, `c`, `d`, `f`: The polynomials, described by row vectors and matrices as detailed above.
- `da`, `db`, `dc`, `dd`, `df`: The estimated standard deviation of the polynomials. Cannot be set.
- `'InitialState'`: How to deal with the initial conditions that are required to compute the prediction of the output. Possible values are:

- `'Estimate'`: The necessary initial states are estimated from data as extra parameters.
- `'Backcast'`: The necessary initial states are estimated by a backcasting (backward filtering) process, described in Knudsen (1994).
- `'Zero'`: All initial states are taken as zero.
- `'Auto'`: An automatic choice among the above is made, guided by the data.

In addition, any `idpoly` object also has all the properties of `idmodel`. See `idmodel` properties and `Algorithm Properties`.

Note that you can set or retrieve all properties either with the `set` and `get` commands or by subscripts. Autofill applies to all properties and values, and these are case insensitive.

```
m.a=[1 -1.5 0.7];
set(m,'ini','b')
p = roots(m.a)
```

For a complete list of property values, use `get(m)`. To see possible value assignments, use `set(m)`.

Examples

To create a system of ARMAX, type

```
A = [1 -1.5 0.7];
B = [0 1 0.5];
C = [1 -1 0.2];
m0 = idpoly(A,B,C);
```

This gives a system with one delay (`nk = 1`).

Create the continuous-time model

$$y(t) = \frac{1}{s(s+1)}u_1(t) + \frac{s+3}{s^2+2s+4}u_2(t) + e(t)$$

Sample it with `T = 0.1` and then simulate it without noise.

```
B=[0 1;1 3];
F=[1 1 0;1 2 4]
m = idpoly(1,B,1,1,F,1,0)
md = c2d(m,0.1)
y = sim(md,[u1 u2])
```

Note that the continuous-time model is automatically sampled to the sampling interval of the data, when simulated, so the above is also achieved by

```
u = iddata([],[u1 u2],0.1)
y = sim(m,u)
```

idpoly Definition of States

The states of an `idpoly` model are defined as those corresponding to the model obtained by converting them to the state-space format using the `idss` command. For example, if you have an `idpoly` model defined by `m1 = idpoly([1 2 1],[2 2]);` , then the initial states of this model correspond to those of `m2 = idss(m1)`. The concept of states is useful for functions such as `sim`, `predict`, `compare` and `findstates`.

References

Ljung (1999) Section 4.2 for the model structure family.

Knudsen, T., (1994), "New method for estimating ARMAX models," In *Proc. 10th IFAC Symposium on System Identification,* pp. 611-617, Copenhagen, Denmark, for the backcast method.

See Also

`idss`

`sim`

"Extracting Parameter Values from Linear Models"

Purpose

Class for storing low-order, continuous-time process models

Syntax

```
m = idproc(Type)
m = idproc(Type,'Property1',Value1,...,'PropertyN',ValueN)
m = pem(Data,Type) % to directly estimate an idproc model
```

Description

The function `idproc` is used to create typical simple, continuous-time process models as `idproc` objects. The model has one output, but can have several inputs.

The character of the model is defined by the argument `Type`. This is an acronym made up of the following symbols:

- P: All `'Type'` acronyms start with this letter.
- 0, 1, 2, or 3: This integer denotes the number of time constants (poles) to be modeled. Possible integrations (poles in the origin) are not included in this number.
- I: The letter I is included to mark that an integration is enforced (self-regulation process).
- D: The letter D is used to mark that the model contains a time delay (dead time).
- Z: The letter Z is used to mark an extra numerator term: a zero.
- U: The letter U is included to mark that underdamped modes (complex-valued poles) are permitted. If U is not included, all poles are restricted to be real.

This means, for example, that `Type = 'P1D'` corresponds to the model with transfer function

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

while `Type = 'POI'` is

$$G(s)= \frac{K_p}{s}$$

and `Type = 'P3UZ'` is

$$G(s)= K_p \frac{1+T_z s}{(1+2\varsigma T_w s+(T_w s)^2)(1+T_{p3}s)}$$

For multiple-input systems, `Type` is a cell array where each cell describes the character of the model from the corresponding input, like `Type = {'P1D','POI'}` for the two-input model

$$Y(s) = \frac{K_p(1)}{1+sT_{p1}(1)}e^{-T_d s}U_1(s)+\frac{K_p(2)}{s}U_2(s) \qquad \textbf{(2-1)}$$

The parameters of the model are

- `Kp`: The static gain
- `Tp1`, `Tp2`, `Tp3`: The real-time constants (corresponding to poles in `1/Tp1`, etc.)
- `Tw` and `Zeta`: The "resonance time constant" and the damping factor corresponding to a denominator factor `(1+2 Zeta Tw s + (Tw s)^2)`. If underdamped modes are allowed, `Tw` and `Zeta` replace `Tp1` and `Tp2`. A third real pole, `Tp3`, could still be included.
- `Td`: The time delay
- `Tz`: The numerator zero

These properties contain fields that give the values of the parameters, upper and lower bounds, and information whether they are locked to zero, have a fixed value, or are to be estimated. For multiple-input models, the number of entries in these fields equals the number of inputs. This is described in more detail below.

The `idproc` object is a child of `idmodel`. Therefore any `idmodel` properties can be set as property name/property value pairs in the

idproc command. They can also be set by the command set, or by subassignment, as in

```
m.InputName = {'speed','voltage'}
m.kp = 12
```

In the multiple-input case, models for specific inputs can be obtained by regular subreferencing.

```
m(ku)
```

There are also two properties, DisturbanceModel and InitialState, that can be used to expand the model. See below.

idproc Properties

- Type: A string or a cell array of strings with as many elements as there are inputs. The string is an acronym made up of the characters P, Z, I, U, D and an integer between 0 and 3. The string must start with P, followed by the integer, while possible other characters can follow in any order. The integer is the number of poles (not counting a possible integration), Z means the inclusion of a numerator zero, D means inclusion of a time delay, while U marks that the modes can be underdamped (a pair of complex conjugated poles). I means that an integration in the model is enforced.
- Kp, Tp1, Tp2, Tp3, Tw, Zeta, Tz, Td: These are the parameters as explained above. Each of these is a structure with the following fields:
  - value: Numerical value of the parameter.
  - max: Maximum allowed value of the parameter when it is estimated.
  - min: Minimum allowed value of the parameter when it is estimated. For multiple-input models, these are row vectors.
  - status: Assumes one of 'Estimate', 'Fixed', or 'Zero'.

    'Zero' means that the parameter is locked to zero and not included in the model. Assigning, for example, Type = 'P1' means that the status of Tp2, Tp3, Tw, and Zeta will be 'Zero'.

The value `'Fixed'` means that the parameter is fixed to its value, and will not be estimated.

The value `'Estimate'` means that the parameter value should be estimated.

For multiple-input modes, `status` is a cell array with one element for each input, while `value`, `max`, and `min` are row vectors.

- `DisturbanceModel`: Allows an additive disturbance model as in

$$y(t) = G(s)u(t) + \frac{C(s)}{D(s)}e(t) \qquad \textbf{(2-2)}$$

where $G(s)$ is a process model and $e(t)$ is white noise, and C/D is a first- or second-order transfer function.

`DisturbanceModel` can assume the following values:

  - `'None'`: This is the default. No disturbance model is included (that is, $C=D=1$).
  - `'arma1'`: The disturbance model is a first-order ARMA model (that is, C and D are first-order polynomials).
  - `'arma2'` or `'Estimate'`: The disturbance model is a second-order ARMA model (that is, C and D are second-order polynomials).

When a disturbance model has been estimated, the property `DisturbanceModel` is returned as a cell array, with the first entry being the status as just defined, and the second entry being the actual model, delivered as a continuous-time `idpoly` object.

- `InitialState`: Affects the parameterization of the initial values of the states of the model. It assumes the same values as for other models:
  - `'Zero'`: The initial states are fixed to zero.
  - `'Estimate'`: The initial states are treated as parameters to be estimated.

  - `'Backcast'`: The initial state vector is adjusted, during the parameter estimation step, to a suitable value, but it is not stored.
  - `'Auto'`: Makes a data-dependent choice among the values above.
- `InputLevel`: The offset level of the input signal(s). This is of particular importance for those input channels that contain an integration. `InputLevel` will then define the level from which the integration takes place, and that cannot be handled by estimating initial states. `InputLevel` has the same structure as the model parameters `Kp`, etc., and thus contains the following fields:
  - `value`: Numerical value of the parameter. For multiple-input models, this is a row vector.
  - `max`: Maximum allowed value of the parameter when it is estimated.
  - `min`: Minimum allowed value of the parameter when it is estimated. For multiple-input models, these are row vectors.
  - `status`: Assumes one of `'Estimate'`, `'Fixed'`, or `'Zero'` with the same interpretations.

In addition, any `idproc` object also has all the properties of `idmodel`. See `Algorithm Properties`, `EstimationInfo`, and `idmodel`.

Note that all properties can be set or retrieved using either the `set` and `get` commands or subscripts. Autofill applies to all properties and values, and these are case insensitive. Also `'u'` and `'y'` are short for `'Input'` and `'Output'`, respectively. You can also set all properties at estimation time as property name/property value pairs in the call to `pem`. An extended syntax allows direct setting of the fields of the parameter values, so that assigning a numerical value is automatically attributed to the value field, while a string is attributed to the status field.

```
m.kp = 10
m.tp1 = 'estimate'
% Initializing the parameter Kp at the value 10
m = pem(Data,'P1D','kp',10)
% Fixing the parameter Kp to the value 10
```

```
m = pem(Data,'P1D','kp',10,'kp','fix')
% constraining Kp to lie between 3 and 4
m = pem(Data,'P2U','kp',{'max',4},'kp',{'min',3})
% For two inputs, estimate the offset level
% of the first input
m = pem(Data,{'P2I','P1D',},'ulevel',{'est','zer'})
% estimate a noise model
m = pem(Data,'P2U','dist','est')
% Use a fixed noisemodel,
% given by the continuous-time idpoly model noimod
m = pem(Data,'P2U','dist',{'fix',noimod})
% (minimum Kp for the second input)
m.kp.min(2) = 12
% fixing the gain for the second input.
m.kp.status{2} = 'fix'
```

For a complete list of property values, use `get(m)`. To see possible value assignments, use `set(m)`.

idproc Definition of States

The states of an `idproc` model are defined as those corresponding to the model obtained by converting them to the state-space format using the `idss` command. For example, if you have an `idproc` model defined by `m1 = idproc('P1D');`, then the initial states of this model correspond to those of `m2 = idss(m1)`. The concept of states is useful for functions such as `sim`, `predict`, `compare` and `findstates`.

Examples

```
m = pem(Data,'P2D','dist','arma1')
```

Purpose

Resample time-domain data by decimation or interpolation

Syntax

```
datar = idresamp(data,R)
datar = idresamp(data,R,order,tol)
[datar,res_fact] = idresamp(data,R,order,tol)
```

Description

`datar = idresamp(data,R)` resamples data on a new sample interval `R` and stores the resampled data as `datar`.

`datar = idresamp(data,R,order,tol)` filters the data by applying a filter of specified `order` before interpolation and decimation. Replaces `R` by a rational approximation that is accurate to a tolerance `tol`.

`[datar,res_fact] = idresamp(data,R,order,tol)` returns `res_fact`, which corresponds to the value of `R` approximated by a rational expression.

Input

`data`

Name of time-domain `iddata` object or a matrix of data. Can be input-output or time-series data.

Data must be sampled at equal time intervals.

`R`

Resampling factor, such that `R>1` results in decimation and `R<1` results in interpolation.

Any positive number you specify is replaced by the rational approximation, `Q/P`.

`order`

Order of the filters applied before interpolation and decimation.

Default: `8`

`tol`

Tolerance of the rational approximation for the resampling factor `R`.

Smaller tolerance might result in larger `P` and `Q` values, which produces more accurate answers at the expense of slower computation.

Default: `0.1`

Output

`datar`
Name of the resampled data variable. `datar` class matches the `data` class, as specified.

`res_fact`
Rational approximation for the specified resampling factor `R` and tolerance `tol`.

Any positive number you specify is replaced by the rational approximation, `Q/P`, where the data is interpolated by a factor `P` and then decimated by a factor `Q`.

See Also

`resample`

Purpose

Class for storing linear state-space models with known and unknown parameters

Syntax

```
m = idss(A,B,C,D)
m = idss(A,B,C,D,K,x0,Ts,'Property1',Value1,...
           'PropertyN',ValueN)
mss = idss(m1)
```

Description

The function `idss` is used to construct state-space model structures with various parameterizations. It is a complement to `idgrey` and deals with parameterizations that do not require the user to write a special M-file. Instead it covers parameterizations that are either `'Free'`, that is, all parameters in the A, B, and C matrices can be adjusted freely, or `'Canonical'`, meaning that the matrices are parameterized as canonical forms. The parameterization can also be `'Structured'`, which means that certain elements in the state-space matrices are free to be adjusted, while others are fixed. This is explained below.

`Ts` is the sampling interval. `Ts = 0` means a continuous-time model. The default is `Ts = 1`.

The `idss` object `m` describes state-space models in innovations form of the following kind:

$$\tilde{x}(t) = A(\theta)x(t) + B(\theta)u(t) + K(\theta)e(t)$$

$$x(0) = x_0(\theta)$$

$$y(t) = C(\theta)x(t) + D(\theta)u(t) + e(t)$$

Here $\tilde{x}(t)$ is the time derivative $\dot{x}(t)$ for a continuous-time model and $x(t+Ts)$ for a discrete-time model.

The model `m` will contain information both about the nominal/initial values of the A, B, C, D, K, and X0 matrices and about how these matrices are parameterized by the parameter vector θ (to be estimated).

The nominal model is defined by `idss(A,B,C,D,K,X0)`. If `K` and `X0` are omitted, they are taken as zero matrices of appropriate dimensions.

Defining an idss object from a given model,

```
mss = idss(m1)
```

constructs an idss model from any idmodel or LTI system m1.

If m1 is an LTI system (ss, tf, or zpk) that has no InputGroup called 'Noise', the corresponding state-space matrices A, B, C, D are used to define the idss object. The Kalman gain K is then set to zero.

If the LTI system has an InputGroup called 'Noise', these inputs are interpreted as white noise with a covariance matrix equal to the identity matrix. The corresponding Kalman gain and noise variance are then computed and entered into the idss model together with A, B, C, and D.

Parameterizations

There are several different ways to define the parameterization of the state-space matrices. The parameterization determines which parameters can be adjusted to data by the parameter estimation routine pem.

- Free black-box parameterizations: This is the default situation and corresponds to letting all parameters in A, B, and C be freely adjustable. You do this by setting the property 'SSParameterization' = 'Free'. The parameterizations of D, K, and $X0$ are then determined by the following properties:
  - 'nk': A row vector of the same length as the number of inputs. The kuth element is the delay from input channel number ku. Thus nk = [0,...,0] means that there is no delay from any of the inputs, and that consequently all elements of the D matrix should be estimated. nk =[1,...,1] means that there is a delay of 1 from each input, so that the D matrix is fixed to be zero.
  - 'DisturbanceModel': This property affects the parameterization of K and can assume the following values:

    'Estimate': All elements of the K matrix are to be estimated.

    'None': All elements of K are fixed to zero.

'Fixed': All elements of *K* are fixed to their nominal/initial values.

- 'InitialState': Affects the parameterization of *X0* and can assume the following values:

  'Auto': An automatic choice of the following is made, depending on data (default).

  'Estimate': All elements of *X0* are to be estimated.

  'Zero': All elements of *X0* are fixed to zero.

  'Fixed': All elements of *X0* are fixed to their nominal/initial values.

  'Backcast': The vector *X0* is adjusted, during the parameter estimation step, to a suitable value, but it is not stored as an estimation result.

- Canonical black-box parameterizations: You do this by setting the property 'SSParameterization' = 'Canonical'. The matrices *A*, *B*, and *C* are then parameterized as an observer canonical form, which means that ny (number of output channels) rows of *A* are fully parameterized while the others contain 0's and 1's in a certain pattern. The *C* matrix is built up of 0's and 1's while the *B* matrix is fully parameterized. See Equation (A.16) in Ljung (1999) for details. The exact form of the parameterization is affected by the property 'CanonicalIndices'. The default value 'Auto' is a good choice. The parameterization of the *D*, *K*, and *X0* matrices in this case is determined by the properties 'nk', 'DisturbanceModel', and 'InitialState'.

- Arbitrarily structured parameterizations: The general case, where arbitrary elements of the state-space matrices are fixed and others can be freely adjusted, corresponds to the case 'SSParameterization' = 'Structured'. The parameterization is determined by the idss properties As, Bs, Cs, Ds, Ks, and X0s. These are the *structure matrices* that are "shadows" of the state-space matrices, so that an element in these matrices that is equal to NaN indicates a freely adjustable parameter, while a numerical value

in these matrices indicates that the corresponding system matrix element is fixed (nonadjustable) to this value.

idss Properties

- `SSParameterization` has the following possible values:
  - `'Free'`: Means that all parameters in *A*, *B*, and *C* are freely adjustable, and the parameterizations of *D*, *K*, and *X0* depend on the properties `'nk'`, `'DisturbanceModel'`, and `'InitialState'`.
  - `'Canonical'`: Means that *A* and *C* are parameterized as an observer canonical form. The details of this parameterization depend on the property `'CanonicalIndices'`. The *B* matrix is always fully parameterized, and the parameterizations of *D*, *K*, and *X0* depend on the properties `'nk'`, `'DisturbanceModel'`, and `'InitialState'`.
  - `'Structured'`: Means that the parameterization is determined by the properties (the structure matrices) `'As'`, `'Bs'`, `'Cs'`, `'Ds'`, `'Ks'`, and `'X0s'`. A `NaN` in any position in these matrices denotes a freely adjustable parameter, and a numeric value denotes a fixed and nonadjustable parameter.
- `nk`: A row vector with as many entries as the number of input channels. The entry number `k` denotes the time delay from input number `k` to `y(t)`. This property is relevant only for `'Free'` and `'Canonical'` parameterizations. If any delay is larger than 1, the structure of the *A*, *B*, and *C* matrices will accommodate this delay, at the price of a higher-order model.
- `DisturbanceModel` has the following possible values:
  - `'Estimate'`: Means that the *K* matrix is fully parameterized.
  - `'None'`: Means that the *K* matrix is fixed to zero. This gives a so-called output-error model, since the model output depends on past inputs only.
  - `'Fixed'`: Means that the *K* matrix is fixed to the current nominal values.
- `InitialState` has the following possible values:

  - `'Estimate'`: Means that *X0* is fully parameterized.
  - `'Zero'`: Means that *X0* is fixed to zero.
  - `'Fixed'`: Means that *X0* is fixed to the current nominal value.
  - `'Backcast'`: The value of *X0* is estimated by the identification routines as the best fit to data, but it is not stored.
  - `'Auto'`: Gives an automatic and data-dependent choice among `'Estimate'`, `'Zero'`, and `'Backcast'`.

- `A`, `B`, `C`, `D`, `K`, and `X0`: The state-space matrices that can be set and retrieved at any time. These contain both fixed values and estimated/nominal values.
- `dA`, `dB`, `dC`, `dD`, `dK`, and `dX0`: The estimated standard deviations of the state-space matrices. These cannot be set, only retrieved. Note that these are not defined for an `idss` model with `'Free'` `SSParameterization`. You can then convert the parameterization to `'Canonical'` and study the uncertainties of the matrix elements in that form.
- `As`, `Bs`, `Cs`, `Ds`, `Ks`, and `X0s`: These are the structure matrices that have the same sizes as `A, B, C,` etc., and show the freely adjustable parameters as `NaN`s in the corresponding position. These properties are used to define the model structure for `'SSParameterization' = 'Structured'`. They are always defined, however, and can be studied also for the other parameterizations.
- `CanonicalIndices`: Determines the details of the canonical parameterization. It is a row vector of integers with as many entries as there are outputs. They sum up to the system order. This is the so-called pseudocanonical multiindex with an exact definition, for example, on page 132 in Ljung (1999). A good default choice is `'Auto'`. This property is relevant only for the canonical parameterization case. Note however, that for `'Free'` parameterizations, the estimation algorithms also store a canonically parameterized model to handle the model uncertainty.

In addition to these properties, `idss` objects also have all the properties of the `idmodel` object. See `idmodel` properties, `Algorithm Properties`, and `EstimationInfo`.

Note that all properties can be set and retrieved either by the `set` and `get` commands or by subscripts. Autofill applies to all properties and values, and these are case insensitive.

```
m.ss='can'
set(m,'ini','z')
p = eig(m.a)
```

For a complete list of property values, use `get(m)`. To see possible value assignments, use `set(m)`.

Examples

Define a continuous-time model structure corresponding to

$$\dot{x} = \begin{bmatrix} \theta_1 & 0 \\ 0 & \theta_2 \end{bmatrix} x + \begin{bmatrix} \theta_3 \\ \theta_4 \end{bmatrix} u$$

$$y = \begin{bmatrix} 1 & 1 \end{bmatrix} x + e$$

with initial values

$$\theta = \begin{bmatrix} -0.2 \\ -0.3 \\ 2 \\ 4 \end{bmatrix}$$

and estimate the free parameters.

```
A = [-0.2, 0; 0, -0.3]; B = [2;4]; C=[1, 1]; D = 0
m0 = idss(A,B,C,D);
m0.As = [NaN,0;0,NaN];
m0.Bs = [NaN;NaN];
m0.Cs = [1,1];
```

```
m0.Ts = 0;
m = pem(z,m0);
```

Estimate a model in free parameterization. Convert it to continuous time, then convert it to canonical form and continue to fit this model to data.

```
m1 = n4sid(data,3);
m1 = d2c(m1);
m1.ss ='can';
m = pem(data,m1);
```

All of this can be done at once by

```
m = pem(data,3,'ss','can','ts',0)
```

See Also

`n4sid`

`pem`

`setstruc`

Purpose

Transform `iddata` objects from frequency to time domain

Syntax

```
dat = ifft(Datf)
```

Description

`ifft` transforms a frequency-domain `iddata` object to the time domain. It requires the frequencies on `Datf` to be equally spaced from frequency 0 to the Nyquist frequency. This means that if there are `N` frequencies in `Datf` and the time sampling interval is `Ts`, then

`Datf.Frequency = [0:df:F]`, where `F` is `pi/Ts` if `N` is odd and `F = pi/Ts*(1-1/N)` if `N` is even.

See Also

`iddata`

`fft`

Purpose

Plot impulse response with confidence interval

Syntax

```
impulse(m)
impulse(data)
impulse(m,'sd',sd,Time)
impulse(m,'sd',sd,Time,'fill')
impulse(data,'sd',sd,'pw',na,Time)
impulse(m1,m2,...,dat1, ...,mN,Time,'sd',sd)
impulse(m1,'PlotStyle1',m2,'PlotStyle2',...,dat1,'PlotStylek',...,
mN,'PlotStyleN',Time,'sd',sd)
[y,t,ysd] = impulse(m)
mod = impulse(data)
```

Description

`impulse` can be applied both to `idmodels` and to `iddata` sets, as well as to any mixture.

For a discrete-time `idmodel` `m`, the impulse response `y` and, when required, its estimated standard deviation `ysd`, are computed using `sim`. When called with output arguments, `y`, `ysd`, and the time vector `t` are returned. When `impulse` is called without output arguments, a plot of the impulse response is shown. If `sd` is given a value larger than zero, a confidence region around zero is drawn. It corresponds to the confidence of `sd` standard deviations. In the plots, the impulse is inversely scaled with the sampling interval so that it has the same energy regardless of the sampling interval.

Adding an argument `'fill'` among the input arguments gives an uncertainty region marked by a filled area rather than by dash-dotted lines.

Setting the Time Interval

You can specify the start time `T1` and the end time `T2` using `Time= [T1 T2]`. If `T1` is not given, it is set to `-T2/4`. The negative time lags (the impulse is always assumed to occur at time 0) show possible feedback effects in the data when the impulse is estimated directly from data. If `Time` is not specified, a default value is used.

Estimating the Impulse Response from data

For an `iddata` set data, `impulse(data)` estimates a high-order, noncausal FIR model after first having prefiltered the data so that the input is "as white as possible." The impulse response of this FIR model and, when asked for, its confidence region, are then plotted. Note that it is not always possible to deliver the demanded time interval when the response is estimated. A warning is then issued. When called with an output argument, `impulse`, in the `iddata` case, returns this FIR model, stored as an `idarx` model. The order of the prewhitening filter can be specified by the property name/property value pair `'pw'/na`. The default value is `na = 10`.

Several Models/Data Sets

Any number and any mixture of models and data sets can be used as input arguments. The responses are plotted with each input/output channel (as defined by the model and data set `InputName` and `OutputName` properties) as a separate plot. Colors, line styles, and marks can be defined by `PlotStyle` values. These are the same as for the regular `plot` command, as in

```
impulse(m1,'b-*',m2,'y--',m3,'g')
```

Noise Channels

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `m.NoiseVariance` = Λ. The model can also be described with unit variance, normalized noise source v:

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

- `impulse(m)` plots the impulse response of the transfer function *G*.
- `impulse(m('n'))` plots the impulse response of the transfer function *H* (*ny* inputs and *ny* outputs). The input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- If `m` is a time series, that is *nu* = `0`, `impulse(m)`, plots the impulse response of the transfer function *H*.
- `impulse(noisecnv(m))` plots the impulse response of the transfer function [*G H*] (*nu*+*ny* inputs and *ny* outputs). The noise input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- `impulse(noisecnv(m,'norm'))p` lots the impulse response of the transfer function [*G HL*] (*nu*+*ny* inputs and *ny* outputs). The noise input channels have names `v@yname`, where `yname` is the name of the corresponding output.

Arguments

If `impulse` is called with a single `idmodel` `m`, the output argument `y` is a 3-D array of dimension `Nt`-by-`ny`-by-`nu`. Here `Nt` is the length of the time vector `t`, `ny` is the number of output channels, and `nu` is the number of input channels. Thus `y(:,ky,ku)` is the response in output `ky` to an impulse in the `ku`th input channel.

`ysd` has the same dimensions as `y` and contains the standard deviations of `y`.

If `impulse` is called with an output argument and a single data set in the input arguments, the output is returned as an `idarx` model `mod` containing the high-order FIR model and its uncertainty. By calling `impulse` with `mod`, the responses can be displayed and returned without your having to redo the estimation.

Examples

`impulse(data,'sd',3)` estimates and plots the impulse response. To take a closer look at subsystems, do the following:

```
mod = impulse(data)
impulse(mod(2,3),'sd',3)
```

See Also

cra

step

Purpose

Set or randomize initial parameter values

Syntax

```
m = init(m0)
m = init(m0,R,pars,sp)
```

Description

This function randomizes initial parameter estimates for model structures `m0` for any `idmodel`, `idnlarx`, and `idnlhw` model object. `m` is the same model structure as `m0`, but with a different nominal parameter vector. This vector is used as the initial estimate by `pem`.

The parameters are randomized around `pars` with variances given by the row vector `R`. Parameter number k is randomized as `pars(k) + e*sqrt(R(k))`, where `e` is a normal random variable with zero mean and a variance of 1. The default value of `R` is all ones, and the default value of `pars` is the nominal parameter vector in `m0`.

Only models that give stable predictors are accepted. If `sp = 'b'`, only models that are both stable and have stable predictors are accepted.

`sp = 's'` requires stability only of the model, and `sp = 'p'` requires stability only of the predictor. `sp = 'p'` is the default.

Sufficiently free parameterizations can be stabilized by direct means without any random search. To just stabilize such an initial model, set `R = 0`. With `R > 0`, randomization is also done.

For model structures where a random search is necessary to find a stable model/predictor, a maximum of 100 trials is made by `init`. It can be difficult to find a stable predictor for high-order systems by trial and error.

See Also

`idnlarx`

`idnlhw`

`idmodel`

`pem`

isreal

Purpose

Determine whether model parameters or data values are real

Syntax

```
isreal(Data)
isreal(Model)
```

Description

`Data` is an `iddata` set and `Model` is any `idmodel`. The `isreal` function returns `1` if all parameters of the model are real and if all signals of the data set are real.

See Also

`realdata`

Purpose

Estimate AR model using instrumental variable method returning `idpoly` object

Syntax

```
m = ivar(y,na)
m = ivar(y,na,nc,maxsize)
```

Description

The parameters of an AR model structure

$$A(q)y(t) = v(t)$$

are estimated using the instrumental variable method. `y` is the signal to be modeled, entered as an `iddata` object (outputs only). `na` is the order of the A polynomial (the number of A parameters). The resulting estimate is returned as an `idpoly` model `m`. The routine is for scalar time-domain signals only.

In the above model, $v(t)$ is an arbitrary process, assumed to be a moving average process of order `nc`, possibly time varying. (Default is `nc = na`.) Instruments are chosen as appropriately filtered outputs, delayed `nc` steps.

The optional argument `maxsize` is explained under `Algorithm Properties`.

Examples

Compare spectra for sinusoids in noise, estimated by the IV method and by the forward-backward least squares method.

```
y = iddata(sin([1:500]'*1.2) + sin([1:500]'*1.5) +
0.2*randn(500,1),[]);
miv = ivar(y,4);
mls = ar(y,4);
bode(miv,mls)
```

References

Stoica, P., et al., *Optimal Instrumental variable estimates of the AR-parameters of an ARMA process*, IEEE Trans. Autom. Control, Vol. AC-30, 1985, pp. 1066-1074.

See Also

Algorithm Properties

EstimationInfo

ar

arx

etfe

idpoly

pem

spa

step

Purpose

Compute loss functions for sets of output-error model structures

Syntax

```
v = ivstruc(ze,zv,NN)
v = ivstruc(ze,zv,NN,p,maxsize)
```

Description

`NN` is a matrix that defines a number of different structures of the ARX type. Each row of `NN` is of the form

```
nn = [na nb nk]
```

with the same interpretation as described for `arx`. See `struc` for easy generation of typical `NN` matrices for single-input systems.

`ze` and `zv` are `iddata` objects containing output-input data. Only time-domain data is supported. Models for each model structure defined in `NN` are estimated using the instrumental variable (IV) method on data set `ze`. The estimated models are simulated using the inputs from data set `zv`. The normalized quadratic fit between the simulated output and the measured output in `zv` is formed and returned in `v`. The rows below the first row in `v` are the transpose of `NN`, and the last row contains the logarithms of the condition numbers of the IV matrix

$$\sum \zeta(t)\varphi^{T}(t)$$

A large condition number indicates that the structure is of unnecessarily high order (see Ljung, L. *System Identification: Theory for the User*, Upper Saddle River, NJ, Prentice-Hal PTR, 1999, p. 498).

The information in `v` is best analyzed using `selstruc`.

If `p` is equal to zero, the computation of condition numbers is suppressed. For the use of `maxsize`, see `Algorithm Properties`.

The routine is for single-output systems only.

Note The IV method used does not guarantee that the models obtained are stable. The output-error fit calculated in `v` can then be misleading.

Examples

Compare the effect of different orders and delays, using the same data set for both the estimation and validation.

```
v = ivstruc(z,z,struc(1:3,1:2,2:4));
nn = selstruc(v)
m = iv4(z,nn);
```

Algorithm

A maximum-order ARX model is computed using the least squares method. Instruments are generated by filtering the input(s) through this model. The models are subsequently obtained by operating on submatrices in the corresponding large IV matrix.

References

Ljung, L. *System Identification: Theory for the User*, Upper Saddle River, NJ, Prentice-Hal PTR, 1999.

See Also

`arxstruc`

`iv4`

`selstruc`

`struc`

Purpose

Estimate parameters of ARX model using instrumental variable method with arbitrary instruments returning `idpoly` or `idarx` object

Syntax

```
m = ivx(data,orders,x)
m = ivx(data,orders,x,maxsize)
```

Description

`ivx` is a routine analogous to the `iv4` routine, except that you can use arbitrary instruments. These are contained in the matrix `x`. Make this the same size as the output, `data.y`. In particular, if data contains several experiments, `x` must be a cell array with one matrix/vector for each experiment. The instruments used are then analogous to the regression vector itself, except that `y` is replaced by `x`.

Note that `ivx` does not return any estimated covariance matrix for `m`, since that requires additional information. `m` is returned as an `idpoly` object for single-output systems and as an `idarx` object for multiple-output systems.

Use `iv4` as the basic IV routine for ARX model structures. The main interest in `ivx` lies in its use for nonstandard situations, for example, when there is feedback present in the data, or when other instruments need to be tried out. Note that there is also an IV version that automatically generates instruments from certain filters you define (type `help iv`).

References

Ljung (1999), page 222.

See Also

`Algorithm Properties`

`EstimationInfo`

`arx`

`idarx`

`idpoly`

```
iv4
pem
```

Purpose

Estimate ARX model using four-stage instrumental variable method returning `idpoly` or `idarx` object

Syntax

```
m = iv4(data,orders)
m = iv4(data,'na',na,'nb',nb,'nk',nk)
m= iv4(data,orders,'Property1',Value1,...,'PropertyN',ValueN)
```

Description

This function is an alternative to `arx` and the use of the arguments is entirely analogous to the `arx` function. The main difference is that the procedure is not sensitive to the color of the noise term $e(t)$ in the model equation.

Examples

Here is an example of a two-input, one-output system with different delays on the inputs u_1 and u_2.

```
z = iddata(y, [u1 u2]);
nb = [2 2];
nk = [0 2];
m= iv4(z,[2 nb nk]);
```

Algorithm

The first stage uses the `arx` function. The resulting model generates the instruments for a second-stage IV estimate. The residuals obtained from this model are modeled as a high-order AR model. At the fourth stage, the input-output data is filtered through this AR model and then subjected to the IV function with the same instrument filters as in the second stage.

For the multiple-output case, optimal instruments are obtained only if the noise sources at the different outputs have the same color. The estimates obtained with the routine are reasonably accurate, however, even in other cases.

References

Ljung (1999), equations (15.21) through (15.26).

See Also

Algorithm Properties

EstimationInfo

arx

idarx

idpoly

ivx

pem

Purpose

Linear approximation of nonlinear ARX and Hammerstein-Wiener models for given input

Syntax

```
lm = linapp(nlmodel,u)
lm = linapp(nlmodel,umin,umax,nsample)
```

Input

`nlmodel`
: Name of the `idnlarx` or `idnlhw` model object you want to linearize.

`u`
: Input signal as an iddata object or a real matrix.

  Dimensions of `u` must match the number of inputs in `nlmodel`.

`[umin,umax]`
: Minimum and maximum input values for generating white-noise input with a magnitude in this rectangular range. The sample length of this signal is `nsample`.

`nsample`
: Optional argument when you specify `[umin,umax]`. Specified the length of the white-noise input. Default: `1024`.

Description

`lm = linapp(nlmodel,u)` computes a linear approximation for a nonlinear black-box model for a given input.

`lm = linapp(nlmodel,umin,umax,nsample)` computes a linear approximation of a nonlinear black-box model for a generated input. The input signal is specified by the magnitude range and (optionally) the number of samples.

The following table summarizes the linear model objects that store the linear approximation for each type of nonlinear model and the number of outputs.

Nonlinear Model Type	Number of Outputs	Linear Model Object
`idnlarx`	Single output	`idpoly`

Nonlinear Model Type	Number of Outputs	Linear Model Object
idnlarx	Multiple outputs	idarx
idnlhw	Single output	idpoly
idnlhw	Multiple outputs	idss

Remarks

linapp computes the best linear approximation—in an mean-square-error sense—of a nonlinear ARX or Hammerstein-Wiener model for a given input or a randomly generated input in a specified range.

See Also

idarx

idnlarx

idnlhw

idpoly

idss

Purpose

Specify to estimate nonlinear ARX model that is linear in (nonlinear) custom regressors

Syntax

```
lin=linear
lin=linear('Parameters',Par)
```

Description

`linear` is an object that specifies that the nonlinear ARX model is linear in custom (nonlinear) regressors. You define custom regressors using `customreg`.

`lin=linear` instantantiates the `linear` object.

`lin=linear('Parameters',Par)` instantantiates the `linear` object and specifies optional values in the `Par` structure. For more information about this structure, see "linear Properties" on page 2-259.

Remarks

`linear` is a linear (affine) function $y = F(x)$, defined as follows:

$$F(x) = xL + d$$

y is scalar, and x is a 1-*by*-`m` vector.

Use `evaluate(lin,x)` to compute the value of the function defined by the `linear` object `lin` at `x`.

linear Properties

You can include property-value pairs in the constructor to specify the object.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% List Parameters values
get(lin)
% Get value of Parameters property
lin.Parameters
```

Property Name	Description
`Parameters`	Structure containing the following fields: • `LinearCoef`: `m`-*by*-1 vector L. • `OutputOffset`: Scalar `d`.

Examples

To specify that the nonlinear ARX model is linear in custom regressors, first specify one or more `customreg` objects. Then, include the `linear` object in the `nlarx` estimator command.

For example, to estimate a nonlinear ARX model linear in the custom regressors that might be nonlinear in input `u` and output `y` data, use the following syntax:

```
m=nlarx(Data,Orders,linear,'custom',{'y(t-1)*u(t-2)'});
```

See Also

`customreg`

`nlarx`

Purpose

Linearize nonlinear ARX model

Syntax

```
SYS = linearize(NLSYS,U0,X0)
```

Description

`SYS = linearize(NLSYS,U0,X0)` linearizes a nonlinear ARX model about the specified operating point `U0` and `X0`. The linearization is based on tangent linearization. For more information about the definition of states for `idnlarx` models, see "Definition of idnlarx States" on page 2-184.

Input

- `NLSYS`: `idnlarx` model.
- `U0`: Matrix containing the constant input values for the model.
- `X0`: Model state values. The states of a nonlinear ARX model are defined by the time-delayed samples of input and output variables. For more information about the states of nonlinear ARX models, see the `getDelayInfo(idnlarx)` reference page.

Note To estimate `U0` and `X0` from operating point specifications, use the `findop(idnlarx)` command.

Input

- `SYS` is an `idss` model.

  When the Control System Toolbox™ product is installed, `SYS` is an LTI object .

Algorithm

The following equations govern the dynamics of an `idnlarx` model:

$$X(t+1) = AX(t) + B\tilde{u}(t)$$
$$y(t) = f(X,u)$$

where $X(t)$ is a state vector, $u(t)$ is the input, and $y(t)$ is the output. A and B are constant matrices. $\tilde{u}(t)$ is $[y(t), u(t)]^T$.

The output at the operating point is given by

$y^* = f(X^*, u^*)$

where X^* and u^* are the state vector and input at the operating point.

The linear approximation of the model response is as follows:

$$\Delta X(t+1) = (A + B_1 f_X)\Delta X(t) + (B_1 f_u + B_2)\Delta u(t)$$
$$\Delta y(t) = f_X \Delta X(t) + f_u \Delta u(t)$$

where

- $\Delta X(t) = X(t) - X^*(t)$
- $\Delta u(t) = u(t) - u^*(t)$
- $\Delta y(t) = y(t) - y^*(t)$
- $B\tilde{U} = [B_1, B_2]\begin{bmatrix} Y \\ U \end{bmatrix} = B_1 Y + B_2 U$
- $f_X = \frac{\partial}{\partial X} f(X,U)\Big|_{X^*,U^*}$
- $f_U = \frac{\partial}{\partial U} f(X,U)\Big|_{X^*,U^*}$

Note For linear approximations over larger input ranges, use `linapp`. For more information, see the `linapp` reference page.

Example

In this example, you linearize a nonlinear ARX model around an operating point corresponding to a simulation snapshot at a specific time. Create an `idnlarx` model estimated using sample data.

1 Load sample data:

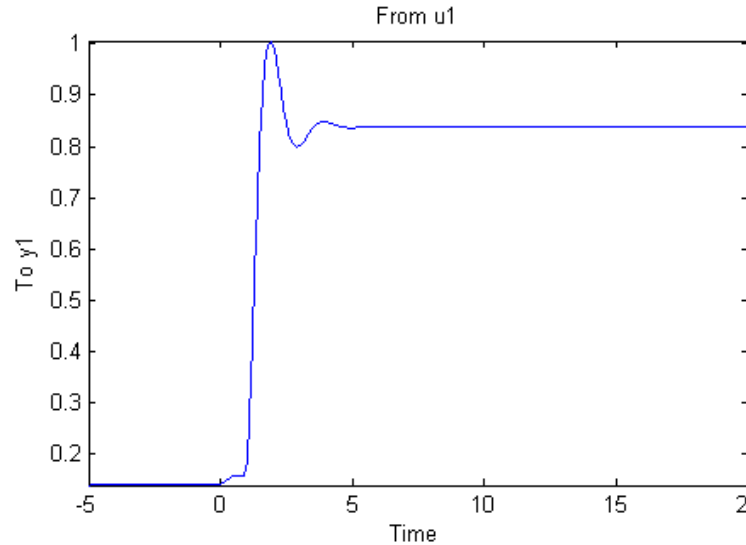
```
load iddata2
```

2 Estimate `idnlarx` model from sample data:

```
model = nlarx(z2,[4 3 10],'tree','custom',...
    {'sin(y1(t-2)*u1(t))+y1(t-2)*u1(t)+u1(t).*u1(t-13)',...
    'y1(t-5)*y1(t-5)*y1(t-1)'},'nlr',[1:5, 7 9]);
```

3 Plot the response of the model for a step input:

```
step(model, 20)
```



The step response shows that the model reached a steady state at `T = 20` seconds.

4 Compute the operating point corresponding to `T = 20`.

```
stepinput = iddata([],[zeros(10,1);ones(200,1)],model.Ts);

 % Compute operating point.
[x,u] = findop(model,'snapshot',20,stepinput);
```

5 Linearize the model about the operating point corresponding to the model snapshot at `T=20`.

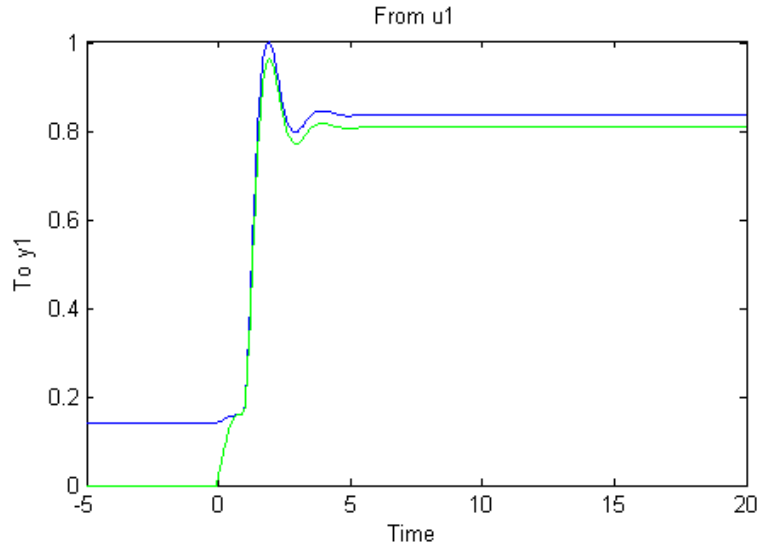
```
sys = linearize(model,u,x)
```

The transfer function from input u_1 to output y_1 displays in the MATLAB® Command Window.

```
 0.02679+0.02681z^-10+0.06429z^-11+0.03942z^-12+0.01117z^-13
-------------------------------------------------------------
1-0.8898z^-1-0.2084z^-2+0.08166z^-3+0.2263z^-4-0.001715z^-5

Sampling time: 0.1
```

6 Compare the step responses of the nonlinear model and the linearized model.

```
step(model,sys,20) % compare step responses.
```



The linearized model response tracks the nonlinear model output.

See Also

`linapp`

`findop(idnlarx)`

`getDelayInfo(idnlarx)`

linearize(idnlhw)

Purpose

Linearize Hammerstein-Wiener model

Syntax

```
SYS = linearize(NLSYS,U0)
SYS = linearize(NLSYS,U0,X0)
```

Description

`SYS = linearize(NLSYS,U0)` linearizes a Hammerstein-Wiener model about an equilibrium operating point. When using this syntax, equilibrium state values for the linearization are calculated automatically using `U0`.

`SYS = linearize(NLSYS,U0,X0)` linearizes the `idnlhw` model `NLSYS` about the operating point specified by the input `U0` and state values `X0`. In this usage, `X0` need not contain equilibrium state values. For more information about the definition of states for `idnlhw` models, see "Definition of idnlhw States" on page 2-219.

The output is a linear model that is the best linear approximation for inputs that vary in a small neighborhood of a constant input $u(t) = U$. The linearization is based on tangent linearization.

Input

- `NLSYS`: `idnlhw` model.
- `U0`: Matrix containing the constant input values for the model.
- `X0`: Operating point state values for the model.

Note To estimate `U0` and `X0` from operating point specifications, use the `findop(idnlhw)` command.

Input

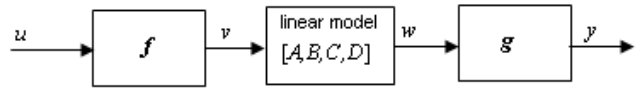
- `SYS` is an `idss` model.

  When the Control System Toolbox™ product is installed, `SYS` is an LTI object .

Algorithm

The `idnlhw` model structure represents a nonlinear system using a linear system connected in series with one or two static nonlinear

systems. For example, you can use a static nonlinearity to simulate saturation or dead-zone behavior. The following figure shows the nonlinear system as a linear system that is modified by static input and output nonlinearities, where function ***f*** represents the input nonlinearity, ***g*** represents the output nonlinearity, and [*A*,*B*,*C*,*D*] represents a state-space parameterization of the linear model.



The following equations govern the dynamics of an `idnlhw` model:

$v(t) = f(u(t))$

$X(t+1) = AX(t)+Bv(t)$

$w(t) = CX(t)+Dv(t)$

$y(t) = g(w(t))$

where

- u is the input signal
- v and w are intermediate signals (outputs of the input nonlinearity and linear model respectively)
- y is the model output

The linear approximation of the Hammerstein-Weiner model about an operating point (X^*, u^*) is as follows:

$$\Delta X(t+1) = A\Delta X(t) + Bf_u\Delta u(t)$$
$$\Delta y(t) \approx g_w C\Delta X(t) + g_w Df_u\Delta u(t)$$

where

- $\Delta X(t) = X(t) - X^*(t)$
- $\Delta u(t) = u(t) - u^*(t)$

- $\Delta y(t) = y(t) - y^*(t)$
- $f_u = \frac{\partial}{\partial u} f(u)\Big|_{u=u^*}$
- $g_w = \frac{\partial}{\partial w} g(w)\Big|_{w=w^*}$

where y^* is the output of the model corresponding to input u^* and state vector X^*, $v^* = f(u^*)$, and w^* is the response of the linear model for input v^* and state X^*.

Note For linear approximations over larger input ranges, use `linapp`. For more information, see the `linapp` reference page.

Examples

In this example you linearize a Hammerstein-Wiener model with two inputs at an equilibrium point, and compare the linearized model response to the original model response.

1 Load the sample data to create `iddata` object `z`.

```
load iddata2
load iddata3
z2.Ts = z3.Ts;
z = [z2(1:300),z3]; % Estimation data
```

2 Estimate an `idnlhw` model using a combination of `pwlinear`, `poly1d`, `sigmoidnet` and `customnet` nonlinearities.

```
orders = [2 2 3 4 1 5; 2 5 1 2 5 2];
model = nlhw(z,orders,[pwlinear;poly1d],...
             [sigmoidnet;customnet(@gaussunit)]);
```

3 Linearize the model at an equilibrium operating point corresponding to input levels of `10` and `5` respectively. To do this you first compute

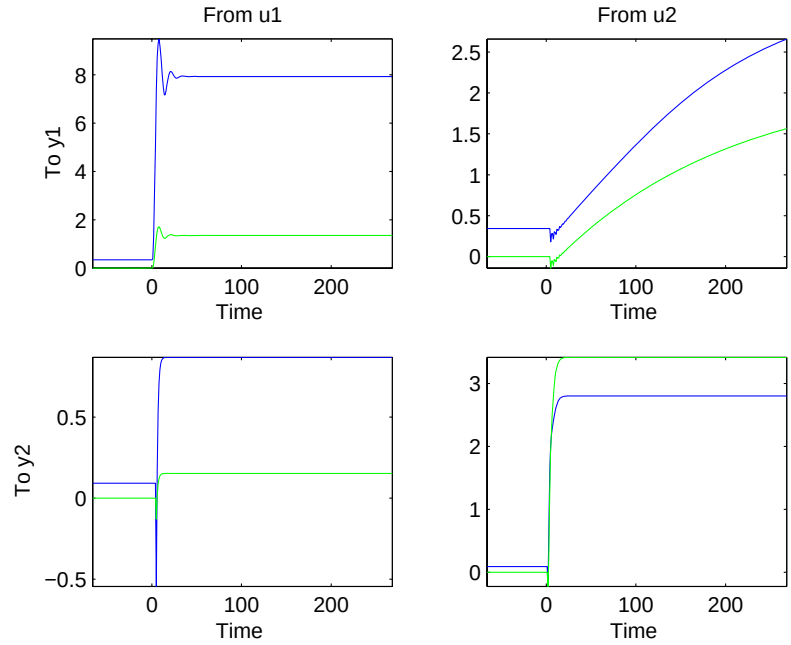
the operating point using `findop`, then linearize the model about the computed input and state values.

```
[x,u] = findop(model,'steady',[10,5]);
sys = linearize(model,u,x); %sys is a state-space model
```

4 Compare the step response of the original model to that of the linearized model.

```
step(model,sys)
```

MATLAB® software responds with the following plots.



The step response of original model is closely tracked by that of the linearized model.

See Also

`linapp`

`findop(idnlhw)`

Purpose

Apply Control System Toolbox™ commands to `idmodel` objects

Syntax

```
append, augstate, balreal, canon, d2d, feedback, inv, minreal,
modred, norm, parallel, series, ss2ss
```

Description

When you have the Control System Toolbox product installed, you can apply the listed LTI commands to `idmodel` objects, including `idarx`, `idgrey`, `idpoly`, and `idss` models. You can also use the overloaded operations +, -, and *. The same operations are performed and the result is delivered as an `idmodel`. The original covariance information is lost most of the time, however.

Examples

You have two more or less identical processes connected in series. Estimate a model for one of them, and use that to form an initial estimate for a model of the connected process.

```
% data concerns one of the processes
m = pem(data)
% data2 is from the entire connected process
m2 = pem(data2,m*m)
```

merge (iddata)

Purpose

Merge data sets into one iddata object

Syntax

```
dat = merge(dat1,dat2,....,datN)
```

Description

dat collects the data sets in dat1,.. datN into one iddata object, with several *experiments*. The number of experiments in dat will be the sum of the number of experiments in datk. For the merging to be allowed, a number of conditions must be satisfied:

- All of datk must have the same number of input channels, and the InputNames must be the same.
- All of datk must have the same number of output channels, and the OutputNames must be the same. If some input or output channel is lacking in one experiment, it can be replaced by a vector of NaNs to conform with these rules.
- If the ExperimentNames of datk have been specified as something other than the default 'Exp1', 'Exp2', etc., they must all be unique. If default names overlap, they are modified so that dat will have a list of unique ExperimentNames.

The sampling intervals, the number of observations, and the input properties (Period, InterSample) might be different in the different experiments.

You can retrieve the individual experiments by using the command getexp. You can also retrieve them by subreferencing with a fourth index.

```
dat1 = dat(:,:,:,ExperimentNumber)
```

or

```
dat1 = dat(:,:,:,ExperimentName)
```

Storing multiple experiments as one iddata object can be very useful for handling experimental data that has been collected on different

occasions, or when a data set has been split up to remove “bad” portions of the data. All the toolbox routines accept multiple-experiment data.

Examples

Bad portions of data have been detected around sample 500 and between samples 720 to 730. Cut out these bad portions and form a multiple-experiment data set that can be used to estimate models without the bad data destroying the estimate.

```
dat = merge(dat(1:498),dat(502:719),dat(719:1000))
m = pem(dat)
```

Use the first two parts to estimate the model and the third one for validation.

```
m = pem(getexp(dat,[1,2]));
compare(getexp(dat,3),m)
```

See also `iddemo` #9.

See Also

`iddata`

`getexp`

Purpose

Merge estimated `idmodel` models

Syntax

```
m = merge(m1,m2,....,mN)
[m,tv] = merge(m1,m2)
```

Description

The models `m1,m2,...,mN` must all be of the same structure, just differing in parameter values and covariance matrices. Then `m` is the merged model, where the parameter vector is a statistically weighted mean (using the covariance matrices to determine the weights) of the parameters of `mk`.

When two models are merged,

```
[m, tv] = merge(m1,m2)
```

returns a test variable `tv`. It is χ^2 distributed with `n` degrees of freedom, if the parameters of `m1` and `m2` have the same means. Here `n` is the length of the parameter vector. A large value of `tv` thus indicates that it might be questionable to merge the models.

Merging models is an alternative to merging data sets and estimating a model for the merged data. Consequently,

```
m1 = arx(z1,[2 3 4]);
m2 = arx(z2,[2 3 4]);
ma = merge(m1,m2);
```

and

```
mb = arx(merge(z1,z2),[2 3 4]);
```

lead to models `ma` and `mb` that are related and should be close. The difference is that merging the data sets assumes that the signal-to-noise ratios are about the same in the two experiments. Merging the models allows one model to be much more uncertain, for example, due to more disturbances in that experiment. If the conditions are about the same, we recommend that you merge data rather than models, since this is more efficient and typically involves better conditioned calculations.

Purpose

Set directory for storing `idprefs.mat` containing GUI startup information

Syntax

```
midprefs
midprefs(path)
```

Description

The graphical user interface `ident` allows a large number of variables for customized choices. These include the window layout, the default choices of plot options, and names and directories of the four most recent sessions with `ident`. This information is stored in the file `idprefs.mat`, which should be placed on the user's `MATLABPATH`. The default, automatic location for this file is in the same directory as the user's `startup.m` file.

`midprefs` is used to select or change the directory where you store `idprefs.mat`. Either type `midprefs` and follow the instructions, or give the directory name as the argument. Include all directory delimiters, as in the PC case

```
midprefs('c:\matlab\toolbox\local\')
```

or in the UNIX® case

```
midprefs('/home/ljung/matlab/')
```

See Also

`ident`

misdata

Purpose

Reconstruct missing input and output data

Syntax

```
Datae = misdata(Data)
Datae = misdata(Data,Model)
Datae = misdata(Data,Maxiter,Tol)
```

Description

`Data` is time-domain input-output data in the `iddata` object format. Missing data samples (both in inputs and in outputs) are entered as `NaN`s.

`Datae` is an `iddata` object where the missing data has been replaced by reasonable estimates.

`Model` is any `idmodel` (`idarx`, `idgrey`, `idpoly`, `idss`) used for the reconstruction of missing data.

If no suitable model is known, it is estimated in an iterative fashion using default order state-space models.

`Maxiter` is the maximum number of iterations carried out (the default is 10). The iterations are terminated when the difference between two consecutive data estimates differs by less than `tol`%. The default value of `tol` is `1`.

Algorithm

For a given model, the missing data is estimated as parameters so as to minimize the output prediction errors obtained from the reconstructed data. See Section 14.2 in Ljung (1999). Treating missing outputs as parameters is not the best approach from a statistical point of view, but is a good approximation in many cases.

When no model is given, the algorithm alternates between estimating missing data and estimating models, based on the current reconstruction.

Purpose

Class representing neural network object created in Neural Network Toolbox™ product for estimating nonlinear ARX and Hammerstein-Wiener models

Syntax

```
n=neuralnet(Network)
```

Description

`neuralnet` is an object that stores the neural network nonlinearity estimator for estimating nonlinear ARX and Hammerstein-Wiener models.

You can use the constructor to create the nonlinearity object, as follows:

`n=neuralnet(Network)` creates a neural network nonlinearity estimator based on the network object you created using the Neural Network Toolbox product.

The neural network must meet the following requirements:

- Neural network must be created using the Neural Network Toolbox `newff` or `newcf` command (feedforward networks used for function approximation).
- Neural network must represent a static mapping between the inputs and the output. It should not contain I/O delays or feedback.
- Neural network must have one output. If you want to use neural networks for multiple-output nonlinear ARX model, you must assign a separate `neuralnet` estimator for each output—that is, each estimator must represent a single-output network object.

Use `evaluate(n,x)` to compute the value of the function defined by the `neuralnet` object `n` at `x`.

Remarks

Use `neuralnet` to define a nonlinear function $y = F(x)$, where F is a multilayer feedforward neural network, as defined in the Neural Network Toolbox documentation.

y is a scalar and x is an `m`-dimensional row vector.

When you have installed the Neural Network Toolbox product, you can create a multilayer feedforward neural network using `newff`:

```
ff = newff(P,T,[nL_1,nL_2,..,nL_r],{tf_1,tf_2,...,tf_r})
```

where `P` is an `m`-*by*-`N` matrix containing inputs `x`, and `T` is a `1`-*by*-`N` matrix containing output (target) values for one of the model outputs. You can also use `newcf`

There are `r+1` layers and `nL_k` neurons in the `k`th layer, except for the last layer. The last layer has one neuron assigned automatically, such that `nL_(r+1)=1`. The transfer function (or unit function) in the `k`th layer is `tf_k`.

If `m` is unknown at the time of creation of the network, use `P = zeros(0,N)` with arbitrary `N>0`. After this initialization, `m` is adjusted to the estimation data by `nlarx` or `nlhw`. Similarly, you can set `T` to any vector (number of rows=1).

neuralnet Properties

You include the property as an argument in the constructor to specify the object.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% List Network property value
get(n)
n.Network
```

You can also use the `set` function to set the value of particular properties. For example:

```
set(d, 'Network', net_obj)
```

The first argument to `set` must be the name of a MATLAB® variable.

Property Name	Description
`Network`	Neural network object. You must use the Neural Network Toolbox `newff` or `newcf` command to create such an object.

The neural network object property values must satisfy the following requirements to work with System Identification Toolbox™ software:

- `net.numInputs=1`

  The single input can be a vector.

- `net.numLayers`

  Must be a positive integer (`nL`).

- `net.inputConnect=[1;zeros(nL-1,1)]`

  The first layer must be connected to the input.

- `net.outputConnect=[zeros(1,nL-1) 1]`

  The last layer must be connected to the output.

- `net.targetConnect=[zeros(1,nL-1) 1]`

  The last layer is subject to target.

- `net.layerConnect`

  Must be an `nL`-*by*-`nL` logical matrix that satisfies the following conditions:

  - Each layer, except the last one, must have its output connected to another layer `all(any(net.layerConnect(:,1:end-1),1),2) == true`
  - Each layer, except the first one, must have its input connected to another layer `all(any(net.layerConnect(2:end,:),2),1) == true`

  Typical value is `diag(true(1,nL-1),-1)` and represents a series connection from the first layer to the last layer.

- `net.trainFcn`

  Must be set to the MATLAB training function name (M-file, MEX-file, built-in, or P-file).

  `all(any(net.layerConnect(2:end,:),2),1) == true`

- `net.inputs{1}.size`

  Must be set zero (undetermined) or positive integer equal to the number of regressors.

- `net.initFcn='initlay'`

  Indicates the use of layer initialization functions.

- `net.gradientFcn='calcjx'`

- `net.performFcn`

  Must be set to the MATLAB performance function name (M-file, MEX-file, built-in, or P-file).

  ```
  any(exist(net.performFcn)==[2 3 5 6]) == true
  ```

  Typical value is `'mse'`.

- `net.trainParam`

  Must be structure with fields:

  ```
  epochs
  goal
  max_fail
  mem_reduc
  min_grad
  mu, .mu_dec
  mu_inc,.mu_max
  show
  time
  ```

- `net.layers(k).initFcn` for `k=1:nL`

  Must be set to the MATLAB initialization function name (M-file, MEX-file, built-in, or P-file).

  ```
  any(exist(net.layers{k}.initFcn)==[2 3 5 6]) == true
  ```

  Typical value is `'initnw'`.

- `net.biasConnect=logical(net.biasConnect)`

Must be a logical value.

Note The layer numbers, namely the values of `k` in `net.layers{k}`, are labels referring to the different layers. The layers are not necessarily connected in the natural order indicated by `k`. It means that, in principle, `net.layers{1}` is not necessarily the input layer, and `net.layers{end}`is not necessarily the output layer. However, topologically, there is no loss of generality to assume that `net.layers{1}` is the input layer and `net.layers{end}` is the output layer. The previous requirements make these assumptions to simplify how you can validate your network object.

Examples

Use `neuralnet` to specify the neural network nonlinearity estimator in nonlinear ARX and Hammerstein-Wiener models. For example:

```
% Create network object using Neural Network Toolbox
net_obj=newff(zeros(0,10),rand(1,10),[6 8 2]...
                  {'logsig','logsig','purelin'})
% Estimate nonlinear ARX model using
% net_obj as the neural network
m=nlarx(z1,[2,6,10],neuralnet(net_obj));
```

See Also

`nlarx`

`nlhw`

nkshift

Purpose

Shift data sequences

Syntax

```
Datas = nkshift(Data,nk)
```

Description

`Data` contains input-output data in the `iddata` format.

`nk` is a row vector with the same length as the number of input channels in `Data`.

`Datas` is an `iddata` object where the input channels in `Data` have been shifted according to `nk`. A positive value of `nk(ku)` means that input channel number `ku` is delayed `nk(ku)` samples.

`nkshift` supports both frequency- and time-domain data. For frequency-domain data it multiplies with $e^{-ink\omega T}$ to obtain the same effect as shifting in the time domain. For continuous-time frequency-domain data (`Ts = 0`), `nk` should be interpreted as the shift in seconds.

`nkshift` lives in symbiosis with the `InputDelay` property of `idmodel`:

```
m1 = pem(dat,4,'InputDelay',nk)
```

is related to

```
m2 = pem(nkshift(dat,nk),4);
```

such that `m1` and `m2` are the same models, but `m1` stores the delay information for use when frequency responses, etc., are computed.

Note the difference from the `idss` and `idpoly` property `nk`.

```
m3 = pem(dat,4,'nk',nk)
```

gives a model that itself explicitly contains a delay of `nk` samples.

See Also

`Algorithm Properties`

`idss`

Purpose

Estimate nonlinear ARX models

Syntax

```
m=nlarx(data,[na nb nk],Nonlinearity)
m=nlarx(data,[na nb nk],P1,V1,...,PN,VN)
```

Input

`data`

Time-domain `iddata` model object.

`[na nb nk]`

`na` is the number of output terms, `nb` is the number of input terms, and `nk` is the input delays from each input to output.

For `ny` outputs and `nu` inputs, `[na nb nk]` has as many rows as there are outputs. In this case, `na` is an `ny`-by-`ny` matrix whose *i*-*j*th entry gives the number of delayed *j*th outputs used to compute the *i*th output. `nb` and `nk` are `ny`-by-`nu` matrices.

These orders specify the regressors and the predicted output is the following function of these regressors:

$$F\left(y(t-1),\ldots,y(t-na),u(t-nk),\ldots,u(t-nk-nb+1)\right)$$

`Nonlinearity`

Specifies the nonlinearity estimator object as one of the following: `sigmoidnet` (default), `wavenet`, `treepartition`, `customnet`, `neuralnet`, and `linear`.

For `ny` outputs, `Nonlinearity` is an `ny`-by-1 array, such as `[sigmoidnet;wavenet]`. However, if you specify a scalar object, this nonlinearity object applies to all outputs.

For more information about nonlinearity properties, see the corresponding reference pages.

Description

`m=nlarx(data,[na nb nk],Nonlinearity)` constructs and estimates a nonlinear ARX model with orders `[na nb nk]` and `Nonlinearity`. data is the estimation data set.

`m=nlarx(data,[na nb nk],P1,V1,...,PN,VN)` constructs and estimates the model with additional property-value pairs. For more information about model `idnlarx` model properties, see the corresponding reference pages.

Examples

The following commands construct and estimate a nonlinear ARX model:

```
load iddata1
m1=nlarx(z1,[4 2 1],'wave','nlr',[1:3])
```

To perturb the parameters slightly and avoid being trapped in local minima, use the `init` command:

```
m2=init(m1)
```

Estimate the model with perturbed initial parameter values, use the following command:

```
m2=nlarx(z1,m2)
```

See Also

`addreg`

`customreg`

`getreg`

`idnlarx`

`init`

`polyreg`

Purpose

Estimate Hammerstein-Wiener models

Syntax

```
m=nlhw(data,[na nb nk],Nonlinearity)
m=nlhw(data,[na nb nk],P1,V1,...,PN,VN)
M = NLHW(DATA, ORDERS, InputNL, OutputNL, 'InitialState',INIT,
   ..)
```

Arguments

`data`

Time-domain `iddata` model object.

`[nb nf nk]`

Model orders and input delays, where `nb` is the number of zeros plus 1, `nf` is the number of poles, and `nk` is the delay from input to output in terms of the number of samples.

For `nu` inputs and `ny` outputs, `nb`, `nf` and, `nk` are `ny`-by-`nu` matrices whose *i-j*th entry specifies the orders and delay of the transfer function from the *j*th input to the *i*th output.

`InputNL` and `OutputNL`

Specify the input and output nonlinearity estimator objects as one of the following: `pwlinear` (default), `deadzone`, `wavenet`, `saturation`, `customnet`, `sigmoidnet`, and `unitgain`. The nonlinearity estimator objects have properties that you can set in the constructor, as follows:

```
m=nlhw(data,[2 2 1],sigmoidnet('num',5),deadzone([-1,2]))
```

To use default nonlinearity properties, specify the nonlinearity object name as a string. For example:

```
m=nlhw(data,[2 2 1],'sigmoidnet','deadzone')
m=nlhw(data,[2 2 1],'sig','dead') % Abbreviated
```

The estimator `unitgain` (can also be entered as `[]`) means no nonlinearity. Thus, `m=nlhw(data,[2 2 1],'saturation','[]')`. For more information about nonlinearity properties, see the corresponding reference pages.

`InitialState`
: The value of the initial state. `InitialState` can have the following values:

  - `z`: Zero (Default).
  - `e`: Estimated initial state.

Description

`m=nlhw(data,[na nb nk],Nonlinearity)` constructs and estimates a Hammerstein-Wiener model with orders `[na nb nk]` and `Nonlinearity`. `data` is the estimation data set.

`m=nlhw(data,[na nb nk],P1,V1,...,PN,VN)` constructs and estimates the model with additional property-value pairs. For more information about model `idnlhw` model properties, see the corresponding reference pages.

`M = NLHW(DATA, ORDERS, InputNL, OutputNL, 'InitialState',INIT,..)` lets you indicate the status of the initial state.

Examples

The following commands construct and estimate a Hammerstein-Wiener model:

```
load iddata1
m1=nlhw(z1,[4 2 1],'sigmoidnet','deadzone')
```

See Also

`idnlhw`

`init`

`pem`

Purpose

Transform `idmodel` object with noise channels to model with measured channels only

Syntax

```
mod1 = noisecnv(mod)
mod2 = noisecnv(mod,'norm')
```

Description

`mod` is any `idmodel`, `idarx`, `idgrey`, `idpoly`, or `idss`.

The noise input channels in `mod` are converted as follows: Consider a model with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `mod.NoiseVariance` = Λ. The model can also be described with unit variance, normalized noise source v:

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

- `mod1 = noisecnv(mod)` converts the model to a representation of the system [G H] with nu+ny inputs and ny outputs. All inputs are treated as measured, and `mod1` does not have any noise model. The former noise input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- `mod2 = noisecnv(mod,'norm')` converts the model to a representation of the system [G HL] with nu+ny inputs and ny outputs. All inputs are treated as measured, and `mod2` does not have any noise model. The former noise input channels have names `v@yname`, where `yname` is the name of the corresponding output. Note that the noise variance matrix factor L typically is uncertain (has a nonzero covariance). This is taken into account in the uncertainty description of `mod2`.

- If `mod` is a time series, that is, `nu = 0`, `mod1` is a model that describes the transfer function *H* with measured input channels. Analogously, `mod2` describes the transfer function *HL*.

Note the difference with subreferencing:

- `mod('m')` gives a description of G only.
- `mod('n')` gives a description of the noise model characteristics as a time-series model, that is, it describes *H* and also the covariance of *e*. In contrast, `noisecnv(m('n'))` describes just the transfer function *H*. To obtain a description of the normalized transfer function *HL*, use `noisecnv(m('n'),'norm')`.

Converting the noise channels to measured inputs is useful to study the properties of the individual transfer functions from noise to output. It is also useful for transforming `idmodel` objects to representations that do not handle disturbance descriptions explicitly.

Purpose

Set step size for numerical differentiation

Syntax

```
nds = nuderst(pars)
```

Description

The function `pem` uses numerical differentiation with respect to the model parameters when applied to state-space structures. The same is true for many functions that transform model uncertainties to other representations.

The step size used in these numerical derivatives is determined by the M-file `nuderst`. The output argument `nds` is a row vector whose `k`th entry gives the increment to be used when differentiating with respect to the `k`th element of the parameter vector `pars`.

The default version of `nuderst` uses a very simple method. The step size is the maximum of 10^{-4} times the absolute value of the current parameter and 10^{-7}. You can adjust this to the actual value of the corresponding parameter by editing `nuderst`. Note that the nominal value, for example 0, of a parameter might not reflect its normal size.

Purpose

Plot Nyquist curve of frequency response with confidence interval

Syntax

```
nyquist(m)
[fr,w] = nyquist(m)
[fr,w,covfr] = nyquist(m)
nyquist(m1,m2,m3,...,w)
nyquist(m1,'PlotStyle1',m2,'PlotStyle2',...)
nyquist(m1,m2,m3,..'sd*5',sd,'mode',mode)
```

Description

`nyquist` computes the complex-valued frequency response of `idmodel` and `idfrd` models. When invoked without left-hand arguments, `nyquist` produces a Nyquist plot on the screen, that is, a graph of the frequency response's imaginary part against its real part.

The argument `m` is an arbitrary `idmodel` or `idfrd` model. This model can be continuous or discrete, and SISO or MIMO. The `InputNames` and `OuputNames` of the models are used to plot the responses for different I/O channels in separate plots. Pressing the **Enter** key advances the plot from one input-output pair to the next one. You can select specific I/O channels with normal subreferencing: `m(ky,ku)`. With `mode = 'same'`, all plots are given in the same diagram.

`nyquist(m,w)` explicitly specifies the frequency range or frequency points to be used for the plot. To focus on a particular frequency interval `[wmin,wmax]`, set `w = {wmin,wmax}`. To use particular frequency points, set `w` to the vector of desired frequencies. Use `logspace` to generate logarithmically spaced frequency vectors. All frequencies should be specified in rad/s.

`nyquist(m1,m2,...,mN)` or `nyquist(m1,m2,...mN,w)` plots the Bode responses of several `idmodels` or `idfrd` models on a single figure. The models can be mixes of different sizes, and continuous or discrete. The sorting of the plots is based on the `InputNames` and `OutputNames`.

`nyquist(m1,'PlotStyle1',...,mN,'PlotStyleN')` further specifies which color, line style, and/or marker should be used to plot each system, as in

```
nyquist(m1,'r--',m2,'gx')
```

When sd is specified as a number larger than zero, confidence regions are also plotted. These are ellipses in the complex plane and correspond to the region where the true response at the frequency in question is to be found with a confidence corresponding to sd standard deviations (of the Gaussian distribution).

If the argument indicating standard deviations is given as in 'sd+5', a confidence region is plotted for every 5:th frequency, marking the center point by '+'. The default is 'sd+10'.

Note that the frequencies cannot be specified for idfrd objects. For those, the plot and response are calculated for the internally stored frequencies. If the frequencies w are specified when several models are treated, they will apply to all non-idfrd models in the list. If you want different frequencies for different models, you should first convert them to idfrd objects using the idfrd command.

For time-series models (no input channels), the Nyquist plot is not defined.

Input

When nyquist is called with a single system and output arguments,

```
fr = nyquist(m,w) or [fr,w,covfr] = nyquist(m)
```

no plot is drawn on the screen. If m has ny outputs and nu inputs, and w contains Nw frequencies, then fr is an ny-by-nu-by-Nw array such that fr(ky,ku,k) gives the complex-valued frequency response from input ku to output ky at the frequency w(k). For a SISO model, use fr(:) to obtain a vector of the frequency response. The uncertainty information covfr is a 5-D array where covfr(ky,ku,k,:,:)) is the 2-by-2 covariance matrix of the response from input ku to output ky at frequency w(k). The 1,1 element is the variance of the real part, the 2,2 element is the variance of the imaginary part, and the 1,2 and 2,1 elements are the covariance between the real and imaginary parts.

squeeze(covfr(ky,ku,k,:,:)) gives the covariance matrix of the corresponding response.

If `m` is a time series (no input), `fr` is returned as the (power) spectrum of the outputs, an `ny`-by-`ny`-by-`Nw` array. Hence `fr(:,:,k)` is the spectrum matrix at frequency `w(k)`. The element `fr(k1,k2,k)` is the cross spectrum between outputs `k1` and `k2` at frequency `w(k)`. When `k1 = k2`, this is the real-valued power spectrum of output `k1`. The `covfr` is then the covariance of the spectrum `fr`, so that `covfr(k1,k1,k)` is the variance of the power spectrum of output `k1` at frequency `w(k)`. No information about the variance of the cross spectra is normally given. (That is, `covfr(k1,k2,k) = 0` for `k1` not equal to `k2`.)

If the model `m` is not a time series, use `fr = nyquist(m('n'))` to obtain the spectrum information of the noise (output disturbance) signals.

Examples

```
g = spa(data)
m = n4sid(data,3)
nyquist(g,m,3)
```

See Also

`bode`

`etfe`

`ffplot`

`freqresp`

`idfrd`

`spa`

`spafdr`

Purpose

Estimate state-space models using subspace method returning `idss` object

Syntax

```
m = n4sid(data)
m = n4sid(data,order,'Property1',Value1,...,'PropertyN',ValueN)
```

Description

`n4sid` estimates models in state-space form and returns them as an `idss` object `m`. `n4sid` handles an arbitrary number of input and outputs, including the time-series case (no input). The state-space model is in the innovations form

$$x(t+Ts) = Ax(t) + Bu(t) + Ke(t)$$
$$y(t) = Cx(t) + Du(t) + e(t)$$

If `data` is continuous-time (frequency-domain) data, a corresponding continuous-time state-space model is estimated.

`data`: An `iddata` object containing the output-input data. Both time-domain and frequency-domain signals are supported. `data` can also be a `frd` or `idfrd` frequency-response data object.

`order`: The desired order of the state-space model. If `order` is entered as a row vector (as in `order = [1:10]`), preliminary calculations for all the indicated orders are carried out. A plot is then given that shows the relative importance of the dimension of the state vector. More precisely, the singular values of the Hankel matrices of the impulse response for different orders are graphed. You are prompted to select the order, based on this plot. The idea is to choose an order such that the singular values for higher orders are comparatively small. If `order = 'best'`, a model of "best" (default choice) order is computed among the orders 1:10. This is the default choice of `order`.

Estimating the D Matrix

Whether the *D* matrix is estimated or not is governed by the property `nk`, which is further described below. The default is that *D* is not estimated. By setting the `k`th entry of `nk` to 0, the `k`th column of *D* (corresponding to the `k`th input) is estimated. To estimate a full *D* matrix thus, let `nk = zeros(1,nu)` as in

```
m = n4sid(data,order,'nk',[0 .. 0])
```

This holds for both discrete- and continuous-time models.

Property Name/Property Value Pairs

The list of property name/property value pairs can contain any `idss` and algorithm properties. See `idss` and `Algorithm Properties`.

`idss` properties that are of particular interest for `n4sid` are

- `nk`: For time-domain data, this gives delays from the inputs to the outputs, a row vector with the same number of entries as the number of input channels. Default is `nk = [1 1... 1]`. Note that delays of 0 or 1 show up as zeros or estimated parameters in the D matrix. Delays larger than 1 mean that a special structure of the A, B, and C matrices is used to accommodate the delays. This also means that the actual order of the state-space model will be larger than `order`. For continuous-time models estimated from continuous-time (frequency-domain) data, the elements of `nk` are restricted to the values 1 and 0.
- `CovarianceMatrix` (can be abbreviated to `'co'`): Setting `CovarianceMatrix` to `'None'` blocks all calculations of uncertainty measures. These can take the major part of the computation time. Note that, for a `'Free'` parameterization, the individual matrix elements cannot be associated with any variance. (These parameters are not identifiable.) Instead, the resulting model `m` stores a hidden state-space model in canonical form that contains covariance information. This is used when the uncertainty of various input-output properties is calculated. It can also be retrieved by `m.ss = 'can'`. The actual covariance properties of `n4sid` estimates are not known today. Instead the Cramer-Rao bound is computed and stored as an indication of the uncertainty.
- `DisturbanceModel`: Setting `DisturbanceModel` to `'None'` will deliver a model with `K = 0`. This has no direct effect on the dynamics model other than that the default choice of `N4Horizon` will not involve past outputs.

- `InitialState`: The initial state is always estimated for better accuracy. However, it is returned with `m` only if `InitialState = 'Estimate'`.

Algorithm properties that are of special interest are

- `Focus`: Assumes the values `'Prediction'` (default), `'Simulation'`, `'Stability'`, passbands, or any SISO linear filter (given as an LTI or `idmodel` object, or as filter coefficients. See `Algorithm Properties`.) Setting `'Focus'` to `'Simulation'` chooses weights that should give a better simulation performance for the model. In particular, a stable model is guaranteed. Selecting a linear filter focuses the fit to the frequency ranges that are emphasized by this filter.
- `N4Weight`: This property determines some weighting matrices used in the singular-value decomposition that is a central step in the algorithm. Two choices are offered: `'MOESP'`, corresponding to the MOESP algorithm by Verhaegen, and `'CVA'`, which is the canonical variable algorithm by Larimore. The default value is `'N4Weight' = 'Auto'`, which gives an automatic choice between the two options. `m.EstimationInfo.N4Weight` tells you what the actual choice turned out to be.
- `N4Horizon`: Determines the prediction horizons forward and backward used by the algorithm. This is a row vector with three elements: `N4Horizon = [r sy su]`, where `r` is the maximum forward prediction horizon. That is, the algorithm uses up to `r` step-ahead predictors. `sy` is the number of past outputs, and `su` is the number of past inputs that are used for the predictions. See pages 209 and 210 in Ljung (1999) for the exact meaning of this. These numbers can have a substantial influence on the quality of the resulting model, and there are no simple rules for choosing them. Making `'N4Horizon'` a `k`-by-3 matrix means that each row of `'N4Horizon'` is tried, and the value that gives the best (prediction) fit to data is selected. (This option cannot be combined with selection of model order.) If the property `'Trace'` is `'On'`, information about the results is given in the MATLAB® Command Window.

If you specify only one column in `'N4Horizon'`, the interpretation is `r=sy=su`. The default choice is `'N4Horizon' = 'Auto'`, which uses an Akaike Information Criterion (AIC) for the selection of `sy` and `su`. If `'DisturbanceModel' = 'None'`, `sy` is set to 0. Typing `m.EstimationInfor.N4Horizon` will tell you what the final choices of horizons were.

Algorithm

The algorithm is described in Section 10.6 in Ljung (1999).

Examples

Build a fifth-order model from data with three inputs and two outputs. Try several choices of auxiliary orders. Look at the frequency response of the model.

```
z = iddata([y1 y2],[ u1 u2 u3]);
m = n4sid(z,5,'n4h',[7:15]','trace','on');
bode(m,'sd',3)
```

Estimate a continuous-time model, in a canonical form parameterization, focusing on the simulation behavior. Determine the order yourself after seeing the plot of singular values.

```
m = n4sid(m,[1:10],'foc','sim','ssp','can','ts',0)
bode(m)
```

Learn More

For definition of state-space models and how to estimate them from input-output data, see “Identifying State-Space Models”.

For more information about estimating state-space models from time-series data, see “Estimating State-Space Time-Series Models”.

Other references:

vanOverschee, P., and B. DeMoor, *Subspace Identification of Linear Systems: Theory, Implementation, Applications*, Kluwer Academic Publishers, 1996.

Verhaegen, M., “Identification of the deterministic part of MIMO state space models,” *Automatica,* Vol. 30, pp. 61-74, 1994.

Larimore, W.E., “Canonical variate analysis in identification, filtering and adaptive control,” In *Proc. 29th IEEE Conference on Decision and Control*, pp. 596-604, Honolulu, 1990.

See Also

`Algorithm Properties`

`idss`

`pem`

Purpose

Estimate parameters of output-error model returning `idpoly` object

Syntax

```
m = oe(data,orders)
m = oe(data,'nb',nb,'nf',nf,'nk',nk)
m = oe(data,orders,'Property1',Value1,'Property2',Value2,...)
```

Description

`oe` returns `m` as an `idpoly` object with the resulting parameter estimates, together with estimated covariances. The parameters of the output-error model structure

$$y(t) = \frac{B(q)}{F(q)}u(t-nk) + e(t)$$

are estimated using a prediction error method.

`data` is an `iddata` object containing the output-input data. Both time- and frequency-domain data are supported. Moreover, `data` can be an `frd` or `idfrd` frequency-response data object.

The structure information can either be given explicitly as

```
(...,'nb',nb,'nf',nf,'nk',nk,...)
```

or in the argument `orders`, given as

```
orders = [nb nf nk]
```

The parameters `nb` and `nf` are the orders of the output-error model and `nk` is the delay. Specifically,

$$nb: \quad B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1}$$

$$nf: \quad F(q) = 1 + f_1 q^{-1} + \ldots + f_{nf} q^{-nf}$$

Alternatively, you can specify the vector as

```
orders = mi
```

where mi is an initial guess at the output-error model given in idpoly format. See “What Are Black-Box Polynomial Models?”.

For multiple-input systems, nb, nf, and nk are row vectors with as many entries as there are input channels. Entry number i then describes the orders and delays associated with the ith input.

Continuous-Time Models

If data is continuous-time (frequency-domain) data, oe estimates a continuous-time model with transfer function

$$G(s) = \frac{B(s)}{F(s)} = \frac{b_{nb}s^{(nb-1)} + b_{nb-1}s^{(nb-2)} + \ldots + b_1}{s^{nf} + f_{nf}s^{(nf-1)} + \ldots + f_1}$$

The orders of the numerator and denominator are thus determined by nb and nf just as in the discrete-time case. However, the delay nk has no meaning and should be omitted. For multiple-input systems, nb and nf are row vectors with obvious interpretation.

Properties

The structure and the estimation algorithm are affected by any property name/property value pairs that are set in the input argument list. Useful properties are 'Focus', 'InitialState', 'InputDelay', 'SearchMethod', 'MaxIter', 'Tolerance', 'LimitError', 'FixedParameter', and 'Trace'.

See Algorithm Properties, idpoly, and idmodel for details of these properties and their possible values.

oe does not support multiple-output models. Use a state-space model for this case (see n4sid and pem).

Algorithm

oe uses essentially the same algorithm as armax, with modifications to the computation of prediction errors and gradients.

Examples

Suppose fast sampled data (`Ts = 0.001`) is available from a plant with a bandwidth of about 500 rad/s. The data is treated as continuous-time frequency-domain data, and a model of the type

$$G(s) = \frac{b}{s^3 + f_1 s^2 + f_2 s + f_3}$$

is estimated.

```
z = iddata(y,u,0.001);
zf = fft(z);
zf.ts = 0;
m = oe(zf,[1 3],'foc',[0 500])
```

See Also

`Algorithm Properties`

`EstimationInfo`

`idmodel`

`idpoly`

`pem`

`n4sid`

Purpose

Construct operating point specification object for `idnlarx` model

Syntax

```
SPEC = operspec(NLSYS)
```

Description

`SPEC = operspec(NLSYS)` creates an operating point specification object for the `idnlarx` model `NLSYS`. The object encapsulates constraints on input and output signal values. These specifications are used to determine an operating point of the `idnlarx` model using `findop(idnlarx)`.

Input

- `NLSYS`: `idnlarx` model.

Input

- `SPEC`: Operating point specification object. `SPEC` contains the following properties:
  - `Input`: Structure with fields:
    - `Value`: Initial guess for the values of the input signals. Specify a vector of length equal to number of model inputs. Default value: Vector of zeros.
    - `Min`: Minimum value constraint on values of input signals for the model. Default: `-Inf` for all channels.
    - `Max`: Maximum value constraint on values of input signals for the model. Default: `Inf` for all channels.
    - `Known`: Specifies when `Value` is known (fixed) or is an initial guess. Use a logical vector to denote which signals are known (logical `1`, or `true`) and which have to be estimated using `findop` (logical `0`, or `false`). Default value: `true`.
  - `Output`: Structure with fields:
    - `Value`: Initial guess for the values of the output signals. Default value: Vector of zeros.
    - `Min`: Minimum value constraint on values of output signals for the model. Default value: `-Inf`.

- `Max`: Maximum value constraint on values of output signals for the model. Default value: `-Inf`.

See Also

`findop(idnlarx)`

Purpose

Construct operating point specification object for `idnlhw` model

Syntax

```
SPEC = operspec(NLSYS)
```

Description

`SPEC = operspec(NLSYS)` creates an operating point specification object for the `idnlhw` model `NLSYS`. The object encapsulates constraints on input and output signal values. These specifications are used to determine an operating point of the `idnlhw` model using `findop(idnlhw)`.

Input

- `NLSYS`: `idnlhw` model.

Input

- `SPEC`: Operating point specification object. `SPEC` contains the following fields:
  - `Value`: Initial guess for the values of the input signals. Specify a vector of length equal to number of model inputs. Default value: Vector of zeros.
  - `Min`: Minimum value constraint on values of input signals for the model. Default: `-Inf` for all channels.
  - `Max`: Maximum value constraint on values of input signals for the model. Default: `Inf` for all channels.
  - `Known`: Specifies when `Value` is known (fixed) or is an initial guess. Use a logical vector to denote which signals are known (logical `1`, or `true`) and which have to be estimated using `findop` (logical `0`, or `false`). Default value: `true`.

Note

1. If the input is completely known (`'Known'` field is set to `true` for all input channels), then the initial state values are determined using input values only. In this case, `findop(idnlhw)` ignores the output signal specifications.
2. If the input values are not completely known,`findop(idnlhw)` uses the output signal specifications to achieve the following objectives:
   - Match target values of known output signals (output channels with `Known = true`).
   - Keep the free output signals (output channels with `Known = false`) within the specified min/max bounds.

See Also

`findop(idnlhw)`

Purpose

Prediction errors associated with model and data set

Syntax

```
e = pe(m,data)
[e,x0] = pe(m,data,init)
```

Description

`data` is the output-input data set, given as an `iddata` object, and `m` is any `idmodel` or `idnlmodel` object. Both time-domain and frequency-domain data are supported, and `data` can also be an `idfrd` object.

`e` is returned as an `iddata` object, so that `e.OutputData` contains the prediction errors that result when model `m` is applied to the data.

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

The argument `init` determines how to deal with the initial conditions:

- `init = 'e(stimate)'` means that the initial state is chosen so that the norm of prediction error is minimized. This initial state is returned as `x0`.
- `init = 'd(elayexpand)'`: Same as `'estimate'`, but for a model with nonzero `InputDelay`, the delays are first converted to explicit model delays (using `inpd2nk`) so that they are contained in x0.
- `init = 'z(ero)'` sets the initial state to zero.
- `init = 'm(odel)'` uses the model's internally stored initial state.
- `init = x0i`, where `x0i` is a column vector of appropriate dimension, uses that value as initial state. For multiexperiment data, `x0i` may be a matrix whose columns give different initial states for each experiment. For a continuous-time model `m`, `x0` is the initial state for this model. Any modifications of the initial state that sampling might require are automatically handled. If `m` has a non-zero `InputDelay`, and you need to access the values of the inputs during this delay, you must first apply `inpd2nk(m)`. If `m` is continuous in time, it must first be sampled before `inpd2nk` can be applied.

If `init` is not specified, the model property `m.InitialState` is used, so that `'Estimate'`, `'Backcast'`, and `'Auto'` set `init = 'Estimate'`, while `m.InitialState = 'Zero'` sets `init = 'zero'`, and `'Fixed'` and `'Model'` set `init = 'model'`.

The output argument `x0` is the value of the initial state used. If `data` contains several experiments, `x0` is a matrix containing the initial states from each experiment.

See Also

`compare`

`predict`

`resid`

`sim`

`simsd`

Purpose

Estimate model parameters using iterative prediction-error minimization method

Syntax

```
m = pem(data)
m = pem(data,mi)
m = pem(data,mi,'Property1',Value1,...,'PropertyN',ValueN)
m = pem(data,orders)
m = pem(data,'P1D')
m = pem(data,'nx',ssorder)
m = pem(data,'na',na,'nb',nb,'nc',nc,'nd',nd,'nf',nf,'nk',nk)
m = pem(data,orders,'Property1',Value1,...,'PropertyN',ValueN)
```

How to Use

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

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

Description

`pem` is the basic estimation command in the toolbox and covers a variety of situations.

`data` is always an `iddata` object that contains the input/output data. Both time-domain and frequency-domain signals are supported. `data` can also be an `frd` or `idfrd` frequency-response data object. Estimation of noise models (`K` in state-space models and `A`, `C`, and `D` in polynomial models) is not supported for frequency-domain data.

With Initial Model

`mi` is any `idmodel` or `idnlmodel` object. It could be a result of another estimation routine, or constructed and modified by the constructors (`idarx`, `idpoly`, `idss`, `idgrey`, `idproc`) and `set`. The properties of `mi`

can also be changed by any property name/property value pairs in `pem` as indicated in the syntax.

`m` is then returned as the best fitting model in the model structure defined by `mi`. The iterative search is initialized at the parameters of the initial/nominal model `mi`. `m` will be of the same class as `mi`.

Black-Box State-Space Models

With `m = pem(data,n)`, where `n` is a positive integer, or `m = pem(data,'nx',n)`, a state-space model of order `n` is estimated.

$$x(t+Ts) = Ax(t) + Bu(t) + Ke(t)$$
$$y(t) = Cx(t) + Du(t) + e(t)$$

If `data` is continuous-time (frequency-domain) data, a corresponding continuous-time state space model is estimated.

The default is that it is estimated in a `'Free'` parameterization that can be further modified by the properties `'nk'`, `'DisturbanceModel'`, and `'InitialState'` (see the corresponding reference pages for `idss` and `n4sid`). The model is initialized by `n4sid` and then further adjusted by optimizing the prediction error fit.

You can choose among several different orders by

```
m = pem(data,'nx',[n1,n2,...nN])
```

and you are then prompted for the “best” order. By

```
m = pem(data,'best')
```

an automatic choice of order among 1:10 is made.

```
m = pem(data)
```

is short for `m = pem(data,'best')`. To work with other delays, use, for example, `m = pem(data,'best','nk',[0,...0])`.

In this case `m` is returned as an `idss` model.

Estimating the D, K, and X0 Matrices

Whether the D matrix is estimated or not is governed by the property `nk`, which is further described below. The default is that D is not estimated. By setting the `k`th entry of `nk` to `0`, the `k`th column of D (corresponding to the `k`th input) is estimated. To estimate a full D matrix, let `nk = zeros(1,nu)`, as in

```
m = pem(data,order,'nk',[0 .. 0])
```

This holds for both discrete- and continuous-time models.

For frequency-domain data, K is always fixed to `0`. For time-domain data, K is estimated by default. To fix K to `0` in this case, use

```
m = pem(data,order,'DisturbanceModel','none')
```

Similarily, `X0` is estimated if `'InitialState'` is set to `'Estimate'`, and fixed to `0` if `'InitialState'` is set to `'Zero'`.

Black-Box Multiple-Input-Single-Output Models

The function `pem` also handles the general multiple-input-single-output structure

$$A(q)y(t) = \frac{B_1(q)}{F_1(q)}u_1(t-nk_1) + \ldots + \frac{B_{nu}(q)}{F_{nu}(q)}u_{nu}(t-nk_{nu}) + \frac{C(q)}{D(q)}e(t)$$

The orders of this general model are given either as

```
orders = [na nb nc nd nf nk]
```

or with `(...'na',na,'nb',nb,...)` as shown in the syntax. Here `na`, `nb`, `nc`, `nd`, and `nf` are the orders of the model, and `nk` is the delay(s). For multiple-input systems, `nb`, `nf`, and `nk` are row vectors giving the orders and delays of each input. (See“What Are Black-Box Polynomial Models?” for exact definitions of the orders.) When the orders are specified with separate entries, those not given are taken as zero.

For frequency-domain data, only estimation of `B` and `F` is supported. It is simpler to use `oe` in that case.

In this case `m` is returned as an `idpoly` object.

Continuous-Time Process Models

Entering for the initial model an acronym for a process model, as in

```
m = pem(data,'P2UI')
```

will estimate a continuous-time process model of the indicated type. See the reference page for `idproc` for details of possible model types and associated property name/property value pairs.

In this case `m` is returned as an `idproc` model.

Properties

In all cases the algorithm is affected by the properties (see `Algorithm Properties` for details):

- `Focus`, with possible values `'Prediction'` (default), `'Simulation'`, or a passband range.
- `MaxIter` and `Tolerance` govern the stopping criteria for the iterative search.
- `LimitError` deals with how the criterion can be made less sensitive to outliers and bad data.
- `MaxSize` determines the largest matrix ever formed by the algorithm. The algorithm goes into `for` loops to avoid larger matrices, which can be more efficient than using virtual memory.
- `Trace`, with possible values `'Off'`, `'On'`, and `'Full'`, governs the information sent to the MATLAB Command Window.

For black-box state-space models, `'N4Weight'` and `'N4Horizon'` will also affect the result, since these models are initialized with an `n4sid` estimate. See the reference page for `n4sid`.

Typical `idmodel` properties are (see `idmodel` properties for more details)

- `Ts` is the sampling interval. Set `'Ts'= 0` to obtain a continuous-time state-space model. For discrete-time models, `'Ts'` is automatically set to the sampling interval of the data. Note that, in the black-box case, it is usually better to first estimate a discrete-time model, and then convert that to continuous time using `d2c`.
- `nk` is the time delays from the inputs (not applicable to structured state-space models). Time delays specified by `'nk'` will be included in the model.
- `DisturbanceModel` determines the parameterization of `K` for free and canonical state-space parameterizations, as well as for `idgrey` models. It also determines whether a noise model should be included for `idproc` models.
- `InitialState`: The initial state can have a substantial influence on the estimation result for systems with slow responses. It is most pronounced for output-error models (`K = 0` for state-space and `na = nc = nd =0` for input/output models). The default value `'Auto'"` estimates the influence of the initial state and sets the value to `'Estimate'`, `'Backcast'`, or `'Zero'` based on this effect. Possible values of `'InitialState'` are `'Auto'`, `'Estimate'`, `'Backcast'`, `'Zero'`, and `'Fixed'`.

Examples

Here is an example of a system with three inputs and two outputs. A canonical form state-space model of order 5 is sought.

```
z = iddata([y1 y2],[ u1 u2 u3]);
m = pem(z,5,'ss','can')
```

Building an ARMAX model for the response to output 2,

```
ma = pem(z(:,2,:),'na',2,'nb',[2 3 1],'nc',2,'nk',[1 2 0])
```

Comparing the models (`compare` automatically matches the channels using the channel names),

```
compare(z,m,ma)
```

Algorithm

`pem` uses essentially the same algorithm as `armax`, with modifications to the computation of prediction errors and gradients.

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

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

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

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

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

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

See Also

`Algorithm Properties`

`EstimationInfo`

`armax`

`bj`

`oe`

Purpose

Level of excitation of input signals

Syntax

```
Ped = pexcit(Data)
[Ped.Maxnr] = pexcit(Data,Maxnr,Threshold)
```

Description

`Data` is an `iddata` object with time- or frequency-domain signals.

`Ped` is the degree or order of excitation of the inputs in `Data`. A row vector of integers with as many components as there are inputs in `Data`. The intuitive interpretation of the degree of excitation in an input is the order of a model that the input is capable of estimating in an unambiguous way.

`Maxnr` is the maximum order tested. Default is `min(N/3,50)`, where `N` is the number of input data.

`Threshold` is the threshold level used to measure which singular values are significant. Default is `1e-9`.

References

Section 13.2 in Ljung (1999).

See Also

`advice`

`iddata`

plot

Purpose

Plot iddata or model objects

Syntax

```
plot(data)
plot(d1,...,dN)
plot(d1,PlotStyle1,...,dN,PlotStyleN)
plot(model)
```

Description

data is the output-input data to be graphed, given as an iddata object. A split plot is obtained with the outputs on top and the inputs at the bottom.

One plot for each I/O channel combination is produced. Pressing the **Enter** key advances the plot. Typing **Ctrl+C** aborts the plotting in an orderly fashion.

To plot a specific interval, use plot(data(200:300)). To plot specific input/output channels, use plot(data(:,ky,ku)), consistent with the subreferencing of iddata objects.

If data.intersample = 'zoh', the input is piecewise constant between sampling points, and it is then graphed accordingly.

To plot severaliddata sets d1,...,dN, use plot(d1,...,dN). I/O channels with the same experiment name, input name, and output name are always plotted in the same plot.

With PlotStyle, the color, line style, and marker of each data set can be specified

```
plot(d1,'y:*',d2,'b')
```

just as in the regular plot command.

model is an idmodel, idnlarx, or idnlhw object.

See Also

iddata

Purpose

Polynomial model parameters from single-input and single-output `idpoly` object

Syntax

```
[A,B,C,D,F] = polydata(m)
[A,B,C,D,F,dA,dB,dC,dD,dF] = polydata(m)
```

Description

This is essentially the inverse of the `idpoly` constructor. It returns the polynomials of the general model

$$A(q)y(t) = \frac{B_1(q)}{F_1(q)}u_1(t-nk_1) + \ldots + \frac{B_{nu}(q)}{F_{nu}(q)}u_{nu}(t-nk_{nu}) + \frac{C(q)}{D(q)}e(t)$$

as contained in the model `m`.

`dA`, `dB`, etc. are the standard deviations of `A`, `B`, etc.

`m` can be any single-output `idmodel`, that is, not just `idpoly`. For multiple-output models you can use `[A,B,C,D,F] = polydata(m(ky,:))` to obtain the polynomials for the `ky`th output.

See Also

`idmodel`

`idpoly`

`tfdata`

polyreg

Purpose

Generate custom regressors by computing powers and products of standard regressors

Syntax

```
polyreg(model,'MaxPower',n)
polyreg(model,'MaxPower',n,'CrossTerm','on')
polyreg(model,'MaxPower',n,'CrossTerm','off')
```

Input

`model`
: Name of the `idnlarx` model object.

`MaxPower—n`
: Property-value pair specifies the maximum power of the multivariable polynomial in terms of standard regressors of the model.

  Default: `2`.

`CrossTerms—'on' or 'off'`
: Property-value pair specifies whether to include or exclude cross-terms of the polynomial, or products of standard regressors.

  Default: `'off'`.

Description

`polyreg` is a method of the `idnlarx` model object.

`polyreg(model,'MaxPower',n)` adds one or more custom regressors to nonlinear ARX model `model`. Custom regressors are powers of custom regressors up to the maximum power `n`, but excluding terms of power `1`.

`polyreg(model,'MaxPower',n,'CrossTerm','on')` one or more custom regressors to nonlinear ARX model `model` and includes cross-terms of the polynomial (products of standards regressors).

`polyreg(model,'MaxPower',n,'CrossTerm','off')` one or more custom regressors to nonlinear ARX model `model` and excludes cross-terms of the polynomial (products of standards regressors).

Examples

```
% Construct a nonlinear ARX model that is
% linear in the regressors.
```

```
M=idnlarx([2 2 1],'linear')
% Define custom regressors with default settings,
% which include second-order polynomial of standard
% regressors and no cross-terms.
P=polyreg(M)
% Estiomate model using custom regressors
% in the nonlinear block.
M=pem(Data,M,'customreg',P)
```

See Also

customreg

getreg

idnlarx

polyreg

poly1d

Purpose

Class representing single-variable polynomial nonlinear estimator for Hammerstein-Wiener models

Syntax

```
t=poly1d('Degree',n)
t=poly1d('Coefficients',C)
t=poly1d(n)
```

Description

`poly1d` is an object that stores the single-variable polynomial nonlinear estimator for Hammerstein-Wiener models.

You can use the constructor to create the nonlinearity object, as follows:

`t=poly1d('Degree',n)` creates a polynomial nonlinearity estimator object of `n`th degree.

`t=poly1d('Coefficients',C)` creates a polynomial nonlinearity estimator object with coefficients `C`.

`t=poly1d(n)` a polynomial nonlinearity estimator object of `n`th degree.

Use `evaluate(p,x)` to compute the value of the function defined by the `poly1d` object `p` at `x`.

Remarks

Use `poly1d` to define a nonlinear function $y = F(x)$, where F is a single-variable polynomial function of x:

$$F(x) = c(1)x^n + c(2)x^{(n-1)} + \ldots + c(n)x + c(n+1)$$

poly1d Properties

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% List all property values
get(p)
% Get value of Coefficients property
p.Coefficients
```

Property Name	Description
`Degree`	Positive integer specifies the degree of the polynomial Default=1. For example: `poly1d('Degree',3)`
`Coefficients`	1-*by*-`(n+1)` matrix containing the polynomial coefficients.

Examples

Use `poly1s` to specify the single-variable polynomial nonlinearity estimator in Hammerstein-Wiener models. For example:

```
m=nlhw(Data,Orders,poly1d('deg',3),[]);
```

where `'deg'` is an abbreviation for the for the property `'Degree'`.

See Also

`nlhw`

Purpose

Predict output k steps ahead

Syntax

```
yp = predict(m,data)
[yp,x0p,mpred] = predict(m,data,k,'InitialState',init)
```

Description

`data` is the output-input data as an `iddata` object, and `m` is any `idmodel` or `idnlmodel` object. `predict` is meaningful only for time-domain data.

The argument `k` indicates that the k step-ahead prediction of `y` according to the model `m` is computed. In the calculation of $yp(t)$, the model can use outputs up to time

$$t-k: y(s), s = t-k, t-k-1, \ldots$$

and inputs up to the current time t. The default value of `k` is `1`.

The output `yp` is an `iddata` object containing the predicted values as `OutputData`.

`x0p` is the used (estimated) initial state vector. For multiexperiment data, `x0p` is a matrix, whose columns contain the initial states for each experiment.

The output argument `mpred` contains the k step-ahead predictor. This is given as a cell array, whose kth entry is an `idpoly` model for the predictor of output number k. Note that these predictor models have as input both input and output signals in the data set. The channel names indicate how the predictor model and the data fit together.

`init` determines how to deal with the initial state:

- `init ='e(stimate)'`: The initial state is set to a value that minimizes the norm of the prediction error associated with the model and the data.
- `init = 'd(elayexpand)'`: Same as `'estimate'`, but for a model with nonzero `InputDelay`, the delays are first converted to explicit model delays (using `inpd2nk`) so that they are contained in `x0p`.
- `init = 'z(ero)'` sets the initial state to zero.

- `init = 'm(odel)'` uses the model's internally stored initial state.
- `init = x0`, where `x0` is a column vector of appropriate dimension, uses that value as initial state. For multiexperiment data, `x0` can be a matrix whose columns give different initial states for each experiment. For a continuous-time model `m`, `x0` is the initial state for this model. Any modifications of the initial state that sampling might require are automatically handled. If `m` has a non-zero `InputDelay`, and you need to access the values of the inputs during this delay, you must first apply `inpd2nk(m)`. When `m` is a continuous-time model, it must first be sampled before `inpd2nk` can be applied.

If `init` is not specified, the model property `m.InitialState` is used, so that `'Estimate'`, `'Backcast'`, and `'Auto'` set `init = 'Estimate'`, while `m.InitialState = 'Zero'` sets `init = 'zero'`, and `'Model'` and `'Fixed'` set `init = 'model'`.

An important use of `predict` is to evaluate a model's properties in the mid-frequency range. Simulation with `sim` (which conceptually corresponds to `k = inf`) can lead to levels that drift apart, since the low-frequency behavior is emphasized. One step-ahead prediction is not a powerful test of the model's properties, since the high-frequency behavior is stressed. The trivial predictor $\hat{y}(t) = y(t-1)$ can give good predictions in case the sampling of the data is fast.

Another important use of `predict` is to evaluate time-series models. The natural way of studying a time-series model's ability to reproduce observations is to compare its k step-ahead predictions with actual data.

Note that for output-error models, there is no difference between the k step-ahead predictions and the simulated output, since, by definition, output-error models only use past inputs to predict future outputs.

Algorithm

The model is evaluated in state-space form, and the state equations are simulated k steps ahead with initial value $x(t-k) = \hat{x}(t-k)$, where $\hat{x}(t-k)$ is the Kalman filter state estimate.

Examples

Simulate a time series, estimate a model based on the first half of the data, and evaluate the four step-ahead predictions on the second half.

```
m0 = idpoly([1 -0.99],[],[1 -1 0.2]);
e = iddata([],randn(400,1));
y = sim(m0,e);
m = armax(y(1:200),[1 2]);
yp = predict(m,y,4);
plot(y(201:400),yp(201:400))
```

Note that the last two commands are also achieved by

```
compare(y,m,4,201:400);
```

See Also

compare

pe

sim

simsd

Purpose

Predict output *k* steps ahead for nonlinear ARX model

Syntax

```
YP = predict(MODEL,DATA,K)
YP = predict(SYS,DATA,K,INIT)
YP = predict(MODEL,DATA,K,'InitialState',INIT)
```

Description

`YP = predict(MODEL,DATA,K)` predicts the *k*-step ahead output with an `idnlarx` model.

`YP = predict(SYS,DATA,K,INIT)` or `YP = predict(MODEL,DATA,K,'InitialState',INIT)` specifies the initialization.

Input

- `MODEL`: `idnlarx` model object.
- `DATA`: `iddata` object.
- `K`: Prediction horizon. Old outputs up to time *t-K* are used to predict the output at time t. All relevant inputs are used. Default value: `K = 1`).
- `INIT`: initialization specification. `INIT` can be the following:
  - `'e'`: Assume the initial states of the model are such that the first `N` values of the predicted output match the first `N` samples of the measured output exactly, where `N` is the maximum channel delay in the model (`N = max(getDelayInfo(model))`). The initial states are not computed explicitly, but are assumed to exist. The prediction starts at the $(N+1)^{th}$ sample, while a perfect match is assumed for the first `N` samples. If you want prediction of response values starting from the first data sample, you must estimate and provide the initial state vector explicitly as described in the following option for `INIT = X0`.
  - Real column vector `X0`, for the state vector corresponding to an appropriate number of output and input data samples prior to the simulation start time. To build an initial state vector from a given set of input-output data or to generate equilibrium states, use `data2state(idnlarx)`, `findstates(idnlarx)` or

`findop(idnlarx)`. For multi-experiment data, `X0` may be a matrix whose columns give different initial states for different experiments.

- `'z'`: (Default) Zero initial state, equivalent to a zero vector of appropriate size.
- `iddata` object containing output and input data samples prior to the simulation start time. If it contains more data samples than necessary, only the last samples are taken into account. This syntax is equivalent to `sim(MODEL, U, 'InitialState', data2state(MODEL,INIT))` where `data2state(idnlarx)` transforms the `iddata` object `INIT` to a state vector.

Input

`YP`: Predicted output as an `iddata` object. If `DATA` contains multiple experiments, so will `YP`.

Note If `predict` is called without an output argument, MATLAB® software displays the predicted output(s) in a plot window.

See Also

`sim(idnlarx)`

`findop(idnlarx)`

`data2state(idnlarx)`

`findstates(idnlarx)`

Purpose

Predict output k steps ahead for nonlinear ODE model

Syntax

```
YP  = predict(NLSYS,DATA);
[YP,X0,XFINAL] = predict(NLSYS,DATA);
[YP,X0,XFINAL] = predict(NLSYS,DATA,K);
[YP,X0,XFINAL] = predict(NLSYS,DATA,K,XOINIT);
```

Description

`YP = predict(NLSYS,DATA);` predicts the k-step ahead output with an `idnlgrey` model.

`[YP,X0,XFINAL] = predict(NLSYS,DATA);` returns the initial states used in the prediction as well as the final states computed, in addition to the predicted output.

`[YP,X0,XFINAL] = predict(NLSYS,DATA,K);` specifies the prediction horizon to use during prediction.

`[YP,X0,XFINAL] = predict(NLSYS,DATA,K,XOINIT);` specifies the initialization for the k-step ahead prediction.

Input

- `NLSYS`: `idnlgrey` model for which output is to be predicted.
- `DATA`: Input-output data `[Y U]`. `U` is the input data that can be given either as an `iddata` object or as a matrix `U = [U1 U2 ... Um]`, where the `k`:th column vector is input U_k. Similarly, `Y` is either an `iddata` object or a matrix of outputs (with as many columns as there are outputs). For time-continuous `idnlgrey` objects, `DATA` passed as a matrix will lead to that the data sample interval, `Ts`, is set to one.
- `K`: Prediction horizon. `K` and can be set to an integer between 1 and `inf` (pure simulation). As `idnlgrey` assumes an output error model structure, where prediction and simulation coincide, `K` has no meaning.
- `XOINIT`: Initial state to use. It can take the following values:
  - `'zero'` : Zero initial state $x(0)$ with all states fixed (`nlsys.InitialStates.Fixed` is thus ignored).

- `'fixed'`: (or `NLSYS.InitialState`) Determines the values of the 'model' initial states, but all states are fixed (`NLSYS.InitialStates.Fixed` is thus ignored). This is the default value `XOINIT`.
- vector/matrix: Column vector of appropriate length is used as initial state. For multiple experiment `DATA`, $x(0)$ may be a matrix whose columns give different initial states for each experiment. All initial states are kept fixed (`nlsys.InitialStates.Fixed` is thus ignored).
- struct array : An `Nx`-by-1 structure array with fields:
  - `Name`: Name of the state (a string).
  - `Unit`: Unit of the state (a string).
  - `Value`: Value of the states (a finite real 1-by-`Ne` vector, where `Ne` is the number of experiments.)
  - `Minimum`: Minimum values of the states (a real 1-by-`Ne` vector or a real scalar, in which case all initial states have the same minimum value).
  - `Maximum`: Maximum values of the states (a real 1-by-`Ne` vector or a real scalar, in which case all initial states have the same maximum value).
  - `Fixed`: `true` (or a `true` 1-by-Ne vector). Any `false` value will be ignored. The initial state can also be specified through the property-value pair with name `'InitialState'` or `'X0'` followed by the value `XOINIT`.

Input

- `YP`: Predicted output. If `DATA` is an `iddata` object, then `YP` will also be an `iddata` object. Otherwise, `YP` will be a matrix where the `k`:th output is found in the `k`:th column of `YP`. If `DATA` is a multiple experiment `iddata` object, then `YP` will be as well.
- `X0`: Initial states. In the single experiment case it is a column vector of length `Nx`. For multi-experiment data, `X0` is an `Nx`-by-`Ne` matrix with the `i`:th column specifying the initial state of experiment `i`.

- `XFINAL`: Final states computed. In the single experiment case it is a column vector of length `Nx`. For multi-experiment data, `XFINAL` is an `Nx`-by-`Ne` matrix with the `i`:th column specifying the initial state of experiment `i`.

Note If `predict` is called without an output argument, MATLAB® software displays the predicted output(s) in a plot window.

See Also

`sim(idnlgrey)`

`pe`

`pem`

predict(idnlhw)

Purpose

Predict output k steps ahead for Hammerstein-Wiener model

Syntax

```
YP = predict(MODEL,DATA,K)
YP = predict(SYS,DATA,K,INIT)
YP = predict(MODEL,DATA,K,'InitialState',INIT)
```

Description

`YP = predict(MODEL,DATA,K)` predicts the k-step ahead output with an `idnlhw` model.

`YP = predict(SYS,DATA,K,INIT)` or `YP = predict(MODEL,DATA,K,'InitialState',INIT)` specifies the initialization.

Input

- `MODEL`: `idnlhw` model object.
- `DATA`: `iddata` object.
- `K`: Prediction horizon. Old outputs up to time t-`K` are used to predict the output at time t. All relevant inputs are used. Default value: `K` = 1).
- `INIT`: Initialization specification. `INIT` can be the following:
  - `'e'`: Estimate initial states minimizing the sum of squared prediction errors. To compute the initial state estimates explicitly, use `findstates(idnlhw)`.
  - Real column vector `X0` for the initial state vector. To build an initial state vector from a given set of input-output data or to generate equilibrium states, see `findstates(idnlhw)` and `findop(idnlhw)`. For multi-experiment data, `X0` may be a matrix whose columns give different initial states for different experiments.
  - `'z'`: (Default) Zero initial state, equivalent to a zero vector of appropriate size.

Input

`YP`: Predicted output as an `iddata` object. If `DATA` contains multiple experiments, so will `YP`.

Note If `predict` is called without an output argument,MATLAB® software displays the predicted output(s) in a plot window.

See Also

`sim(idnlhw)`

`findop(idnlhw)`

`findstates(idnlhw)`

present

Purpose

Display model information, including estimated uncertainty

Syntax

```
present(m)
```

Description

The `present` function displays the model `m`, together with the estimated standard deviations of the parameters, loss function, and Akaike's Final Prediction Error (FPE) Criterion (which essentially equals the AIC). It also displays information about how `m` was created.

`m` is any `idmodel` or `idnlmodel` object.

`present` thus gives more detailed information about the model than the standard `display` function.

Purpose

Class representing piecewise-linear nonlinear estimator for Hammerstein-Wiener models

Syntax

```
t=pwlinear('NumberOfUnits',N)
t=pwlinear('BreakPoints',BP)
t=pwlinear(Property1,Value1,...PropertyN,ValueN)
```

Description

`pwlinear` is an object that stores the piecewise-linear nonlinear estimator for estimating Hammerstein-Wiener models.

You can use the constructor to create the nonlinearity object, as follows:

`t=pwlinear('NumberOfUnits',N)` creates a piecewise-linear nonlinearity estimator object with `N` breakpoints.

`t=pwlinear('BreakPoints',BP)` creates a piecewise-linear nonlinearity estimator object with breakpoints at values `BP`.

`t=pwlinear(Property1,Value1,...PropertyN,ValueN)` creates a piecewise-linear nonlinearity estimator object specified by properties in "pwlinear Properties" on page 2-332.

Use `evaluate(p,x)` to compute the value of the function defined by the `pwlinear` object `p` at `x`.

Remarks

Use `pwlinear` to define a nonlinear function $y = F(x)$, where F is a piecewise-linear (affine) function of x and there are `n` breakpoints `(x_k,y_k), k=1,...,n`. `y_k = F(x_k)`. F is linearly interpolated between the breakpoints.

y and x are scalars.

F is also linear to the left and right of the extreme breakpoints. The slope of these extension is a function of `x_i` and `y_i` breakpoints. The breakpoints are ordered by ascending `x`-values, which is important when you set a specific breakpoint to a different value.

There are minor deviations from the breakpoint values you set and the values actually stored in the object because the toolbox represent breakpoints differently internally.

pwlinear Properties

You can include property-value pairs in the constructor to specify the object.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% List all property values
get(p)
% Get value of NumberOfUnits property
p.NumberOfUnits
```

Property Name	Description
`NumberOfUnits`	Integer specifies the number of breakpoints Default=`10`. For example: `pwlinear('NumberOfUnits',5)`
`BreakPoints`	2-*by*-`n` matrix containing the breakpoint `x` and `y` value, specified using the following format: `[x_1,x_2,...,x_n;y_1,y_2,...,y_n]` If set to a 1-*by*-`n` vector, the values are interpreted as `x`-values and the corresponding `y`-values are set to zero.

Examples

Use `pwlinear` to specify the piecewise nonlinearity estimator in Hammerstein-Wiener models. For example:

```
m=nlhw(Data,Orders,pwlinear('Br',[-1:0.1:1]),[]);
```

The piecewise nonlinearity is initialized at the specified breakpoints. The breakpoint values are adjusted to the estimation data by `nlhw`.

See Also

`nlhw`

Purpose

Plot zeros and poles with confidence interval

Syntax

```
pzmap(m)
pzmap(m,'sd',sd)
pzmap(m1,m2,m3,...)
pzmap(m1,'PlotStyle1',m2,'PlotStyle2',...,'sd',sd)
pzmap(m1,m2,m3,..,'sd',sd,'mode',mode,'axis',axis)
```

Description

`m` is any `idmodel` object: `idarx`, `idgrey`, `idss`, `idproc`, or `idpoly`.

The zeros and poles of `m` are graphed, with `o` denoting zeros and `x` denoting poles. Poles and zeros at infinity are ignored. For discrete-time models, zeros and poles at the origin are also ignored.

The Property/Value pairs `'sd'/sd, 'mode'/mode` and `` `axis'/axis `` can appear in any order. They are explained below.

If `sd` has a value larger than zero, confidence regions around the poles and zeros are also graphed. The regions corresponding to `sd` standard deviations are marked. The default value is `sd = 0`. Note that the confidence regions might sometimes stretch outside the plot, but they are always symmetric around the indicated zero or pole.

If the poles and zeros are associated with a discrete-time model, a unit circle is also drawn. For continuous-time models, the real and imaginary axes are drawn.

When `mi` contains information about several different input/output channels, you have the following options:

`mode = 'sub'` splits the screen into several plots, one for each input/output channel. These are based on the `InputName` and `OutputName` properties associated with the different models.

`mode = 'same'` gives all plots in the same diagram. Pressing the **Enter** key advances the plots.

`mode = 'sep'` erases the previous plot before the next channel pair is treated.

The default value is `mode = 'sub'`.

`axis = [x1 x2 y1 y2]` fixes the axis scaling accordingly. `axis = s` is the same as

```
axis = [-s s -s s]
```

You can select the colors associated with the different models by using the argument `PlotStyle`. Use `PlotStyle = 'b'`, `'g'`, etc. Markers and line styles are not used.

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `m.NoiseVariance` = Λ. The model can also be described with a unit variance, normalized noise source v.

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

Then,

- `pzmap(m)` plots the zeros and poles of the transfer function G.
- `pzmap(m('n'))` plots the zeros and poles of the transfer function H (ny inputs and ny outputs). The input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- If `m` is a time series, that is nu = 0, `pzmap(m)` plots the zeros and poles of the transfer function H.
- `pzmap(noisecnv(m))` plots the zeros and poles of the transfer function $[G\ H]$ (nu+ny inputs and ny outputs). The noise input channels have names `e@yname`, where `yname` is the name of the corresponding output.

- `pzmap(noisecnv(m,'norm')` plots the zeros and poles of the transfer function [*G HL*] (*nu+ny* inputs and *ny* outputs). The noise input channels have names `v@yname`, where `yname` is the name of the corresponding output.

Examples

```
mbj = bj(data,[2 2 1 1 1]);
mar = armax(data,[2 2 2 1]);
pzmap(mbj,mar,'sd',3)
```

shows all zeros and poles of two models along with the confidence regions corresponding to three standard deviations.

See Also

`idmodel`

`zpkdata`

Purpose

Estimate recursively parameters of ARMAX or ARMA models

Syntax

```
thm = rarmax(z,nn,adm,adg)
[thm,yhat,P,phi,psi] = rarmax(z,nn,adm,adg,th0,P0,phi0,psi0)
```

Description

The parameters of the ARMAX model structure

$$A(q)y(t) = B(q)u(t-nk) + C(q)e(t)$$

are estimated using a recursive prediction error method.

The input-output data is contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. `nn` is given as

```
nn = [na nb nc nk]
```

where `na`, `nb`, and `nc` are the orders of the ARMAX model, and `nk` is the delay. Specifically,

$$na: \quad A(q) = 1 + a_1 q^{-1} + \ldots + a_{na} q^{-na}$$

$$nb: \quad B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1}$$

$$nc: \quad C(q) = 1 + c_1 q^{-1} + \ldots + c_{nc} q^{-nc}$$

See“What Are Black-Box Polynomial Models?” for more information.

If `z` represents a time series `y` and `nn = [na nc]`, `rarmax` estimates the parameters of an ARMA model for `y`.

$$A(q)y(t) = C(q)e(t)$$

Only single-input, single-output models are handled by `rarmax`. Use `rpem` for the multiple-input case.

The estimated parameters are returned in the matrix `thm`. The `k`th row of `thm` contains the parameters associated with time `k`; that is, they are based on the data in the rows up to and including row `k` in `z`. Each row of `thm` contains the estimated parameters in the following order:

```
thm(k,:) = [a1,a2,...,ana,b1,...,bnb,c1,...,cnc]
```

`yhat` is the predicted value of the output, according to the current model; that is, row k of `yhat` contains the predicted value of `y(k)` based on all past data.

The actual algorithm is selected with the two arguments `adm` and `adg`. These are described under `rarx`.

The input argument `th0` contains the initial value of the parameters, a row vector consistent with the rows of `thm`. The default value of `th0` is all zeros.

The arguments `P0` and `P` are the initial and final values, respectively, of the scaled covariance matrix of the parameters. See `rarx`. The default value of `P0` is 10^4 times the unit matrix. The arguments `phi0`, `psi0`, `phi`, and `psi` contain initial and final values of the data vector and the gradient vector, respectively. The sizes of these depend on the chosen model orders. The normal choice of `phi0` and `psi0` is to use the outputs from a previous call to `rarmax` with the same model orders. (This call could be a dummy call with default input arguments.) The default values of `phi0` and `psi0` are all zeros.

Note that the function requires that the delay `nk` be larger than `0`. If you want `nk = 0`, shift the input sequence appropriately and use `nk = 1`.

Algorithm

The general recursive prediction error algorithm (11.44), (11.47) through (11.49) of Ljung (1999) is implemented. See "Algorithms for Recursive Estimation" for more information.

Examples

Compute and plot, as functions of time, the four parameters in a second-order ARMA model of a time series given in the vector `y`. The forgetting factor algorithm with a forgetting factor of 0.98 is applied.

```
thm = rarmax(y,[2 2],'ff',0.98);
plot(thm)
```

Purpose

Estimate recursively parameters of ARX or AR models

Syntax

```
thm = rarx(z,nn,adm,adg)
[thm,yhat,P,phi] = rarx(z,nn,adm,adg,th0,P0,phi0)
```

Description

The parameters of the ARX model structure

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

are estimated using different variants of the recursive least squares method.

The input-output data is contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. `nn` is given as

```
nn = [na nb nk]
```

where `na` and `nb` are the orders of the ARX model, and `nk` is the delay. Specifically,

$$na: \qquad A(q) = 1 + a_1 q^{-1} + \ldots + a_{na} q^{-na}$$

$$nb: \qquad B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1}$$

If `z` is a time series `y` and `nn = na`, `rarx` estimates the parameters of an AR model for `y`.

Models with several inputs

$$A(q)y(t) = B_1(q)u_1(t-nk_1)+ \ldots B_{nu}u_{nu}(t-nk_{nu})+e(t)$$

are handled by allowing `u` to contain each input as a column vector,

```
u = [u1 ... unu]
```

and by allowing `nb` and `nk` to be row vectors defining the orders and delays associated with each input.

Only single-output models are handled by `rarx`.

The estimated parameters are returned in the matrix `thm`. The `k`th row of `thm` contains the parameters associated with time `k`; that is, they are based on the data in the rows up to and including row `k` in `z`. Each row of `thm` contains the estimated parameters in the following order.

```
thm(k,:) = [a1,a2,...,ana,b1,...,bnb]
```

In the case of a multiple-input model, all the *b* parameters associated with input number 1 are given first, and then all the *b* parameters associated with input number 2, and so on.

`yhat` is the predicted value of the output, according to the current model; that is, row `k` of `yhat` contains the predicted value of `y(k)` based on all past data.

The actual algorithm is selected with the two arguments `adg` and `adm`. These are described in "Algorithms for Recursive Estimation". The options are as follows:

- With `adm = 'ff'` and `adg = lam` the *forgetting factor* algorithm is obtained with forgetting factor λ = `lam`. This is what is often referred to as recursive least squares (RLS). In this case the matrix `P` has the following interpretation: R_2`/2 * P` is approximately equal to the covariance matrix of the estimated parameters. Here R_2 is the variance of the innovations (the true prediction errors *e*(*t*).
- With `adm ='ug'` and `adg = gam`, the *unnormalized gradient* algorithm is obtained with gain *gamma* = `gam`. This algorithm is commonly known as normalized least mean squares (LMS).
- Similarly, `adm ='ng'` and `adg = gam` give the *normalized gradient* or normalized least mean squares (NLMS) algorithm. In these cases, `P` is not defined or applicable.
- With `adm ='kf'` and `adg = R1`, the *Kalman filter based* algorithm is obtained with R_2=`1` and R_1 = `R1`. If the variance of the innovations *e*(*t*) is not unity but R_2; then R_2 * `P` is the covariance matrix of the parameter estimates, while R_1 = `R1` /R_2 is the covariance matrix of the parameter changes.

- The input argument `th0` contains the initial value of the parameters, a row vector consistent with the rows of `thm`. The default value of `th0` is all zeros.
- The arguments `P0` and `P` are the initial and final values, respectively, of the scaled covariance matrix of the parameters. The default value of `P0` is 10^4 times the identity matrix.
- The arguments `phi0` and `phi` contain initial and final values, respectively, of the data vector.

$$\varphi(t) = [y(t-1), \ldots, y(t-na), u(t-1), \ldots u(t-nb-nk+1)]$$

Then, if

```
z = [y(1),u(1); ... ;y(N),u(N)]
```

you have `phi0`= $\varphi(1)$ and `phi`= $\varphi(N)$. The default value of `phi0` is all zeros. For online use of `rarx`, use `phi0`, `th0`, and `P0` as the previous outputs `phi`, `thm` (last row), and `P`.

Note that the function requires that the delay `nk` be larger than `0`. If you want `nk = 0`, shift the input sequence appropriately and use `nk = 1`. See `nkshift`.

Examples

Adaptive noise canceling: The signal *y* contains a component that has its origin in a known signal *r*. Remove this component by estimating, recursively, the system that relates *r* to *y* using a sixth-order FIR model together with the NLMS algorithm.

```
z = [y r];
[thm,noise] = rarx(z,[0 6 1],'ng',0.1);
% noise is the adaptive estimate of the noise
% component of y
plot(y-noise)
```

If the above application is a true online one, so that you want to plot the best estimate of the signal `y - noise` at the same time as the data *y* and *u* become available, proceed as follows.

```
phi = zeros(6,1); P=1000*eye(6);
th = zeros(1,6); axis([0 100 -2 2]);
plot(0,0,'*'), hold on
% The loop:
while ~abort
[y,r,abort] = readAD(time);
[th,ns,P,phi] = rarx([y r],'ff',0.98,th,P,phi);
plot(time,y-ns,'*')
time = time +Dt
end
```

This example uses a forgetting factor algorithm with a forgetting factor of 0.98. `readAD` represents an M-file that reads the value of an A/D converter at the indicated time instant.

See Also

`nkshift`

`rarmax`

`rbj`

`roe`

`rpem`

`rplr`

Purpose

Estimate recursively parameters of Box-Jenkins models

Syntax

```
thm = rbj(z,nn,adm,adg)
[thm,yhat,P,phi,psi] = ... rbj(z,nn,adm,adg,th0,P0,phi0,psi0)
```

Description

The parameters of the Box-Jenkins model structure

$$y(t) = \frac{B(q)}{F(q)}u(t-nk) + \frac{C(q)}{D(q)}e(t)$$

are estimated using a recursive prediction error method.

The input-output data is contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. `nn` is given as

```
nn = [nb nc nd nf nk]
```

where `nb`, `nc`, `nd`, and `nf` are the orders of the Box-Jenkins model, and `nk` is the delay. Specifically,

$$nb: \quad B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1}$$

$$nc: \quad C(q) = 1 + c_1 q^{-1} + \ldots + c_{nc} q^{-nc}$$

$$nd: \quad D(q) = 1 + d_1 q^{-1} + \ldots + d_{nd} q^{-nd}$$

$$nf: \quad F(q) = 1 + f_1 q^{-1} + \ldots + f_{nf} q^{-nf}$$

See“What Are Black-Box Polynomial Models?” for more information.

Only single-input, single-output models are handled by `rbj`. Use `rpem` for the multiple-input case.

The estimated parameters are returned in the matrix `thm`. The `k`th row of `thm` contains the parameters associated with time `k`; that is, they are based on the data in the rows up to and including row `k` in `z`. Each row of `thm` contains the estimated parameters in the following order.

```
thm(k,:) = [b1,...,bnb,c1,...,cnc,d1,...,dnd,f1,...,fnf]
```

`yhat` is the predicted value of the output, according to the current model; that is, row `k` of `yhat` contains the predicted value of `y(k)` based on all past data.

The actual algorithm is selected with the two arguments `adm` and `adg`. These are described under `rarx`.

The input argument `th0` contains the initial value of the parameters, a row vector consistent with the rows of `thm`. The default value of `th0` is all zeros.

The arguments `P0` and `P` are the initial and final values, respectively, of the scaled covariance matrix of the parameters. See `rarx`. The default value of `P0` is 10^4 times the unit matrix. The arguments `phi0`, `psi0`, `phi`, and `psi` contain initial and final values of the data vector and the gradient vector, respectively. The sizes of these depend on the chosen model orders. The normal choice of `phi0` and `psi0` is to use the outputs from a previous call to `rbj` with the same model orders. (This call could be a dummy call with default input arguments.) The default values of `phi0` and `psi0` are all zeros.

Note that the function requires that the delay `nk` be larger than `0`. If you want `nk = 0`, shift the input sequence appropriately and use `nk = 1`.

Algorithm

The general recursive prediction error algorithm (11.44) of Ljung (1900) is implemented. See also “Algorithms for Recursive Estimation”.

See Also

`nkshift`

`rarmax`

`rarx`

`roe`

`rpem`

`rplr`

Purpose

Determine whether `iddata` is based on real-valued signals

Syntax

```
realdata(data)
```

Description

`realdata` returns 1 if

- `data` contains only real-valued signals.
- `data` contains frequency-domain signals, obtained by Fourier transformation of real-valued signals.

Otherwise `realdata` returns `0`.

Notice the difference with `isreal`:

```
load iddata1
isreal(z1); % returns 1
zf = fft(z1);
isreal(zf) % returns 0
realdata(zf) % returns 1
zf = complex(zf) % adds negative frequencies to zf
realdata(zf) % still returns 1
```

Purpose

Resample time-domain data by decimation or interpolation (requires Signal Processing Toolbox™ software)

Syntax

```
resample(data,P,Q)
resample(data,P,Q,order)
```

Description

`resample(data,P,Q)` resamples data such that the data is interpolated by a factor `P` and then decimated by a factor `Q`. `resample(z,1,Q)` results in decimation by a factor `Q`.

`resample(data,P,Q,order)` filters the data by applying a filter of specified `order` before interpolation and decimation.

Input

`data`

Name of time-domain `iddata` object. Can be input-output or time-series data.

Data must be sampled at equal time intervals.

`P`, `Q`

Integers that specify the resampling factor, such that the new sampling interval is `Q/P` times the original one.

`(Q/P)>1` results in decimation and `(Q/P)<1` results in interpolation.

`order`

Order of the filters applied before interpolation and decimation.

Default: `10`

Algorithm

If you have installed the Signal Processing Toolbox software, `resample` calls the Signal Processing Toolbox `resample` function. The algorithm takes into account the intersample characteristics of the input signal, as described by `data.InterSample`.

Examples

In this example, you increase the sampling rate by a factor of `1.5` and compare the resampled and the original data signals.

```
plot(u)
ur = resample(u,3,2);
plot(u,ur)
```

See Also

idresamp

Purpose

Compute and test model residuals (prediction errors)

Syntax

```
resid(m,data)
resid(m,data,Type)
resid(m,data,Type,M)
e = resid(m,data);
```

Description

`data` contains the output-input data as an `iddata` object. Both time-domain and frequency-domain data are supported. `data` can also be an `idfrd` object.

`m` is any `idmodel` or `idnlmodel` object.

In all cases the residuals *e* associated with the data and the model are computed. This is done as in the command `pe` with a default choice of `init`.

When called without output arguments, `resid` produces a plot. The plot can be of three kinds depending on the argument `Type`:

- `Type = 'Corr'` (only available for time-domain data): The autocorrelation function of `e` and the cross correlation between `e` and the input(s) `u` are computed and displayed. The 99% confidence intervals for these values are also computed and shown as a yellow region. The computation of the confidence region is done assuming `e` to be white and independent of `u`. The functions are displayed up to lag `M`, which is 25 by default.
- `Type = 'ir'`: The impulse response (up to lag `M`, which is 25 by default) from the input to the residuals is plotted with a 99% confidence region around zero marked as a yellow area. Negative lags up to `M/4` are also included to investigate feedback effects. (The result is the same as `impulse(e,'sd',2.58,',M))`.
- `Type = 'fr'`: The frequency response from the input to the residuals (based on a high-order FIR model) is shown as a Bode plot. A 99% confidence region around zero is also marked as a yellow area.

The default for time-domain data is `Type = 'Corr'`. For frequency-domain data, the default is `Type = 'fr'`.

With an output argument, no plot is produced, and `e` is returned with the residuals (prediction errors) associated with the model and the data. It is an `iddata` object with the residuals as outputs and the input in `data` as inputs. That means that `e` can be directly used to build model error models, that is, models that describe the dynamics from the input to the residuals (which should be negligible if `m` is a good description of the system).

Examples

Here are some typical model validation commands.

```
e = resid(m,data);
plot(e)
compare(data,m);
```

To compute a model error model, that is, a model to input to the residuals to see if any essential unmodeled dynamics are left, do the following:

```
e = resid(m,data);
me = arx(e,[10 10 0]);
bode(me,'sd',3,fill')
```

References

Ljung (1999), Section 16.6.

See Also

`compare`

`predict`

`sim`

`simsd`

Purpose

Estimate recursively output-error models (IIR-filters)

Syntax

```
thm = roe(z,nn,adm,adg)
[thm,yhat,P,phi,psi] = roe(z,nn,adm,adg,th0,P0,phi0,psi0)
```

Description

The parameters of the output-error model structure

$$y(t) = \frac{B(q)}{F(q)}u(t-nk) + e(t)$$

are estimated using a recursive prediction error method.

The input-output data are contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. `nn` is given as

```
nn = [nb nf nk]
```

where `nb` and `nf` are the orders of the output-error model, and `nk` is the delay. Specifically,

$$nb: \qquad B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1}$$

$$nf: \qquad F(q) = 1 + f_1 q^{-1} + \ldots + f_{nf} q^{-nf}$$

See“What Are Black-Box Polynomial Models?” for more information.

Only single-input, single-output models are handled by `roe`. Use `rpem` for the multiple-input case.

The estimated parameters are returned in the matrix `thm`. The `k`th row of `thm` contains the parameters associated with time `k`; that is, they are based on the data in the rows up to and including row `k` in `z`.

Each row of `thm` contains the estimated parameters in the following order.

```
thm(k,:) = [b1,...,bnb,f1,...,fnf]
```

`yhat` is the predicted value of the output, according to the current model; that is, row `k` of `yhat` contains the predicted value of `y(k)` based on all past data.

The actual algorithm is selected with the two arguments `adg` and `adm`. These are described under `rarx`.

The input argument `th0` contains the initial value of the parameters, a row vector consistent with the rows of `thm`. The default value of `th0` is all zeros.

The arguments `P0` and `P` are the initial and final values, respectively, of the scaled covariance matrix of the parameters. See `rarx`. The default value of `P0` is 10^4 times the unit matrix. The arguments `phi0`, `psi0`, `phi`, and `psi` contain initial and final values of the data vector and the gradient vector, respectively. The sizes of these depend on the chosen model orders. The normal choice of `phi0` and `psi0` is to use the outputs from a previous call to `roe` with the same model orders. (This call could be a dummy call with default input arguments.) The default values of `phi0` and `psi0` are all zeros.

Note that the function requires that the delay `nk` be larger than `0`. If you want `nk = 0`, shift the input sequence appropriately and use `nk = 1`.

Algorithm

The general recursive prediction error algorithm (11.44) of Ljung (1999) is implemented. See also “Algorithms for Recursive Estimation”.

See Also

`nkshift`

`rarmax`

`rarx`

`rbj`

`rpem`

`rplr`

Purpose

Estimate general input-output models using recursive prediction-error minimization method

Syntax

```
thm = rpem(z,nn,adm,adg)
[thm,yhat,P,phi,psi] = rpem(z,nn,adm,adg,th0,P0,phi0,psi0)
```

Description

The parameters of the general linear model structure

$$A(q)y(t) = \frac{B_1(q)}{F_1(q)}u_1(t-nk_1) + \ldots + \frac{B_{nu}(q)}{F_{nu}(q)}u_{nu}(t-nk_{nu}) + \frac{C(q)}{D(q)}e(t)$$

are estimated using a recursive prediction error method.

The input-output data is contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. (In the multiple-input case, `u` contains one column for each input.) `nn` is given as

```
nn = [na nb nc nd nf nk]
```

where `na`, `nb`, `nc`, `nd`, and `nf` are the orders of the model, and `nk` is the delay. For multiple-input systems, `nb`, `nf`, and `nk` are row vectors giving the orders and delays of each input. See "What Are Black-Box Polynomial Models?" for an exact definition of the orders.

The estimated parameters are returned in the matrix `thm`. The `k`th row of `thm` contains the parameters associated with time `k`; that is, they are based on the data in the rows up to and including row `k` in `z`. Each row of `thm` contains the estimated parameters in the following order.

```
thm(k,:) = [a1,a2,...,ana,b1,...,bnb,...
  c1,...,cnc,d1,...,dnd,f1,...,fnf]
```

For multiple-input systems, the B part in the above expression is repeated for each input before the C part begins, and the F part is also repeated for each input. This is the same ordering as in `m.par`.

`yhat` is the predicted value of the output, according to the current model; that is, row `k` of `yhat` contains the predicted value of `y(k)` based on all past data.

The actual algorithm is selected with the two arguments `adg` and `adm`. These are described under `rarx`.

The input argument `th0` contains the initial value of the parameters, a row vector consistent with the rows of `thm`. The default value of `th0` is all zeros.

The arguments `P0` and `P` are the initial and final values, respectively, of the scaled covariance matrix of the parameters. See `rarx`. The default value of `P0` is 10^4 times the unit matrix. The arguments `phi0`, `psi0`, `phi`, and `psi` contain initial and final values of the data vector and the gradient vector, respectively. The sizes of these depend on the chosen model orders. The normal choice of `phi0` and `psi0` is to use the outputs from a previous call to `rpem` with the same model orders. (This call could be a dummy call with default input arguments.) The default values of `phi0` and `psi0` are all zeros.

Note that the function requires that the delay `nk` be larger than `0`. If you want `nk = 0`, shift the input sequence appropriately and use `nk = 1`.

Algorithm

The general recursive prediction error algorithm (11.44) of Ljung (1999) is implemented. See also “Algorithms for Recursive Estimation”.

For the special cases of ARX/AR models, and of single-input ARMAX/ARMA, Box-Jenkins, and output-error models, it is more efficient to use `rarx`, `rarmax`, `rbj`, and `roe`.

See Also

`nkshift`

`rarmax`

`rarx`

`rbj`

`roe`

`rplr`

Purpose

Estimate general input-output models using recursive pseudolinear regression method

Syntax

```
thm = rplr(z,nn,adm,adg)
[thm,yhat,P,phi] = rplr(z,nn,adm,adg,th0,P0,phi0)
```

Description

This is a direct alternative to `rpem` and has essentially the same syntax. See `rpem` for an explanation of the input and output arguments.

`rplr` differs from `rpem` in that it uses another gradient approximation. See Section 11.5 in Ljung (1999). Several of the special cases are well-known algorithms.

When applied to ARMAX models, (`nn = [na nb nc 0 0 nk]`), `rplr` gives the extended least squares (ELS) method. When applied to the output-error structure (`nn = [0 nb 0 0 nf nk]`), the method is known as HARF in the `adm = 'ff'` case and SHARF in the `adm = 'ng'` case.

`rplr` can also be applied to the time-series case in which an ARMA model is estimated with

```
z = y
nn = [na nc]
```

You can thus use `rplr` as an alternative to `rarmax` for this case.

See Also

`nkshift`

`rarmax`

`rarx`

`rbj`

`roe`

`rpem`

saturation

Purpose

Class representing saturation nonlinearity estimator for Hammerstein-Wiener models

Syntax

```
s=saturation(LinearInterval,L)
```

Description

`saturation` is an object that stores the saturation nonlinearity estimator for estimating Hammerstein-Wiener models.

You can use the constructor to create the nonlinearity object, as follows:

`s=saturation(LinearInterval,L)` creates a saturation nonlinearity estimator object, initialized with the linear interval `L`.

Use `evaluate(s,x)` to compute the value of the function defined by the `saturation` object `s` at `x`.

Remarks

Use `saturation` to define a nonlinear function $y = F(x)$, where F is a function of x and has the following characteristics:

$$\begin{aligned} a \le x < b \quad & F(x) = x \\ a > x \quad & F(x) = a \\ b \le x \quad & F(x) = b \end{aligned}$$

y and x are scalars.

saturation Properties

You can specify the property value as an argument in the constructor to specify the object.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% List LinearInterval property value
get(s)
s.LinearInterval
```

You can also use the `set` function to set the value of particular properties. For example:

```
set(s, 'LinearInterval', [-1.5 1.5])
```

The first argument to `set` must be the name of a MATLAB® variable.

Property Name	Description
`LinearInterval`	1-*by*-2 row vector that specifies the initial interval of the saturation Default=`[NaN NaN]`. For example: `saturation('LinearInterval',[-1.5 1.5])`

Examples

Use `saturation` to specify the saturation nonlinearity estimator in Hammerstein-Wiener models. For example:

```
m=nlhw(Data,Orders,saturation([-1 1]),[]);
```

The saturation nonlinearity is initialized at the interval `[-1 1]`. The interval values are adjusted to the estimation data by `nlhw`.

See Also

`nlhw`

Purpose

Segment data and estimate models for each segment

Syntax

```
segm = segment(z,nn)
[segm,V,thm,R2e] = segment(z,nn,R2,q,R1,M,th0,P0,ll,mu)
```

Description

`segment` builds models of AR, ARX, or ARMAX/ARMA type,

$$A(q)y(t) = B(q)u(t-nk) + C(q)e(t)$$

assuming that the model parameters are piecewise constant over time. It results in a model that has split the data record into segments over which the model remains constant. The function models signals and systems that might undergo abrupt changes.

The input-output data is contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. If the system has several inputs, `u` has the corresponding number of columns.

The argument `nn` defines the model order. For the ARMAX model

```
nn = [na nb nc nk]
```

where `na`, `nb`, and `nc` are the orders of the corresponding polynomials. See “What Are Black-Box Polynomial Models?”. Moreover, `nk` is the delay. If the model has several inputs, `nb` and `nk` are row vectors, giving the orders and delays for each input.

For an ARX model (`nc = 0`) enter

```
nn = [na nb nk]
```

For an ARMA model of a time series

```
z = y
nn = [na nc]
```

and for an AR model

```
nn = na
```

The output argument `segm` is a matrix, whose `k` row contains the parameters corresponding to time `k`. This is analogous to the output argument `thm` in `rarx` and `rarmax`. The output argument `thm` of `segment` contains the corresponding model parameters that have not yet been segmented. That is, they are not piecewise constant, and therefore correspond to the outputs of the functions `rarmax` and `rarx`. In fact, `segment` is an alternative to these two algorithms, and has a better capability to deal with time variations that might be abrupt.

The output argument `V` contains the sum of the squared prediction errors of the segmented model. It is a measure of how successful the segmentation has been.

The input argument `R2` is the assumed variance of the innovations $e(t)$ in the model. The default value of `R2`, `R2 = []`, is that it is estimated. Then the output argument `R2e` is a vector whose `k`th element contains the estimate of `R2` at time `k`.

The argument `q` is the probability that the model undergoes at an abrupt change at any given time. The default value is `0.01`.

`R1` is the assumed covariance matrix of the parameter jumps when they occur. The default value is the identity matrix with dimension equal to the number of estimated parameters.

`M` is the number of parallel models used in the algorithm (see below). Its default value is `5`.

`th0` is the initial value of the parameters. Its default is zero. `P0` is the initial covariance matrix of the parameters. The default is 10 times the identity matrix.

`ll` is the guaranteed life of each of the models. That is, any created candidate model is not abolished until after at least `ll` time steps. The default is `ll = 1`. `Mu` is a forgetting parameter that is used in the scheme that estimates `R2`. The default is `0.97`.

The most critical parameter for you to choose is `R2`. It is usually more robust to have a reasonable guess of `R2` than to estimate it. Typically, you need to try different values of `R2` and evaluate the results. (See the example below.) `sqrt(R2)` corresponds to a change in the value $y(t)$

that is normal, giving no indication that the system or the input might have changed.

Algorithm

The algorithm is based on `M` parallel models, each recursively estimated by an algorithm of Kalman filter type. Each is updated independently, and its posterior probability is computed. The time-varying estimate `thm` is formed by weighting together the `M` different models with weights equal to their posterior probability. At each time step the model (among those that have lived at least `ll` samples) that has the lowest posterior probability is abolished. A new model is started, assuming that the system parameters have jumped, with probability `q`, a random jump from the most likely among the models. The covariance matrix of the parameter change is set to `R1`.

After all the data are examined, the surviving model with the highest posterior probability is tracked back and the time instances where it jumped are marked. This defines the different segments of the data. (If no models had been abolished in the algorithm, this would have been the maximum likelihood estimates of the jump instances.) The segmented model `segm` is then formed by smoothing the parameter estimate, assuming that the jump instances are correct. In other words, the last estimate over a segment is chosen to represent the whole segment.

Examples

Check how the algorithm segments a sinusoid into segments of constant levels. Then use a very simple model $y(t) = b_1 * 1$, where `1` is a fake input and b_1 describes the piecewise constant level of the signal $y(t)$ (which is simulated as a sinusoid).

```
y = sin([1:50]/3)';
thm = segment([y,ones(length(y),1)],[0 1 1],0.1);
plot([thm,y])
```

By trying various values of `R2` (`0.1` in the above example), more levels are created as `R2` decreases.

Purpose

Select model order for single-output ARX models

Syntax

```
nn = selstruc(v)
[nn,vmod] = selstruc(v,c)
```

Description

Note Use `selstruc` for single-output systems only. `selstruc` supports both single-input and multiple-input systems.

`selstruc` is a function to help choose a model structure (order) from the information contained in the matrix `v` obtained as the output from `arxstruc` or `ivstruc`.

The default value of `c` is `'plot'`. The plot shows the percentage of the output variance that is not explained by the model as a function of the number of parameters used. Each value shows the best fit for that number of parameters. By clicking in the plot you can examine which orders are of interest. When you click `'Select'`, the variable `nn` is returned in the workspace as the optimal model structure for your choice of number of parameters. Several choices can be made.

`c = 'aic'` gives no plots, but returns in `nn` the structure that minimizes Akaike's Information Criterion (AIC),

$$V_{mod} = V\left(1 + \frac{2d}{N}\right)$$

where V is the loss function, d is the total number of parameters in the structure in question, and N is the number of data points used for the estimation. See `aic` for more details.

`c = 'mdl'` returns in `nn` the structure that minimizes Rissanen's Minimum Description Length (MDL) criterion.

$$V_{mod} = V\left(1 + \frac{d\log(N)}{N}\right)$$

When `c` equals a numerical value, the structure that minimizes

$$V_{mod} = V\left(1 + \frac{cd}{N}\right)$$

is selected.

The output argument `vmod` has the same format as `v`, but it contains the logarithms of the accordingly modified criteria.

Examples

```
V = arxstruc(data(1:200),data(201:400),...
             struc(1:10,1:10,1:10))
nn = selstruc(V,0); %best fit to validation data
m = arx(data,nn)
```

Purpose

Set properties of data and model objects

Syntax

```
set(m,'Property',Value)
set(m,'Property1',Value1,...'PropertyN',ValueN)
set(m,'Property')
set(m)
```

Description

`set` is used to set or modify the properties of any of the objects in the toolbox (`iddata`, `idmodel`, `idgrey`, `idarx`, `idpoly`, `idss`, `idnlarx`, `idnlgrey`, `idnlhw`). See the corresponding reference pages for a complete list of properties.

`set(m,'Property',Value)` assigns the value `Value` to the property of the object `m` specified by the string `'Property'`. This string can be the full property name (for example, `'SSParameterization'`) or any unambiguous case-insensitive abbreviation (for example, `'ss'`).

`set(m,'Property1',Value1,...'PropertyN',ValueN)` sets multiple properties with a single statement. In certain cases this might be necessary, since the model `m` must, for example, have state-space matrices of consistent dimensions after each `set` statement.

`set(m,'Property')` displays admissible values for the property specified by `'Property'`.

`set(m)` displays all assignable values of `m` and their admissible values.

The same result is also obtained by subassignment.

```
m.Property = Value
```

setinit

Purpose

Set initial states of `idnlgrey` model object

Syntax

```
setinit(model)
setinit(model,prop,values)
```

Input

`model`
: Name of the `idnlgrey` model object.

`Property`
: Name of the `InitialStates` model property field, such as `'Name'`, `'Unit'`, `'Value'`, `'Minimum'`, `'Maximum'`, and `'Fixed'`.

`Values`
: Values of the specified property `Property`. `Values` are an `Nx`-by-1 cell array of values, where `Nx` is the number of states.

Description

`setinit(model)` sets the initial-state values in the `'Value'` field of the `InitialStates` model property.

`setinit(model,prop,values)` sets the `values` of the `prop` field of the `InitialStates` model property. `prop` can be `'Name'`, `'Unit'`, `'Value'`, `'Minimum'`, `'Maximum'`, and `'Fixed'`.

See Also

`getinit`

`getpar`

`idnlgrey`

`setpar`

Purpose

Set initial parameter values of `idnlgrey` model object

Syntax

```
setpar(model)
setpar(model,prop)
```

Input

`model`
: Name of the `idnlgrey` model object.

`Property`
: Name of the `Parameters` model property field, such as `'Name'`, `'Unit'`, `'Value'`, `'Minimum'`, or `'Maximum'`.

  Default: `'Value'`.

Description

`setpar(model)` sets the initial model parameter values in the `'Value'` field of the `Parameters` model property.

`setpar(model,prop)` sets the model parameter values in the `prop` field of the `Parameters` model property. `prop` can be `'Name'`, `'Unit'`, `'Value'`, `'Minimum'`, and `'Maximum'`.

See Also

`getinit`

`getpar`

`idnlgrey`

`setinit`

setpname

Purpose

Set mnemonic parameter names for black-box model structures

Syntax

```
model = setpname(model)
```

Description

`model` is an `idmodel` object of `idarx`, `idpoly`, `idproc`, or `idss` type. The returned model has the `'PName'` property set to a cell array of strings that correspond to the symbols used in this manual to describe the parameters.

For single-input `idpoly` models, the parameters are called `'a1', 'a2', ...,'fn'`.

For multiple-input `idpoly` models, the *b* and *f* parameters have the output/input channel number in parentheses, as in `'b1(1,2)'`, `'f3(1,2)'`, etc.

For `idarx` models, the parameter names are as in `'-A(ky,ku)'` for the negative value of the *ky*-*ku* entry of the matrix in *A(q)* polynomial of the multiple-output ARX equation, and similarly for the *B* parameters.

For `idss` models, the parameters are named for the matrix entries they represent, such as `'A(4,5)'`, `'K(2,3)'`, etc.

For `idproc` models, the parameter names are as described under `idproc`.

This function is particularly useful when certain parameters are to be fixed. See the property `FixedParameter` under `Algorithm Properties`.

Purpose

Set matrix structure for `idss` objects

Syntax

```
setstruc(M,As,Bs,Cs,Ds.Ks,XOs)setstruc(M,Mods)
```

Description

```
setstruc(M,As,Bs,Cs,Ds.Ks,XOs)
```

is the same as

```
set(M,'As',As,'Bs',Bs,'Cs',Cs,'Ds',Ds,'Ks',Ks,'XOs',XOs)
```

Use empty matrices for structure matrices that should not be changed. You can omit trailing arguments.

In the alternative syntax, `Mods` is a structure with field names `As`, `Bs`, etc., with the corresponding values of the fields.

See Also

`idss`

sigmoidnet

Purpose

Class representing sigmoid network nonlinearity estimator for nonlinear ARX and Hammerstein-Wiener models

Syntax

```
s=sigmoidnet('NumberOfUnits',N)
s=sigmoidnet(Property1,Value1,...PropertyN,ValueN)
```

Description

`sigmoidnet` is an object that stores the sigmoid network nonlinear estimator for estimating nonlinear ARX and Hammerstein-Wiener models.

You can use the constructor to create the nonlinearity object, as follows:

`s=sigmoidnet('NumberOfUnits',N)` creates a sigmoid nonlinearity estimator object with `N` terms in the sigmoid expansion.

`s=sigmoidnet(Property1,Value1,...PropertyN,ValueN)` creates a sigmoid nonlinearity estimator object specified by properties in "sigmoidnet Properties" on page 2-367.

Use `evaluate(s,x)` to compute the value of the function defined by the `sigmoidnet` object `s` at `x`.

Remarks

Use `sigmoidnet` to define a nonlinear function $y = F(x)$, where y is scalar and x is an `m`-dimensional row vector. The sigmoid network function is based on the following expansion:

$$F(x) = (x-r)PL + a_1 f\left((x-r)Qb_1 - c_1\right) + \ldots + a_n f\left((x-r)Qb_n - c_n\right) + d$$

where f is the sigmoid function, given by the following equation:

$$f(z) = \frac{1}{e^{-z}+1}$$

P and Q are `m`-*by*-`p` and `m`-*by*-`q` projection matrices. The projection matrices P and Q are determined by principal component analysis of estimation data. Usually, `p=m`. If the components of x in the estimation data are linearly dependent, then `p<m`. The number of columns of Q,

q, corresponds to the number of components of x used in the sigmoid function.

When used in a nonlinear ARX model, q is equal to the size of the NonlinearRegressors property of the idnlarx object. When used in a Hammerstein-Wiener model, m=q=1 and Q is a scalar.

r is a 1-*by*-m vector and represents the mean value of the regressor vector computed from estimation data.

d, a_k, and c_k are scalars.

L is p-*by*-1 vector.

b_k are q-*by*-1 vectors.

sigmoidnet Properties

You can include property-value pairs in the constructor to specify the object.

After creating the object, you can use get or dot notation to access the object property values. For example:

```
% List all property values
get(s)
% Get value of NumberOfUnits property
s.NumberOfUnits
```

You can also use the set function to set the value of particular properties. For example:

```
set(s, 'LinearTerm', 'on')
```

The first argument to set must be the name of a MATLAB® variable.

Property Name	Description
`NumberOfUnits`	Integer specifies the number of nonlinearity units in the expansion. Default=`10`. For example: `sigmoidnet(H,'NumberOfUnits',5)`
`LinearTerm`	Can have the following values: • `'on'`—Estimates the vector L in the expansion. • `'off'`—Fixes the vector L to zero. For example: `sigmoidnet(H,'LinearTerm','on')`
`Parameters`	A structure containing the parameters in the nonlinear expansion, as follows: • `RegressorMean`: 1-*by*-`m` vector containing the means of `x` in estimation data, `r`. • `NonLinearSubspace`: `m`-*by*-`q` matrix containing Q. • `LinearSubspace`: `m`-*by*-`p` matrix containing P. • `LinearCoef`: `p`-*by*-1 vector L. • `Dilation`: `q`-*by*-`n` matrix containing the values `b_k`. • `Translation`: 1-*by*-`n` vector containing the values `c_k`. • `OutputCoef`: `n`-*by*-1 vector containing the values `a_k`. • `OutputOffset`: scalar `d`. Typically, the values of this structure are set by estimating a model with a `sigmoidnet` nonlinearity.

Examples

Use `sigmoidnet` to specify the nonlinear estimator in nonlinear ARX and Hammerstein-Wiener models. For example:

```
m=nlarx(Data,Orders,sigmoidnet('num',5));
```

See Also

`nlarx`

`nlhw`

Purpose

Simulate linear models with confidence interval

Syntax

```
y = sim(m,u)
y = sim(m,u,'noise')
[y, ysd] = sim(m,u,'InitialState',init)
```

Description

`m` is any `idmodel` or `idnlmodel` object.

`u` is an `iddata` object, containing inputs only. (Any outputs are ignored). Both time-domain and frequency-domain signals are supported. The number of input channels in `u` must either be equal to the number of inputs of the model `m` or equal to the sum of the number of inputs and noise sources (number of outputs). In the latter case the last inputs in `u` are regarded as noise sources and a noise-corrupted simulation is obtained. The noise is scaled according to the property `m.NoiseVariance` in `m`. To obtain the right noise level according to the model, the noise inputs should be white noise with zero mean and unit covariance matrix. A simpler way of obtaining a noise-corrupted simulation with Gaussian noise is to add the argument `'noise'`. If no noise sources are contained in `u`, a noise-free simulation is obtained. `sim` applies both to time-domain and frequency-domain `iddata` objects, but no standard deviations are obtained for frequency-domain signals.

`sim` returns `y`, containing the simulated output, as an `iddata` object.

`init` gives access to the initial states:

- `init = 'm'` (default) uses the internally stored initial state of model `m`.
- `init = 'z'` uses zero initial state.
- `init = x0`, where `x0` is a column vector of appropriate length, uses this value as the initial state. For multi-experiment inputs, `x0` has as many columns as there are experiments to allow for different initial conditions. For a continuous-time model `m`, `x0` is the initial state for this model. Any modifications of the initial state that sampling might

require are automatically handled. If `m` has a non-zero `InputDelay`, and you need to access the values of the inputs during this delay, you must first apply `inpd2nk(m)`. If `m` is a continuous-time model, it must first be sampled before `inpd2nk` can be applied.

The second output argument `ysd` is the standard deviation of the simulated output. This is not available for frequency-domain data.

`u` can also be given as a matrix with the number of columns being either the number of inputs in `m` or the sum of the number of inputs and outputs. Then `y` and `ysd` are returned as matrices. Continuous-time models, however, require `u` to be given as `iddata`.

If `m` is a continuous-time model, it is first converted to discrete time with the sampling interval given by `ue`, taking into account the inter-sample behavior of the input (`ue.InterSample`).

Examples

Simulate a given system `m0` (for example, created by `idpoly`).

```
e = iddata([],randn(500,1));
u = iddata([],idinput(500,'prbs'));
y = sim(m0,[u e]);
% iddata object with output y and input u.
z = [y u];
```

The same result is obtained by

```
u = iddata([],idinput(500,'prbs'));
y = sim(m0,u,'noise');
z = [ y u];
```

or

```
u = idinput(500,'prbs');
y = sim(m0,u,'noise');
z = iddata(y,u);
```

Validate a model by comparing a measured output `y` with one simulated using an estimated model `m`.

```
yh = sim(m,u);
plot(y,yh)
```

See Also

`compare`

`idmdlsim`

`pe`

`predict`

`simsd`

Purpose

Simulate nonlinear ARX model

Syntax

```
YS = sim(MODEL,U)
YS = sim(MODEL,U,'Noise')
YS = sim(MODEL,U,'InitialState',INIT)
```

Description

`YS = sim(MODEL,U)` simulates a dynamic system with an `idnlarx` model.

`YS = sim(MODEL,U,'Noise')` produces a noise corrupted simulation with an additive Gaussian noise scaled according to the value of the `NoiseVariance` property of `MODEL`.

`YS = sim(MODEL,U,'InitialState',INIT)` specifies the initial conditions for simulation using various options, such as numerical initial state vector or past I/O data.

To simulate the model with user-defined noise, set the input `U = [UIN E]`, where `UIN` is the input signal and `E` is the noise signal. `UIN` and `E` must both be one of the following:

- `iddata` objects: `E` stores the noise signals as inputs, where the number of inputs matches the number of model outputs.N_y is the number of model outputs.
- Matrices: `E` has as many columns as there are noise signals, corresponding to the number of model outputs.

Input

- `MODEL`: `idnlarx` model object.
- `U`: Input data for simulation, an `iddata` object (where only the input channels are used) or a matrix. For noisy simulations, `U` contains both input and noise channels.
- `INIT`: Initial condition specification. `INIT` can be one of the following:
  - A real column vector `X0`, for the state vector corresponding to an appropriate number of output and input data samples prior to the simulation start time. To build an initial state vector from a given set of input-output data or to generate equilibrium

states, see `data2state(idnlarx)`, `findstates(idnlarx)` and `findop(idnlarx)`. For multi-experiment data, `X0` may be a matrix whose columns give different initial states for different experiments.

- `'z'`: (Default) Zero initial state, equivalent to a zero vector of appropriate size.
- `iddata` object containing output and input data samples prior to the simulation start time. If it contains more data samples than necessary, only the last samples are taken into account. This syntax is equivalent to`sim(MODEL, U, 'InitialState', data2state(MODEL,INIT))` where `data2state(idnlarx)` transforms the `iddata` object `INIT` to a state vector.

Input

- `YS`: Simulated output. An `iddata` object if `U` is an `iddata` object, a matrix otherwise.

Note If `sim` is called without an output argument, MATLAB® software displays the simulated output(s) in a plot window.

Examples

Simulation of single-input single-output `idnlarx` model

In this example you simulate a single-input single-output `idnlarx` model `M` about a known equilibrium condition, with an input level of 1 and output level of 10.

1 Load sample data.

```
load iddata2;
```

2 Estimate an `idnlarx` model from data.

```
M = nlarx(z2, [2 2 1], 'tree');
```

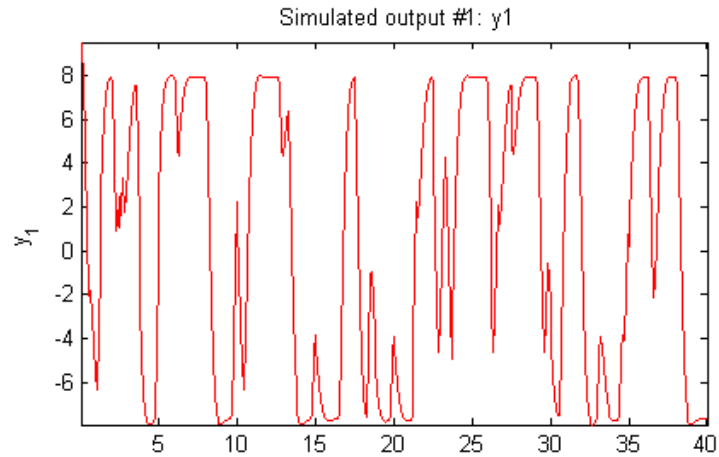
3 Estimate current states of model based on past data.

```
x0 = data2state(M, struct('Input',1, 'Output', 10));
```

4 Simulate the model using the initial states returned by `data2state`.

```
sim(M, z2, 'init', x0);
```

MATLAB software responds with a plot of the simulated response.



Continuing from End of Previous Simulation

In this example you continue the simulation of a model from the end of a previous simulation run.

1 Estimate the `idnlarx` model from data.

```
load iddata2
M = nlarx(z2, [2 2 1], 'tree'); % idnlarx model
```

2 Simulate the model using first half of input data of `z2`

```
u1 = z2(1:200,[]);
```

```
% Simulate starting from zero initial states
ys1 = sim(M, u1, 'init', 'z');
```

3 Start another simulation, using the same states of the model from when the first simulation ended. For the second simulation, you use the second half of the input data of `z2`.

```
% input data for next simulation (second half of z2's input)
u2 = z2(201:end, []);
```

4 In order to set the initial states for second simulation correctly, package input `u1` and output `ys1` from the first simulation into one `iddata` object.

```
firstSimData = [ys1,u1];
```

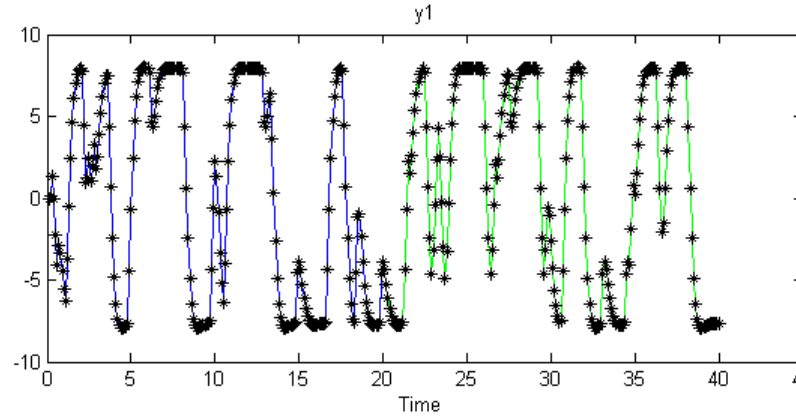
5 Pass this data as initial conditions for the next simulation.

```
ys2 = sim(M, u2, 'init', firstSimData);
```

6 Verify the two simulations by comparing to a complete simulation using all the input data of `z2`.

```
uTotal = z2(:,[]); % extract the whole input data
ysTotal = sim(M, uTotal, 'init', 'z');

% Compare the values of ys1, ys2 and ysTotal.
% ys1 should be equal to first half of ysTotal.
% ys2 should be equal to the second half of ysTotal
plot(ys1,'b', ys2, 'g', ysTotal, 'k*') % plot the three responses
```

MATLAB software responds with a plot showing the three responses, with `ysTotal` overlaying `ys1` and `ys2` to verify that they match.



Matching Model Response to Output Data

In this example, you estimate initial states of model `M` such that the response best matches the output in data set`z2`.

1 Load the sample data and create data object `z2`.

```
load iddata2;
z2 = z2(1:50);
```

2 Estimate `idnlarx` model from data.

```
M = nlarx(z2,[4 3 2],'wave');
```

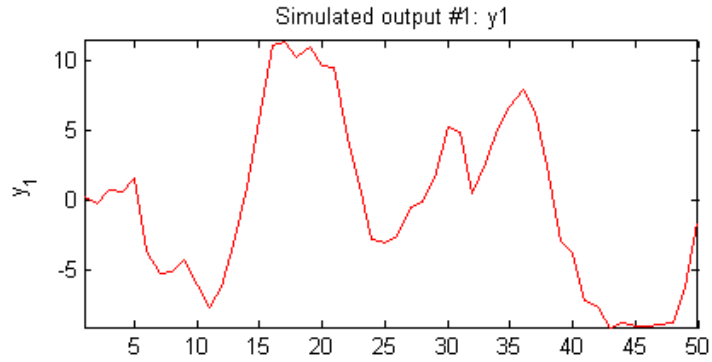
3 Estimate initial states of `M` to best fit `z2.y` in the simulated response.

```
x0 = findstates(M, z2, [], 'sim');
```

4 Simulate the model.

```
sim(M, z2.u, 'init', x0)
```

MATLAB software responds with the following plot.



Simulation About Steady-State Condition with Known Input and Unknown Output

In this example you start simulation of a model about a steady-state condition, where the input is known to be 1, but the output is unknown.

- Load sample data and create data object `z2`.

```
load iddata2
z2 = z2(1:50);
```

- Estimate `idnlarx` model from data.

```
M = nlarx(z2, [4 3 2], 'wave');
```

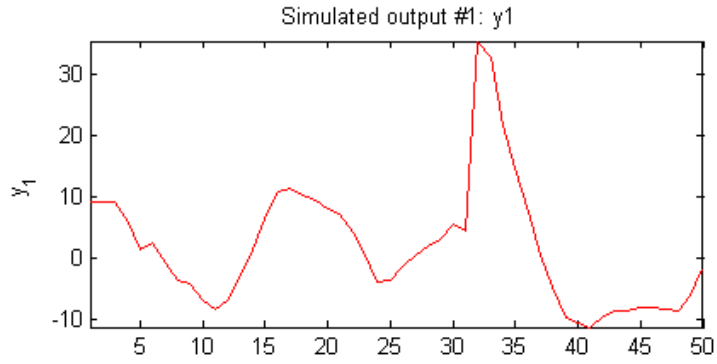
- Determine equilibrium state values for input 1 and the unknown target output.

```
x0 = findop(M, 'steady', 1, NaN);
```

- Simulate the model using initial states `x0`.

```
sim(M, z2.u, 'init', x0)
```

MATLAB software responds with the following plot.



See Also

`predict(idnlarx)`

`findop(idnlarx)`

`data2state(idnlarx)`

`finstates(idnlarx)`

Purpose

Simulate nonlinear ODE model

Syntax

```
YS = sim(NLSYS,DATA)
YS = sim(NLSYS,DATA,'Noise');
YS = sim(NLSYS,DATA,XOINIT);
YS = sim(NLSYS,DATA,'Noise',XOINIT);
YS = sim(NLSYS,DATA,'Noise','InitialState',XOINIT);
[YS, YSD, XFINAL] = sim(NLSYS,DATA,'Noise','InitialState',
   XOINIT);
```

Description

`YS = sim(NLSYS,DATA)` simulates the output of an `idnlgrey` model.

`YS = sim(NLSYS,DATA,'Noise');` simulates the model with Gaussian noise added to `YS`.

`YS = sim(NLSYS,DATA,XOINIT);` simulates the model with the specified initial states.

`YS = sim(NLSYS,DATA,'Noise',XOINIT);` performs simulation starting with the specified initial states and with Gaussian noise added.

`YS = sim(NLSYS,DATA,'Noise','InitialState',XOINIT);` simulates the model with the specified initial states.

`[YS, YSD, XFINAL] = sim(NLSYS,DATA,'Noise','InitialState',XOINIT);` performs simulation starting with the specified initial states and with Gaussian noise added, and returns the final states of the model after the simulation is completed.

To simulate the model with user-defined noise, set the input `DATA = [UIN E]`, where `UIN` is the input signal and `E` is the noise signal. `UIN` and `E` must both be one of the following:

- `iddata` objects: `E` stores the noise signals as inputs, where the number of inputs matches the number of model outputs.N_y is the number of model outputs.
- Matrices: `E` has as many columns as there are noise signals, corresponding to the number of model outputs.

Input

- `NLSYS`: `idnlgrey` model object.
- `DATA`: Input-noise data `[U E]`. If `E` is omitted and `'Noise'` is not given as an input argument, then a noise-free simulation is obtained. If `E` is omitted and `'Noise'` is given as an input argument, then Gaussian noise created by `randn(size(YS))*sqrtm(NLSYS.NoiseVariance)` will be added to `YS`. If both `E` and `'Noise'` are given, then `E` specifies the noise to add to `YS`. For time-continuous `idnlgrey` objects, `DATA` passed as a matrix will lead to that the data sample interval, `Ts`, is set to one.
- `XOINIT`: Initial state to use. It can take the following values:
  - `'zero'` : Zero initial state x(0) with all states fixed (nlsys.InitialStates.Fixed is thus ignored).
  - `'fixed'`: (or `NLSYS.InitialState`) Determines the values of the 'model' intial states, but all states are fixed (`NLSYS.InitialStates.Fixed` is thus ignored). This is the default value `XOINIT`.
  - vector/matrix: Column vector of initial state values. For multiple experiment `DATA`, `XOINIT` may be a matrix whose columns give different initial states for each experiment. All initial states are kept fixed (`nlsys.InitialStates.Fixed` is thus ignored).
  - struct array : `Nx`-by-1 structure array with fields:
    - `Name`: Name of the state (a string).
    - `Unit`: Unit of the state (a string).
    - `Value`: Value of the states (a finite real 1-by-`Ne` vector, where `Ne` is the number of experiments.)
    - `Minimum`: Minimum values of the states (a real 1-by-`Ne` vector or a real scalar, in which case all initial states have the same minimum value).

- `Maximum`: Maximum values of the states (a real 1-by-`Ne` vector or a real scalar, in which case all initial states have the same maximum value).
- `Fixed`: `true` (or a `true` 1-by-Ne vector). Any `false` value will be ignored. The initial state can also be specified through the property-value pair with name `'InitialState'` or `'X0'` followed by the value `XOINIT`.

Input

- `YS`: Simulated output. If `DATA` is an `iddata` object, then `YS` will also be an `iddata` object. Otherwise, `YS` will be a matrix where the `k`:th output is found in the `k`:th column of `YS`. If `DATA` is a multiple-experiment `iddata` object, then `YS` will be a multiple experiment object as well.
- `YSD`: Empty vector (`[].`) In the future, it will contain the estimated standard deviation of the simulated output.
- `XFINAL` Final states computed. In the single experiment case it is a column vector of length `Nx`. For multi-experiment data, `XFINAL` is an `Nx`-by-`Ne` matrix with the `i`:th column specifying the initial state of experiment `i`.

Note If `sim` is called without an output argument, MATLAB® software displays the simulated output(s) in a plot window.

Examples

In this example you simulate an `idnlgrey` model for a data set `z`. This example uses the `nlgr` model created in `idnlgreydemo2`.

1 Load the data set and create an `idnlgrey` model.

```
load twotankdata;
```

2 Estimate the `idnlgrey` model.

```
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 initial states.
Ts = 0;
nlgr = idnlgrey(FileName, Order, Parameters, ...
               InitialStates, Ts, ...
               'Name', 'Two tanks');
```

3 Set the properties of the model.

```
% Set properties of model nlgr
set(nlgr, 'InputName', 'Pump voltage', 'InputUnit', ...
          'V', 'OutputName', 'Lower tank water level', ...
          'OutputUnit', 'm', ...
          'OutputUnit', {'rad' 'rad/s'}, ...
           'TimeUnit', 's');
```

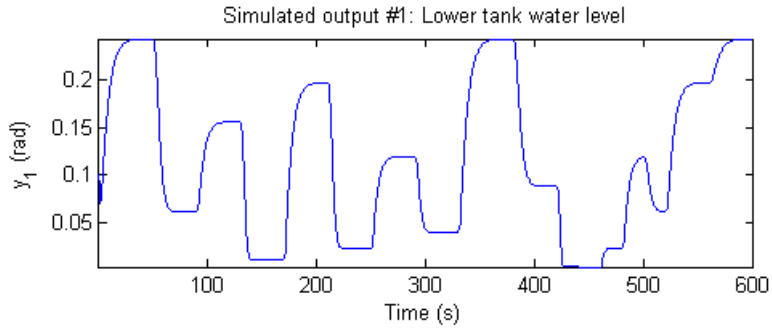
4 Create an `iddata` object `z`.

```
z = iddata(y, u, 0.2, 'Name', 'Two tanks');
```

5 Simulate the model with data set `z`.

```
sim(nlgr, z)
```

MATLAB software responds with the following plot.



See Also

`predict(idnlgrey)`

`findstates(idnlgrey)`

`pe`

`pem`

Purpose

Simulate Hammerstein-Wiener model

Syntax

```
YS = sim(MODEL,U)
YS = sim(MODEL,U,'Noise')
YS = sim(MODEL,U,'InitialState',INIT)
```

Description

`YS = sim(MODEL,U)` simulates the output of an `idnlhw` model.

`YS = sim(MODEL,U,'Noise')` simulates the model output with an additive Gaussian noise scaled according to the value of the `NoiseVariance` property of `MODEL`.

`YS = sim(MODEL,U,'InitialState',INIT)` specifies initial conditions for starting the simulation.

To simulate the model with user-defined noise, set the input `U = [UIN E]`, where `UIN` is the input signal and `E` is the noise signal. `UIN` and `E` must both be one of the following:

- `iddata` objects: `E` stores the noise signals as inputs, where the number of inputs matches the number of model outputs.N_y is the number of model outputs.
- Matrices: `E` has as many columns as there are noise signals, corresponding to the number of model outputs.

Input

- `MODEL`: `idnlhw` model object.
- `U`: Input data for simulation, which is an `iddata` object (where only the input channels are used) or a matrix. For noisy simulations, `U` contains both input and noise channels.
- `INIT`: Initial condition for simulation. `INIT` has one of the following values:
  - Vector of initial state values. To estimate an initial state vector from input-output data or to generate equilibrium states, see the `findstates(idnlhw)` and `findop(idnlhw)` reference pages. For multiple-experiment data, enter a matrix with the same number of columns as the number of experiments.

- `'z'`: (Default) Vector containing zeros and corresponding to a system starting from rest.

Input

- `YS`: Simulated output, which is an `iddata` object when `U` is an `iddata` object, or a matrix otherwise.

Note If `sim` is called without an output argument, MATLAB® software displays the simulated output(s) in a plot window.

Examples

Simulation Using Initial States to Best Fit Model Response to Measured Output

In this example you simulate the model output using initial states that minimize the error between the simulated and the measured output. `z2` is the measured data.

1. Load the sample data.

```
load iddata2
```

2. Create a Hammerstein-Wiener model.

```
M = nlhw(z2,[4 3 2],'wave','pwl');
```

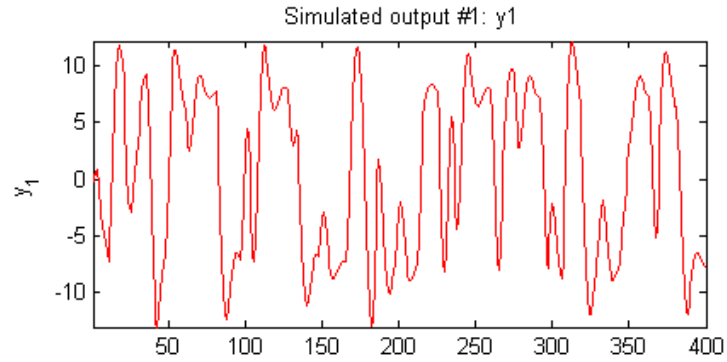
3. Compute the initial states that best fit the model response to the measured output.

```
x0 = findstates(M,z2);
```

4. Simulate the model using the estimated initial states.

```
sim(M,z2.u,'init',x0)
```

MATLAB software responds with the following plot.



Simulating a Hammerstein-Wiener Model at Steady-State with Known Input and Unknown Output

In this example, you simulate a single-input single-output `idnlhw` model about a steady-state operating point, where the input level is known to be 1 and the output level is unknown.

1 Load the sample data.

```
load iddata2
```

2 Create a Hammerstein-Weiner model.

```
M = nlhw(z2,[4 3 2],'wave','pwl');
```

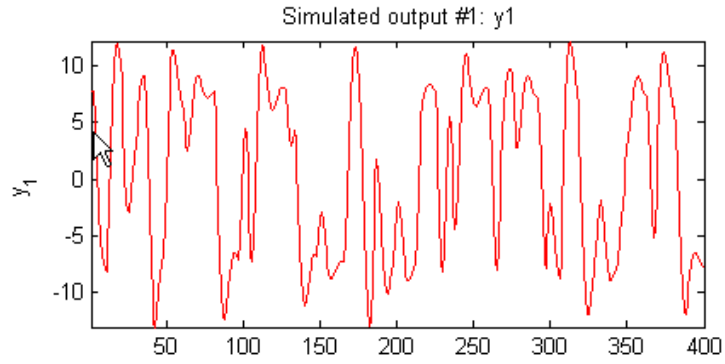
3 Compute steady-state operating point values corresponding to an input level of 1 and an unknown output level.

```
x0 = findop(M,'steady',1,NaN);
```

4 Simulate the model using the estimated initial states.

```
sim(M,z2.u,'init',x0)
```

MATLAB software responds with the following plot.



See Also

`predict(idnlhw)`

`findop(idnlhw)`

`finstates(idnlhw)`

Purpose

Simulate models with uncertainty using Monte Carlo method

Syntax

```
simsd(m,u)
simsd(m,u,N,'noise',Ky)
[y,ysd] = simsd(m,u)
```

Description

`u` is an `iddata` object containing the inputs. `m` is a model given as any `idmodel` object. `N` random models are created according to the covariance information given in `m`. The responses of each of these models to the input `u` are computed and graphed in the same diagram. If the argument `'noise'` is included, noise is added to the simulation in accordance with the noise model of `m` and its own uncertainty. `Ky` denotes the output numbers to be plotted. (The default is `all`).

The default value is `N=10`.

With output arguments

```
[y,ysd] = simsd(m,u)
```

No plots are produced, but `y` is returned as a cell array with the simulated outputs, and `ysd` is the estimated standard deviation of `y`, based on the `N` different simulations. If `u` is an `iddata` object, so are the contents of the cells of `y` and `ysd`; otherwise, they are returned as vectors/matrices. In the `iddata` case,

```
plot(y{:})
```

thus plots all the responses.

`sim` and `simsd` have similar syntaxes. Note that `simsd` computes the standard deviation by Monte Carlo simulation, while `sim` uses differential approximations (the Gauss approximation formula). They might give different results.

Examples

Plot the step response of the model `m` and evaluate how it varies in view of the model's uncertainty.

```
step1 = [zeros(5,1); ones(20,1)];
```

```
simsd(m,step1)
```

See Also

compare

idmdlsim

pe

predict

sim

Purpose

Dimensions of `iddata`, `idmodel`, and `idfrd` objects

Syntax

```
d = size(m)
[ny,nu,Npar,Nx] = size(model)
[N, ny, nu, Nexp] = size(data)
ny = size(data,2)
ny = size(data,'ny')
size(model)
size(idfrd_object)
```

Description

`size` describes the dimensions of `iddata`, `idmodel`, and `idfrd` objects.

iddata

For `iddata` objects, the sizes returned are, in this order,

- `N` = the length of the data record. For multiple-experiment data, `N` is a row vector with as many entries as there are experiments.
- `ny` = the number of output channels.
- `ny` = the number of input channels.
- `Ne` = the number of experiments.

To access just one of these sizes, use `size(data,k)` for the `k` `size(data,'N'),size(data,'ny')`, etc.

When called with one output argument, `d = size(data)` returns

- `d = [N ny nu]` if the number of experiments is `1`.
- `d = [sum(N) ny nu Ne]` if the number of experiments is `Ne > 1`.

idmodel

For `idmodel` objects the sizes returned are, in this order,

- `ny` = the number of output channels.
- `nu` = the number of input channels.

- `Npar` = the length of the `ParameterVector` (number of estimated parameters).
- `Nx` = the number of states for `idss` and `idgrey` models.

In this case the individual dimensions are obtained by `size(mod,2)`, `size(mod,'Npar')`, etc.

When `size` is called with one output argument, `d = size(mod)`, `d` is given by

```
[ny nu Npar]
```

idfrd

For `idfrd` models, the sizes returned are, in this order,

- `ny` = the number of output channels.
- `nu` = the number of input channels.
- `Nf` = the number of frequencies.
- `Ns` = the number of spectrum channels.

In this case the individual dimensions are obtained by `size(mod,2)`, `size(mod,'Nf')`, etc.

When `size` is called with one output argument, `d = size(fre)`, `d` is given by

```
[ny nu Nf Ns]
```

When `size` is called with no output arguments, in any of these cases, the information is displayed in the MATLAB® Command Window.

Purpose

Estimate frequency response and spectrum using spectral analysis returning `idfrd` object

Syntax

```
g = spa(data)
g = spa(data,M,w,maxsize)
[g,phi,spe] = spa(data)
```

Description

`spa` estimates the transfer function `g` and the noise spectrum Φ_v of the general linear model

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

where $\Phi_v(\omega)$ is the spectrum of $v(t)$.

`data` contains the output-input data as an `iddata` object. The data can be complex valued. `data` can be both time domain and frequency domain. `data` can also be an `idfrd` object.

`g` is returned as an `idfrd` object (see `idfrd`) with the estimate of $G(e^{i\omega})$ at the frequencies ω specified by row vector `w`. The default value of `w` is

```
w = [1:128]/128*pi/Ts
```

Here `Ts` is the sampling interval of `data`.

`g` also includes information about the spectrum estimate of $\Phi_v(\omega)$ at the same frequencies. Both outputs are returned with estimated covariances, included in `g`. See `idfrd`.

`M` is the length of the lag window used in the calculations. The default value is

```
M = min(30,length(data)/10)
```

Changing the value of `M` controls the frequency resolution of the estimate. The resolution corresponding to `M` is approximately π/`M` rad/sampling interval. The value of `M` exchanges bias for variance in the spectral estimate. As `M` is increased, the estimated functions show more detail, but are more corrupted by noise. The sharper peaks a true

frequency function has, the higher `M` it needs. See `etfe` as an alternative for narrowband signals and systems. The function `spafdr` allows the frequency resolution to depend on the frequency. See also “Identifying Frequency-Response Models”.

`maxsize` controls the memory-speed tradeoff (see `Algorithm Properties`).

For time series, where `data` contains no input channels, `g` is returned with the estimated output spectrum and its estimated standard deviation.

When `spa` is called with two or three output arguments,

- `g` is returned as an `idfrd` model with just the estimated frequency response from input to output and its uncertainty.
- `phi` is returned as an `idfrd` model, containing just the spectrum data for the output spectrum $\Phi_y(\omega)$ and its uncertainty.
- `spe` is returned as an `idfrd` model containing spectrum data for all output-input channels in `data`. That is, if `z = [data.OutputData, data.InputData]`, `spe` contains as spectrum data the matrix-valued power spectrum of `z`.

$$S = \sum_{m=-M}^{M} Ez(t+m)z(t)' \exp(-iWmT)win(m)$$

Here $win(m)$ is weight at lag `m` of an M-size Hamming window and W is the frequency value i rad/s. Note that `'` denotes complex-conjugate transpose.

The normalization of the spectrum differs from the normalization used by the Signal Processing Toolbox `spectrum` function. See“Understanding Spectrum Normalization” for a more precise definition.

Algorithm

The covariance function estimates are computed using `covf`. These are multiplied by a Hamming window of lag size `M` and then transformed using a Fourier transform. The relevant ratios and differences are then

formed. For the default frequencies, this is done using a fast Fourier transform, which is more efficient than for user-defined frequencies. For multivariable systems, a straightforward `for` loop is used.

Note that `M = Y` is in Table 6.1 of Ljung (1999). The standard deviations are computed as on pages 184 and 188 in Ljung (1999).

Examples

With default frequencies,

```
g = spa(z);
bode(g)
```

With logarithmically spaced frequencies,

```
w = logspace(-2,pi,128);
g= spa(z,[],w); % (empty matrix gives default)
bode(g,'sd',3)
bode(g('noise'),'sd',3) % noise spectrum with confidence
interval of 3 standard deviations.
```

See Also

`bode`

`etfe`

`ffplot`

`freqresp`

`idfrd`

`nyquist`

`spafdr`

Purpose

Estimate frequency response and spectrum using spectral analysis with frequency-dependent resolution returning `idfrd` object

Syntax

```
g = spafdr(data)
g = spafdr(data,Resol,w)
```

Description

`spafdr` estimates the transfer function `g` and the noise spectrum Φ_v of the general linear model

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

where $\Phi_v(\omega)$ is the spectrum of $v(t)$.

`data` contains the output-input data as an `iddata` object. The data can be complex valued, and either time or frequency domain. It can also be an `idfrd` object containing frequency-response data.

`g` is returned as an `idfrd` object (see `idfrd`) with the estimate of $G(e^{i\omega})$ at the frequencies ω specified by row vector `w`. `g` also includes information about the spectrum estimate of $\Phi_v(\omega)$ at the same frequencies. Both results are returned with estimated covariances, included in `g`. See `idfrd`. The normalization of the spectrum is the same as described under `spa`.

Frequencies

The frequency variable `w` is either specified as a row vector of frequencies, or as a cell array `{wmin,wmax}`. In the latter case the covered frequencies will be 50 logarithmically spaced points from `wmin` to `wmax`. You can change the number of points to `NP` by entering `{wmin,wmax,NP}`.

Omitting `w` or entering it as an empty matrix gives the default value, which is 100 logarithmically spaced frequencies between the smallest and largest frequency in data. For time-domain data, this means from `1/N*Ts` to `pi*Ts`, where `Ts` is the sampling interval of data and `N` is the number of data.

Resolution

The argument `Resol` defines the frequency resolution of the estimates. The resolution (measured in rad/s) is the size of the smallest detail in the frequency function and the spectrum that is resolved by the estimate. The resolution is a tradeoff between obtaining estimates with fine, reliable details, and suffering from spurious, random effects: The finer the resolution, the higher the variance in the estimate. `Resol` can be entered as a scalar (measured in rad/s), which defines the resolution over the whole frequency interval. It can also be entered as a row vector of the same length as `w`. Then `Resol(k)` is the local, frequency-dependent resolution around frequency `w(k)`.

The default value of `Resol`, obtained by omitting it or entering it as the empty matrix, is `Resol(k) = 2(w(k+1)-w(k))`, adjusted upwards, so that a reasonable estimate is guaranteed. In all cases, the resolution is returned in the variable `g.EstimationInfo.WindowSize`.

Algorithm

If the data is given in the time domain, it is first converted to the frequency domain. Then averages of `Y(w)Conj(U(w))` and `U(w)Conj(U(w))` are formed over the frequency ranges `w`, corresponding to the desired resolution around the frequency in question. The ratio of these averages is then formed for the frequency-function estimate, and corresponding expressions define the noise spectrum estimate.

See Also

`bode`

`etfe`

`ffplot`

`freqresp`

`idfrd`

`nyquist`

`spa`

Purpose

Convert `idmodel` objects to Control System Toolbox™ LTI models

Syntax

```
sys = ss(mod)
sys = ss(mod,'m')
```

Description

`mod` is any `idmodel` object: `idgrey`, `idarx`, `idpoly`, `idproc`, `idss`, or `idmodel`.

`sys` is returned as an `ss` LTI model object. The noise input channels in `mod` are treated as follows: consider a model `mod` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

Both measured input channels u and normalized noise input channels v in `mod` are input channels in `sys`. The noise input channels belong to the `InputGroup 'Noise'`, while the others belong to the `InputGroup 'Measured'`. The names of the noise input channels are `v@yname`, where `yname` is the name of the corresponding output channel. This means that the LTI object realizes the transfer function [G HL].

To transform only the measured input channels in `sys`, use

```
sys = ss(mod('m')) or sys = ss(mod,'m')
```

This gives a representation of G only.

For a time series (no measured input channels, nu = 0), the LTI representations in `ss` contains the transfer functions from the normalized noise sources v to the outputs, that is, HL. If the model `mod` has both measured and noise inputs, `sys = ss(mod('n'))` gives a representation of the additive noise.

In addition, the normal subreferencing can be used.

```
sys = ss(mod(1,[3 4]))
```

If you want to describe [*G H*] or *H* (unnormalized noise), from *e* to *y*, first use

```
mod = noisecnv(mod)
```

to convert the noise channels *e* to regular input channels. These channels are assigned the names `e@yname`.

See Also

`frd`

`tf`

`zpk`

Purpose

State-space matrices from `idmodel` object

Syntax

```
[A,B,C,D,K,X0] = ssdata(m)
[A,B,C,D,K,X0,dA,dB,dC,dD,dK,dX0] = ssdata(m)
```

Description

`m` is the model given as any `idmodel` object. `A`, `B`, `C`, `D`, `K`, and `X0` are the matrices in the state-space description

$$\tilde{x}(t) = Ax(t) + Bu(t) + Ke(t)$$

$$x(0) = x0$$

$$y(t) = Cx(t) + Dx(t) + e(t)$$

where $\tilde{x}(t)$ is $\dot{x}(t)$ or $x(t+Ts)$ depending on whether `m` is a continuous-time or discrete-time model.

`dA`, `dB`, `dC`, `dD`, `dK`, and `dX0` are the standard deviations of the state-space matrices.

If the underlying model itself is a state-space model, the matrices correspond to the same basis. If the underlying model is an input-output model, an observer canonical form representation is obtained.

For a time-series model (no measured input channels, u = `[]`), B and D are returned as the empty matrices.

Subreferencing models in the usual way (see `idmodel` properties) will give the state-space representation of the chosen channels. Notice in particular that

```
[A,B,C,D] = ssdata(m('m'))
```

gives the response from the measured inputs. This is a model without a disturbance description. Moreover,

```
[A,B,C,D,K] = ssdata(m('n'))
```

('n' as in "noise") gives the disturbance description, that is, a time-series description of the additive noise with no measured inputs, so that B = [] and D = [].

To obtain state-space descriptions that treat all input channels, both u and e, as measured inputs, first apply the conversion

```
m = noisecnv(m)
```

or

```
m = noisecnv(m,'norm')
```

where the latter case first normalizes e to v, where v has a unit covariance matrix. See the reference page for noisecnv.

Algorithm

The computation of the standard deviations in the input-output case assumes that an A polynomial is not used together with an F or D polynomial in the general polynomial equation (see "What Are Black-Box Polynomial Models?". For the computation of standard deviations in the case that the state-space parameters are complicated functions of the parameters, the Gauss approximation formula is used together with numerical derivatives. The step sizes for this differentiation are determined by nuderst.

See Also

idmodel

idss

nuderst

Purpose

Plot step response with confidence interval

Syntax

```
step(m)
step(data)
step(m,'sd',sd,Time)
step(data,'sd',sd,'PW',na,Time)
step(m1,m2,...,dat1, ...,mN,Time,'sd',sd)
step(m1,'PlotStyle1',m2,'PlotStyle2',...,dat1,'PlotStylek',...,mN,
'PlotStyleN',Time,'sd',sd)
[y,t,ysd] = step(m)
mod = step(data)
```

Description

`step` can be applied both to any `idmodel` or `idnlmodel` object and to `iddata` sets.

For a discrete-time `idmodel` `m`, the step response `y` and, when required, its estimated standard deviation `ysd`, are computed using `sim`. When called with output arguments, `y`, `ysd`, and the time vector `t` are returned. When `step` is called without output arguments, a plot of the step response is shown. If `sd` is given a value larger than zero, a confidence region around the response is drawn. It corresponds to the confidence of `sd` standard deviations. If the input argument list contains `'fill'`, this region is plotted as a filled area.

Setting the Time Interval

The start time `T1` and the end time `T2` can be specified by `Time = [T1 T2]`. If `T1` is not given, it is set to `-T2/4`. The negative time lags (the step is always assumed to occur at time `0`) show possible feedback effects in the data when the step is estimated directly from data. If `Time` is not specified, a default value is used.

Estimating the Step Response from data

For an `iddata` set data, `step(data)` estimates a high-order, noncausal FIR model after first having prefiltered the data so that the input is “as white as possible.” The step response of this FIR model and, when asked for, its confidence region, are then plotted. Note that it might not be possible always to deliver the demanded time interval in

this case, because of lack of excitation in the data. A warning is then issued. When called with an output argument, `step`, in the `iddata` case, returns this FIR model, stored as an `idarx` model. The order of the prewhitening filter can be specified as `na`. The default value is `na = 10`.

Several Models/Data Sets

Any number and any mixture of models and data sets can be used as input arguments. The responses are plotted with each input/output channel (as defined by the models and data sets `InputName` and `OutputName`) as a separate plot. Colors, line styles, and marks can be defined by `PlotStyle` values, as in

```
step(m1,'b-*',m2,'y--',m3,'g')
```

Noise Channels

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `m.NoiseVariance` = Λ. The model can also be described with a unit variance, normalized noise source v:

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

- `step(m)` plots the step response of the transfer function G.
- `step(m('n'))` plots the step response of the transfer function H (ny inputs and ny outputs).The input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- If `m` is a time series, that is, nu = `0`, `step(m)` plots the step response of the transfer function H.

- `step(noisecnv(m))` plots the step response of the transfer function [*G H*] (*nu+ny* inputs and *ny* outputs). The noise input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- `step(noisecnv(m,'norm'))` plots the step response of the transfer function [*G HL*] (*nu+ny* inputs and *ny* outputs). The noise input channels have names `v@yname`, where `yname` is the name of the corresponding output.

Arguments

If `step` is called with a single `idmodel` `m`, the output argument `y` is a 3-D array of dimension `Nt`-by-`ny`-by-`nu`. Here `Nt` is the length of the time vector `t`, `ny` is the number of output channels, and `nu` is the number of input channels. Thus `y(:,ky,ku)` is the response in the `ky`th output channel to a step in the `ku`th input channel. No plot is produced when output arguments are used.

`ysd` has the same dimensions as `y` and contains the standard deviations of `y`. This is normally computed using `sim`. However, when the model `m` contains an estimated delay (dead time) as in certain process models, the standard deviation is estimated with Monte Carlo techniques, using `simsd`.

If `step` is called with an output argument and a single data set in the input arguments, the output is returned as an `idarx` model`mod` containing the high-order FIR model, and its uncertainty. By calling `step` with `mod`, the responses can be displayed and returned without your having to redo the estimation.

Examples

`step(data,'sd',3)` estimates and plots the step response

```
mod = step(data)
step(mod,'sd',3)
```

See Also

`cra`

`impulse`

Purpose

Generate model structure matrices for single-input and single-output systems

Syntax

```
NN = struc(NA,NB,NK)
```

Description

`struc` returns in `NN` the set of model structures composed of all combinations of the orders and delays given in row vectors `NA`, `NB`, and `NK`. The format of `NN` is consistent with the input format used by `arxstruc` and `ivstruc`. The command is intended for single-input systems only.

Examples

The statement

```
NN = struc(1:2,1:2,4:5);
```

produces

```
NN =
     1     1     4
     1     1     5
     1     2     4
     1     2     5
     2     1     4
     2     1     5
     2     2     4
     2     2     5
```

See Also

`arxstruc`

`ivstruc`

`selstruc`

Purpose

Convert `idmodel` objects to transfer-function Control System Toolbox™ LTI models

Syntax

```
sys = tf(mod)
sys = tf(mod,'m')
```

Description

`mod` is any `idmodel` object: `idgrey`, `idarx`, `idpoly`, `idproc`, `idss`, or `idmodel`.

`sys` is returned as a transfer function `tf` LTI model object. The noise input channels in `mod` are treated as follows:

Consider a model `mod` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. `mod.NoiseVariance` = Λ. The model can also be described with a unit variance, normalized noise source v.

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

Both measured input channels u and normalized noise input channels v in `mod` are input channels in `sys`. The noise input channels belongs to the `InputGroup 'Noise'`, while the others belong to the `InputGroup 'Measured'`. The names of the noise input channels will be `v@yname`, where `yname` is the name of the corresponding output channel. This means that the LTI object realizes the transfer function [G HL].

To transform only the measured input channels in `mod`, use

```
sys = tf(mod('m')) or sys = tf(mod,'m')
```

This gives a representation of G only.

For a time series, (no measured input channels, *nu* = 0), the LTI representation contains the transfer functions from the normalized noise sources *v* to the outputs, that is, *HL*. If the model mod has both measured and noise inputs, sys = tf(mod('n')) gives a representation of the additive noise.

In addition, you can use normal subreferencing.

```
sys = tf(mod(1,[3 4]))
```

If you want to describe [*G H*] or *H* (unnormalized noise), from *e* to *y*, first use

```
mod = noisecnv(mod)
```

to convert the noise channels *e* to regular input channels. These channels are assigned the names e@yname.

See Also

frd

ss

zpk

tfdata

Purpose

Numerator and denominator of transfer function from `idmodel` object

Syntax

```
[num,den] = tfdata(m)
[num,den,sdnum,sdden] = tfdata(m)
[num,den,sdnum,sdden] = tfdata(m,'v')
```

Description

`m` is a model given as any `idmodel` object with `ny` output channels and `nu` input channels.

`num` is a cell array of dimension `ny`-by-`nu`. `num{ky,ku}` (note the curly brackets) contains the numerator of the transfer function from input `ku` to output `ky`. This numerator is a row vector whose interpretation is described below.

Similarly, `den` is an `ny`-by-`nu` cell array of the denominators.

`sdnum` and `sdden` have the same formats as `num` and `den`. They contain the standard deviations of the numerator and denominator coefficients.

If `m` is a SISO model, adding an extra input argument `'v'` (for vector) will return `num` and `den` as vectors rather than cell arrays.

The formats of `num` and `den` are the same as those used by the Signal Processing Toolbox™ and Control System Toolbox™ products, both for continuous-time and discrete-time models.

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `m.NoiseVariance` = Λ. The model can also be described with a unit variance, normalized noise source v:

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

- `tfdata(m)` returns the transfer function *G*.
- `tfdata(m('n'))` returns the transfer function *H* (*ny* inputs and *ny* outputs).
- If `m` is a time series, that is, *nu* = `0`, `tfdata(m)` returns the transfer function *H*.
- `tfdata(noisecnv(m))` returns the transfer function [*G H*] (*nu*+*ny* inputs and *ny* outputs).
- `tfdata(noisecnv(m,'norm'))` returns the transfer function [*G HL*] (*nu*+*ny* inputs and *ny* outputs).

Examples

For a continuous-time model,

```
num = [1 2]
den = [1 3 0]
```

corresponds to the transfer function

$$G(s) = \frac{s+2}{s^2+3s}$$

For a discrete-time model,

```
num = [2 4 0]
den = [1 2 3 5]
```

corresponds to the transfer function

$$H(z) = \frac{2z^2+4z}{z^3+2z^2+3z+5}$$

which is the same as

$$H(q) = \frac{2q^{-1}+4q^{-2}}{1+2q^{-1}+3q^{-2}+5q^{-3}}$$

Note that for discrete-time models, `idpoly` and `polydata` have a different interpretation of the numerator vector, in case it does not have the same length as the denominator vector. To avoid confusion, fill out with zeros to make numerator and denominator vectors the same length. Do this with `tfdata`.

See Also

`idpoly`

`noisecnv`

Purpose

Return date and time when object was created or last modified

Syntax

```
timestamp(obj)
ts = timestamp(obj)
```

Description

`obj` is any `idmodel`, `iddata`, or `idfrd` object. `timestamp` returns or displays a string with information about when the object was created and last modified.

treepartition

Purpose

Class representing binary-tree nonlinearity estimator for nonlinear ARX models

Syntax

```
t=treepartition(Property1,Value1,...PropertyN,ValueN)
t=treepartition('NumberOfUnits',N)
```

Description

`treepartition` is an object that stores the binary-tree nonlinear estimator for estimating nonlinear ARX models.

You can use the constructor to create the nonlinearity object, as follows:

`t=treepartition(Property1,Value1,...PropertyN,ValueN)` creates a binary tree nonlinearity estimator object specified by properties in "treepartition Properties" on page 2-413. The tree has the number of leaves equal to `2^(J+1)-1`, where `J` is the number of levels and set by the property `NumberOfUnits`. The default value of `NumberOfUnits` is computed automatically.

Note `NumberOfUnits` sets an upper limit on the actual number of leaves used by the estimator.

`t=treepartition('NumberOfUnits',N)` creates a binary tree nonlinearity estimator object with `N` terms in the binary tree expansion. When you estimate a model containing `t`, the value of `NumberOfUnits` in `t` is automatically changed to show the actual number of leaves used—which is the largest integer of the form `2^n-1` and less than or equal to `N`.

Use `evaluate(t,x)` to compute the value of the function defined by the `treepartition` object `t` at `x`. At this stage, an adaptive *pruning algorithm* is used to select an active partition `D_a(= D_a(x))` on the branch of tree partitions that contain `x`.

Remarks

Use `treepartition` to define a nonlinear function $y = F(x)$, where F is a piecewise-linear (affine) function of x, y is scalar, and x is a

1-*by*-`m` vector. F is a local linear mapping, where x-space partitioning is determined by a binary tree.

The binary-tree network function is based on the following function expansion:

$$F(x) = xL + [1, x]C_a + d$$

x belongs to the active partition D_a. D_k is a partition of x-space. L is 1-*by*-`m` vector.

C_k is a 1-*by*-`(m+1)` vector.

d is a scalar.

The active partition D_a is computed as an intersection of half-spaces by a binary tree, as follows:

1. Tree with `N` nodes and `J` levels is initialized.
2. Node at level `J` is a terminating leaf and a node at level `j<J` has two descendants at level `j+1`. The number of leaves in the tree is `N = 2^(J+1)-1`, which is determined by the `NumberOfUnits` property of the `treepartition` object.
3. Partition at node `r` is based on `[1,x]*B_r > 0` or `<= 0` (move to left or right descendant), where `B_r` is chosen to improve the stability of least-square computation on the partitions at the descendant nodes.
4. Compute at each node `r` the coefficients `C_r` of best linear approximation of unknown regression function on `D_r` using penalized least-squares algorithm.

treepartition Properties

You can include property-value pairs in the constructor to specify the object.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% List all property values
```

```
get(t)
% Get value of NumberOfUnits property
t.NumberOfUnits
```

You can also use the `set` function to set the value of particular properties. For example:

```
set(t, 'NumberOfUnits', 5)
```

The first argument to `set` must be the name of a MATLAB® variable.

Property Name	Description
`NumberOfUnits`	Integer specifies the number of nodes in the tree. Default=`'auto'` select the number of nodes from the data using the pruning algorithm. When you estimate a model containing a `treepartition` nonlinearity, the value of `NumberOfUnits` is automatically changed to show the actual number of leaves used—which is the largest integer of the form `2^n-1` and less than or equal to `N`.For example: `treepartition('NumberOfUnits',5)`

Property Name	Description
`Parameters`	Structure containing the following fields: • `RegressorMean`: 1-*by*-`m` vector containing the means of `x` in estimation data, `r`. • `RegressorMinMax`: `m`-*by*-2 matrix containing the maximum and minimum estimation-data regressor values. • `OutputOffset`: scalar `d`. • `LinearCoef`: `m`-*by*-1 vector L. • `SampleLength`: Length of estimation data. • `NoiseVariance`: Estimated variance of the noise in estimation data. • `NonlinearParameters`: A structure containing the following tree parameters: - `TreeLevelPntr`: `N`-*by*-1 vector containing the levels `j` of each node. - `AncestorDescendantPntr`: `N`-*by*-3 matrix, such that the entry `(k,1)` is the ancestor of node `k`, and entries `(k,2)` and `(k,3)` are the left and right descendants, respectively. - `LocalizingVectors`: `N`-*by*-`(m+1)` matrix, such that the `r`th row is `B_r`. - `LocalParVector`: `N`-*by*-`(m+1)` matrix, such that the `k`th row is `C_k`. - `LocalCovMatrix`: `N`-*by*-`((m+1)m/2)` matrix such that the `k`th row is the covariance matrix of `C_k`. `C_k` is reshaped as a row vector.

Property Name	Description
`Options`	Structure containing the following fields that affect the initial model: • `FinestCell`: Integer or string specifying the minimum number of data points in the smallest partition. Default: `'auto'`, which computes the value from the data. • `Threshold`: Threshold parameter used by the adaptive pruning algorithm. Smaller threshold values correspond to shorter is the branch that is terminated by the active partition `D_a`. Higher threshold value results in a longer branch. Default: `1.0`. • `Stabilizer`: Penalty parameter of the penalized least-squares algorithm used to compute local parameter vectors `C_k`. Higher stabilizer value improves stability, but may deteriorate the accuracy of the least-square estimate. Default: `1e-6`.

Examples

Use `treepartition` to specify the nonlinear estimator in nonlinear ARX models. For example:

```
m=nlarx(Data,Orders,treepartition('num',5));
```

The following commands provide an example examples of using advanced `treepartition` options:

```
% Define the treepartition object
t=treepartition('num',100);
% Set the Threshold, which is a field
% in the Options structure
t.Options.Threshold=2;
% Estimate the nonlinear ARX model
m=nlarx(Data,Orders,t);
```

See Also

`nlarx`

unitgain

Purpose

Specify absence of nonlinearities for specific input or output channels in Hammerstein-Wiener models

Syntax

```
unit=unitgain
```

Description

`unit=unitgain` instantiates an object that specifies an identity mapping $F(x)=x$ to exclude specific input and output channels from being affected by a nonlinearity in Hammerstein-Wiener models.

Use the `unitgain` object as an argument in the `nlhw` estimator to set the corresponding channel nonlinearity to unit gain.

For example, for a two-input and one-output model, to exclude the second input from being affected by a nonlinearity, you the following syntax:

```
m = nlhw(data,orders,['saturation''unitgain'],'deadzone')
```

In this case, the first input saturates and the output has an associated `deadzone` nonlinearity.

Remarks

Use the `unitgain` object to exclude specific input and output channels from being affected by a nonlinearity in Hammerstein-Wiener models.

`unitgain` is a linear function $y = F(x)$, where $F(x)=x$.

unitgain Properties

`unitgain` does not have properties.

Examples

For example, for a one-input and one-output model, to exclude the output from being affected by a nonlinearity, you the following syntax:

```
m = nlhw(Data,Orders,'saturation','unitgain')
```

In this case, the input saturates.

If nonlinearities are absent in input or output channels, you can replace `unitgain` with an empty matrix. For example, to specify a Wiener

model with a sigmoid nonlinearity at the output and a unit gain at the input, use the following command:

```
m = nlhw(Data,Orders,[],'sigmoid');
```

See Also

deadzone

nlhw

saturation

sigmoidnet

Purpose

Plot model characteristics using Control System Toolbox™ LTI Viewer GUI

Syntax

```
view(m)
view(m('n'))
view(m1,...,mN,Plottype)
view(m1,PlotStyle1,...,mN,PlotStyleN)
```

Description

`m` is the output-input data to be graphed, given as any `idfrd` or `idmodel` object. After appropriate model transformations, the Control System Toolbox LTI Viewer opens. This allows `bode`, `nyquist`, `impulse`, `step`, and `zero/poles` plots.

To compare several models `m1,...,mN`, use `view(m1,...,mN)`. With `PlotStyle`, the color, line style, and marker of each model can be specified.

```
view(m1,'y:*',m2,'b')
```

Adding `Plottype` as a last argument specifies the type of plot in which `view` is initialized. `Plottype` is any of `'impulse'`,`'step'`,`'bode'`,`'nyquist'`,`'nichols'`,`'sigma'`, or `'pzmap'`. It can also be given as a cell array containing any collection of these strings (up to 6) in which case a multiplot is shown.

`view` does not display confidence regions. For that, use `bode`, `nyquist`, `impulse`, `step`, and `pzmap`.

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + He$$

$$cov(e) = \Lambda = LL^T$$

where L is a lower triangular matrix. Note that `m.NoiseVariance` = Λ. The model can also be described with a unit variance, normalized noise source v:

$$y = Gu + HLv$$
$$cov(v) = I$$

- `view(m)` plots the characteristics of the transfer function *G*.
- `view(m('n'))` plots the characteristics of the transfer function *HL* (*ny* inputs and *ny* outputs). The input channels have names `v@yname`, where `yname` is the name of the corresponding output.
- If `m` is a time series, that is, *nu* = `0`, `view(m)` plots the characteristics of the transfer function *HL*.
- `view(noisecnv(m))` plots the characteristics of the transfer function [*G H*] (*nu*+*ny* inputs and *ny* outputs). The noise input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- `view(noisecnv(m,'norm'))` plots the characteristics of the transfer function [*G HL*] (*nu*+*ny* inputs and *ny* outputs). The noise input channels have names `v@yname`, where `yname` is the name of the corresponding output.

`view` does not give access to all of the features of `ltiview`. Use

```
ml = ss(m), ltiview(Plottype,ml,...)
```

to reach these options.

See Also

`bode`

`impulse`

`nyquist`

`pzmap`

`step`

Purpose

Class representing wavelet network nonlinearity estimator for nonlinear ARX and Hammerstein-Wiener models

Syntax

```
s=wavenet('NumberOfUnits',N)
s=wavenet(Property1,Value1,...PropertyN,ValueN)
```

Description

`wavenet` is an object that stores the wavelet network nonlinear estimator for estimating nonlinear ARX and Hammerstein-Wiener models.

You can use the constructor to create the nonlinearity object, as follows:

`s=wavenet('NumberOfUnits',N)` creates a wavelet nonlinearity estimator object with `N` terms in the wavelet expansion.

`s=wavenet(Property1,Value1,...PropertyN,ValueN)` creates a wavelet nonlinearity estimator object specified by properties in "wavenet Properties" on page 2-423.

Use `evaluate(s,x)` to compute the value of the function defined by the `wavenet` object `s` at `x`.

Remarks

Use `wavenet` to define a nonlinear function $y = F(x)$, where y is scalar and x is an `m`-dimensional row vector. The wavelet network function is based on the following function expansion:

$$\begin{aligned} F(x) = (x-r)PL + a_{s1} f\left(\left(b_{s1}(x-r)\right)Q - c_{s1}\right) + \ldots \\ + a_{sn} f\left(\left(b_{sn} s(x-r)\right)Q - c_{sn}\right) \\ + a_{w1} g\left(\left(b_{w1}(x-r)\right)Q - c_{w1}\right) + \ldots \\ + a_{wn} w g\left(\left(b_{wn} w(x-r)\right)Q - c_{wn}\right) + d \end{aligned}$$

where f is a scaling function and g is the wavelet function. P and Q are `m`-*by*-`p` and `m`-*by*-`q` projection matrices, respectively. The projection matrices P and Q are determined by principal component analysis of estimation data. Usually, `p=m`. If the components of x in the estimation data are linearly dependent, then `p<m`. The number of columns of Q, `q`,

corresponds to the number of components of `x` used in the scaling and wavelet function.

When used in a nonlinear ARX model, `q` is equal to the size of the `NonlinearRegressors` property of the `idnlarx` object. When used in a Hammerstein-Wiener model, `m=q=1` and Q is a scalar.

r is a 1-*by*-`m` vector and represents the mean value of the regressor vector computed from estimation data.

d, a_s, b_s, a_w, and b_w are scalars. Parameters with the s subscript are scaling parameters, and parameters with the w subscript are wavelet parameters.

L is `p`-*by*-1 vector.

c_s and c_w are 1-*by*-`q` vectors.

The scaling function f and the wavelet function g are both radial functions, as follows:

$$f(x) = e^{-0.5x'x}$$

$$g(x) = (\dim(x) - x'x)e^{-0.5x'x}$$

wavenet Properties

You can include property-value pairs in the constructor to specify the object.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% List all property values
get(w)
% Get value of NumberOfUnits property
w.NumberOfUnits
```

You can also use the `set` function to set the value of particular properties. For example:

```
h set(w, 'LinearTerm', 'on')
```

The first argument to `set` must be the name of a MATLAB® variable.

Property Name	Description
`NumberOfUnits`	Integer specifies the number of nonlinearity units in the expansion. Default=`'auto'`. For example: `wavenet('NumberOfUnits',5)`
`LinearTerm`	Can have the following values: • `'on'`—(Default) Estimates the vector L in the expansion. • `'off'`—Fixes the vector L to zero and omits the term $(x-r)PL$. For example: `wavenet(H,'LinearTerm','on')`

Property Name	Description
`Parameters`	Structure containing the parameters in the nonlinear expansion, as follows: • `RegressorMean`: 1-*by*-`m` vector containing the means of `x` in estimation data, `r`. • `NonLinearSubspace`: `m`-*by*-`q` matrix containing Q. • `LinearSubspace`: `m`-*by*-`p` matrix containing P. • `LinearCoef`: `p`-*by*-1 vector L. • `ScalingDilation`: `ns`-*by*-1 matrix containing the values `bs_k`. • `WaveletDilation`: `nw`-*by*-1 matrix containing the values `bw_k`. • `ScalingTranslation`: `ns`-*by*-`q` matrix containing the values `cs_k`. • `WaveletTranslation`: `nw`-*by*-`q` matrix containing the values `cw_k`. • `ScalingCoef`: `ns`-*by*-1 vector containing the values `as_k`. • `WaveletCoef`: `nw`-*by*-1 vector containing the values `aw_k`. • `OutputOffset`: scalar `d`.

Property Name	Description
`Options`	Structure containing the following fields that affect the initial model: • `FinestCell`: Integer or string specifying the minimum number of data points in the smallest cell. A *cell* is the area covered by the significantly nonzero portion of a wavelet. Default: `'auto'`, which computes the value from the data. • `MinCells`: Integer specifying the minimum number of cells in the partition. Default: `16`. • `MaxCells`: Integer specifying the maximum number of cells in the partition. Default: `128`. • `MaxLevels`: Integer specifying the maximum number of wavelet levels. Default: `10`. • `DilationStep`: Real scalar specifying the dilation step size. Default: `2`. • `TranslationStep`: Real scalar specifying the translation step size. Default: `1`.

Examples

Use `wavenet` to specify the nonlinear estimator in nonlinear ARX and Hammerstein-Wiener models. For example:

```
m=nlarx(Data,Orders,wavenet);
```

See Also

`nlarx`

`nlhw`

Purpose

Convert `idmodel` objects to Control System Toolbox™ state-space LTI models

Syntax

```
sys = zpk(mod)
sys = zpk(mod,'m')
```

Description

`mod` is any `idmodel` object: `idgrey`, `idarx`, `idpoly`, `idproc`, `idss`, or `idmodel`.

`sys` is returned as a `zpk` LTI model object. The noise input channels in `mod` are treated as follows: consider a model `mod` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `mod.NoiseVariance` = Λ. The model can also be described with a unit variance, normalized noise source v.

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

Both measured input channels u and normalized noise input channels v in `mod` are input channels in `sys`. The noise input channels belongs to the `InputGroup 'Noise'`, while the others belong to the `InputGroup 'Measured'`. The names of the noise input channels are given by `v@yname`, where `yname` is the name of the corresponding output channel. This means that the LTI object realizes the transfer function [G HL].

To transform only the measured input channels in `sys`, use

```
sys = zpk(mod('m')) or sys = zpk(mod,'m')
```

This gives a representation of G only.

For a time series, (no measured input channels, nu = `0`), the LTI representation contains the transfer functions from the normalized

noise sources v to the outputs, that is, HL. If the model `mod` has both measured and noise inputs, `sys = zpk(mod('n'))` gives a representation of the additive noise.

In addition, the normal subreferencing can be used.

```
sys = zpk(mod(1,[3 4]))
```

If you want to describe [GH] or H (unnormalized noise), from e to y, first use

```
mod = noisecnv(mod)
```

to convert the noise channels e to regular input channels. These channels are assigned have the names `e@yname`.

See Also

`frd`

`ss`

`tf`

Purpose

Zeros, poles, and gains of transfer function from idmodel object

Syntax

```
[z,p,k] = zpkdata(m)
[z,p,k,dz,dp,dk] = zpkdata(m)
[z,p,k,dz,dp,dk] = zpkdata(m,'v')
```

Description

m is a model given as any idmodel object with ny output channels and nu input channels.

z is a a cell array of dimension ny-by-nu. z{ky,ku} (note the curly brackets) contains the zeros of the transfer function from input ku to output ky. This is a column vector of possibly complex numbers.

Similarly, p is an ny-by-nu cell array containing the poles.

k is a ny-by-nu matrix whose ky-ku entry is the transfer function gain of the transfer function from input ku to output ky. Note that the transfer function gain is the value of the leading coefficient of the numerator when the leading coefficient of the denominator is normalized to 1. It thus differs from the static gain. The static gain can be retrieved as Ks = freqresp(m,0).

dz contains the covariance matrices of the zeros in the following way: dz is a ny-by-nu cell array. dz{ky,ku} contains the covariance information about the zeros of the transfer function from ku to ky. It is a 3-D array of dimension 2-by-2-by-Nz, where Nz is the number of zeros. dz{ky,ku}(:,:,kz) is the covariance matrix of the zero z{ky,ku}(kz), so that the 1-1 element is the variance of the real part, the 2-2 element is the variance of the imaginary part, and the 1-2 and 2-1 elements contain the covariance between the real and imaginary parts.

dp contains the covariance matrices of the poles in the same way.

dk is a matrix containing the variances of the elements of k.

If m is a SISO model, adding an extra input argument 'v' (for vector) returns z and p as vectors rather than cell arrays.

Note that the zeros and the poles are associated with the different channel combinations. To obtain the so-called transmission zeros, use tzero.

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `m.NoiseVariance` = Λ. The model can also be described with a unit variance, normalized noise source v.

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

Then,

- `zpkdata(m)` returns the zeros and poles of G.
- `zpkdata(m('n'))` returns the zeros and poles of H (ny inputs and ny outputs).
- If `m` is a time series, that is, nu = `0`, `zpkdata(m)` returns the zeros and poles of H.
- `zpkdata(noisecnv(m))` returns the zeros and poles of the transfer function [G H] (nu+ny inputs and ny outputs).
- `zpkdata(noisecnv(m,'norm'))` returns the zeros and poles of the transfer function [GHL] (nu+ny inputs and ny outputs).

The procedure handles both models in continuous and discrete time.

Note that you cannot rely on information about zeros and poles at the origin and at infinity for discrete-time models. (This is a somewhat confusing issue anyway.)

Algorithm

The poles and zeros are computed using `ss2zp`. The covariance information is computed using the Gauss approximation formula, using the parameter covariance matrix contained in `m`. When the transfer function depends on the parameters, numerical differentiation is

applied. The step sizes for the differentiation are determined in the M-file `nuderst`.

3

Block Reference

Preparing Data (p. 3-2)	Blocks for transferring data between System Identification Toolbox™ and Simulink® software
Identifying Linear Models (p. 3-3)	Blocks for estimating linear models during simulation in Simulink software
Simulating and Predicting Model Output (p. 3-4)	Blocks for simulating linear and nonlinear models in Simulink software

Preparing Data

IDDATA Sink	Export `iddata` object to MATLAB® workspace
IDDATA Source	Import `iddata` object from MATLAB workspace

Identifying Linear Models

AR Estimator	Estimate parameters of AR model from scalar time series in Simulink® software returning `idpoly` object
ARMAX Estimator	Estimate parameters of ARMAX model from SISO data in Simulink software returning `idpoly` object
ARX Estimator	Estimate parameters of ARX model from SISO data in Simulink software returning `idpoly` object
BJ Estimator	Estimate parameters of Box-Jenkins model from SISO data in Simulink software returning `idpoly` object
OE Estimator	Estimate parameters of Output-Error model from SISO data in Simulink software returning `idpoly` object
PEM Estimator	Estimate generic input-output polynomial model parameters from SISO data using iterative prediction-error minimization method

Simulating and Predicting Model Output

IDMODEL Model	Simulate `idmodel` object in Simulink® software
IDNLARX Model	Simulate nonlinear ARX model in Simulink software
IDNLGREY Model	Simulate nonlinear grey-box model in Simulink software
IDNLHW Model	Simulate Hammerstein-Wiener model in Simulink software

4

Blocks — Alphabetical List

AR Estimator

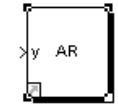
Purpose

Estimate parameters of AR model from scalar time series in Simulink® software returning `idpoly` object

Library

System Identification Toolbox

Description



AutoRegressive
model estimator

The AR Estimator block estimates the parameters of an AR model for a scalar time series and returns the model as an `idpoly` object. A time series is time-domain data consisting of one or more outputs *y(t)* and no corresponding measured input.

For information about the default algorithm settings used for model estimation, see the `Algorithm Properties` reference page.

Each estimation generates a figure with the following plots:

- Actual (measured) output versus the simulated or predicted model output.
- Error in simulated model, which is the difference between the measure output and the model output.

Model Definition

The AR model is defined, as follows:

$$y(t) + a_1 y(t-1) + \cdots + a_{n_a} y(t-n_a) = e(t)$$

where

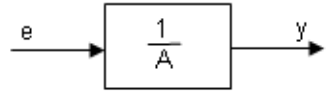
- *y(t)* is the output at time *t*.
- $a_1 \cdots a_n$ are the parameters to be estimated from the data.
- n_a is the number of poles of the system.
- $y(t-1) \ldots y(t-n_a)$ are the previous outputs on which the current output depends.
- *e(t)* is white-noise disturbance.

The AR model can be written compactly for a single output *y(t)* using the following notation:

$$A(q)y(t) = e(t)$$

where $A(q) = 1 + a_1 q^{-1} + \cdots + a_{n_a} q^{-n_a}$ and q^{-1} is the backward shift operator defined by $q^{-1}u(t) = u(t-1)$.

The following block diagram shows the AR model structure.



Input

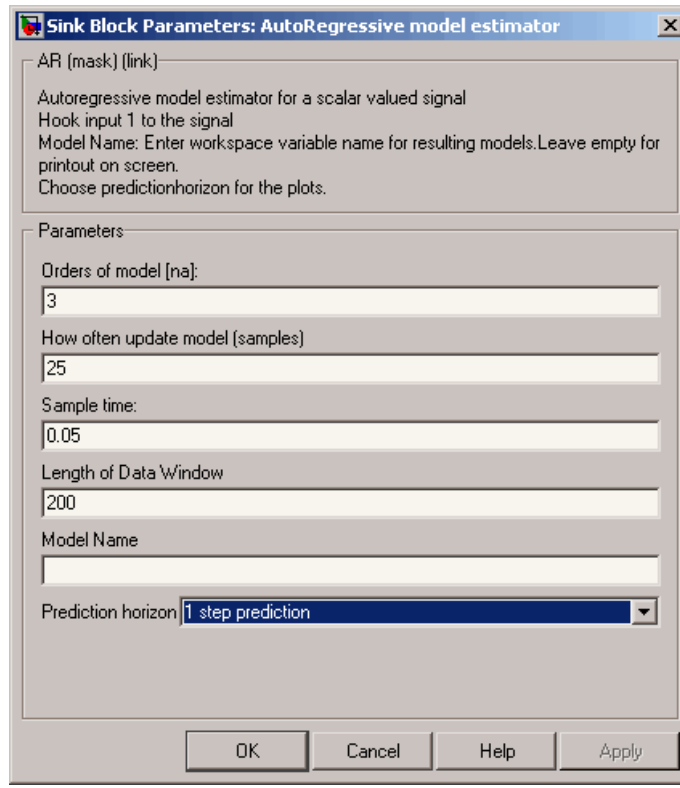
Time-series signal.

Output

The AR Estimator block outputs a sequence of multiple models (`idpoly` objects), estimated at regular intervals during the simulation. The **Data window** field in the block parameter dialog box specifies the number of data samples to use for estimation, as the simulation progresses.

The output format depends on whether you specify the **Model Name** in the block parameter dialog box.

Dialog Box



Orders of model [na]

Integer n_a corresponds to the number of a parameters (poles) in the AR model.

How often to update model (samples)

Number of input data samples that specify the interval after which to estimate a new model.

Default: 25

Sample time

Sampling time for the model.

Note If you use a fixed step-size solver, the fixed step size must be consistent with this sample time.

Length of Data Window

Number of past data samples used to estimate each model. A longer data window should be used for higher-order models. Too small a value might cause poor estimation results, and too large a value leads to slower computation.

Default: 200.

Model Name

Name of the model.

Whether you specify the model name determines the output format of the resulting models, as follows:

- If you do not specify a model name, the estimated models display in the MATLAB® Command Window in a transfer-function format.
- If you specify a model name, the resulting models are output to the MATLAB workspace as a cell array.

Prediction Horizon

Specifies the forward-prediction horizon for computing the response `K` steps in the future, where `K` is 1, 5, or 10.

Examples

This example shows how you can use the AR Estimator block in a Simulink model.

1 Generate sample input and output data.

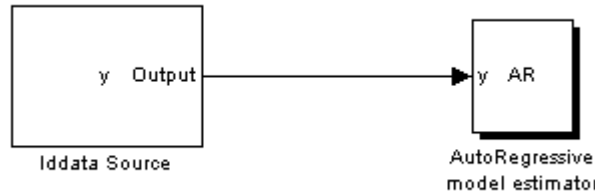
```
y = sin([1:300]') + 0.5*randn(300,1);
y = iddata(y);
```

2 Create a new Simulink model, as follows.

Add the IDDATA Source block and specify `y` in the **Iddata object** field of the IDDATA Source block parameter dialog box.

Add the AR Estimator block to the model and accept default block parameter values.

Connect the Output port of the IDDATA Source block to the y port of the AR Estimator block.



3 Run the estimation.

The estimated models display in the MATLAB Command Window every 25 samples. The following are the first two estimated transfer functions:

```
Transfer function:

num/den =

   0
   -
   1
Noise model:

num/den =

                          z^3
   -----------------------------------------
   z^3 - 0.98484 z^2 - 1.5613e-016 z + 0.015155
```

```
Transfer function:

num/den =

   0
   -
   1
Noise model:

num/den =

                          z^3
   -------------------------------------------
   z^3 - 0.98562 z^2 + 9.3675e-017 z + 0.014377
```

See Also

Related Commands

ar

idpoly

Topics in the System Identification Toolbox™ User's Guide

"Identifying Time-Series Models"

ARMAX Estimator

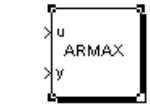
Purpose

Estimate parameters of ARMAX model from SISO data in Simulink® software returning `idpoly` object

Library

System Identification Toolbox

Description



AutoRegressive Moving Average with eXternal input model estimator

The ARMAX Estimator block estimates the parameters of a single-input and single-output ARMAX model and returns the model as an `idpoly` object.

For information about the default algorithm settings used for model estimation, see the `Algorithm Properties` reference page.

Each estimation generates a figure with the following plots:

- Actual (measured) output versus the simulated or predicted model output.
- Error in simulated model, which is the difference between the measure output and the model output.

Model Definition

The ARMAX model is defined, as follows:

$$y(t) + a_1 y(t-1) + \cdots + a_{n_a} y(t-n_a) =$$
$$b_1 u(t-1) + \cdots + b_{n_b} u(t-n_b) + e(t) + c_1 e(t-1) + \cdots + c_{n_c} e(t-n_c)$$

where

- *y(t)* is the output at time t.
- $a_1 \cdots a_n$, $b_1 \cdots b_n$, and $c_1 \cdots c_n$ are the parameters to be estimated.
- n_a is the number of poles of the system.
- $n_b - 1$ is the number of zeros of the system.
- n_c is the number of previous error terms on which the current output depends.

- n_k is the number of input samples that occur before the inputs affecting the current output.
- $y(t-1)\ldots y(t-n_a)$ are the previous outputs on which the current output depends.
- $u(t-n_k)\ldots u(t-n_k-n_b+1)$ are the previous inputs on which the current output depends.
- $e(t), e(t-1), \cdots e(t-n_c)$ are the white-noise disturbance values on which the current output depends.

The ARMAX model can also be written in a compact way using the following notation:

$$A(q)y(t) = B(q)u(t) + C(q)e(t)$$

where

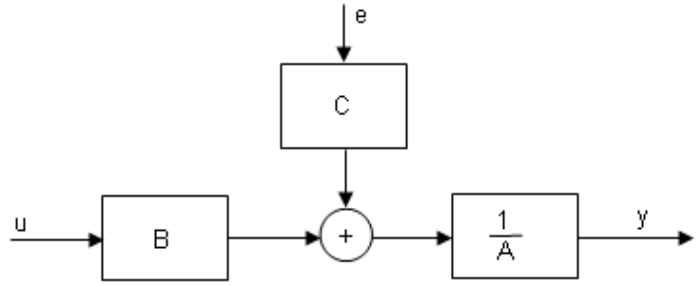
$$A(q) = 1 + a_1 q^{-1} + \cdots + a_{n_a} q^{-n_a}$$

$$B(q) = b_1 + b_2 q^{-1} + \cdots + b_{n_b} q^{-n_b+1}$$

$$C(q) = 1 + c_1 q^{-1} + \cdots a_{n_c} q^{-n_c}$$

and q^{-1} is the backward shift operator, defined by $q^{-1}u(t) = u(t-1)$.

The following block diagram shows the ARMAX model structure.



Input

The block accepts two inputs, corresponding to the measured input-output data for estimating the model.

First input: Input signal.

Second input: Output signal.

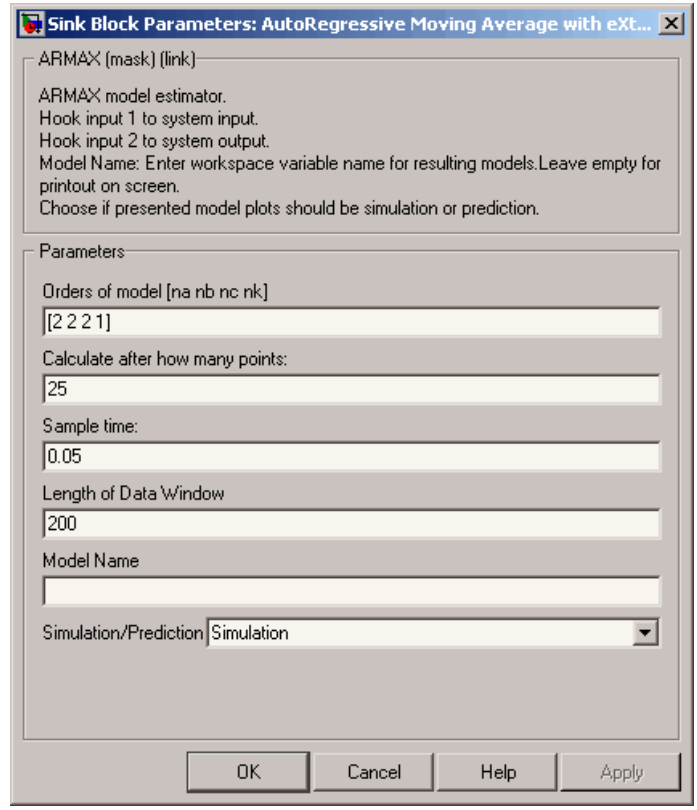
Output

The ARMAX Estimator block outputs a sequence of multiple models (`idpoly` objects), estimated at regular intervals during the simulation.

The **Data window** field in the block parameter dialog box specifies the number of data samples to use for estimation, as the simulation progresses.

The output format depends on whether you specify the **Model Name** in the block parameter dialog box.

Dialog Box



Orders of model [na nb nc nk]

Integers n_a, n_b, n_c, and n_k specify the number of A, B, and C model parameters and the input-output delay, respectively.

Calculate after how many points

Number of input data samples that specify the interval after which to estimate a new model.

Default: 25

Sample time

Sampling time for the model.

Note If you use a fixed step-size solver, the fixed step size must be consistent with this sample time.

Length of Data Window

Number of past data samples used to estimate each model. A longer data window should be used for higher-order models. Too small a value might cause poor estimation results, and too large a value leads to slower computation.

Default: 200.

Model Name

Name of the model.

Whether you specify the model name determines the output format of the resulting models, as follows:

- If you do not specify a model name, the estimated models display in the MATLAB® Command Window in a transfer-function format.
- If you specify a model name, the resulting models are output to the MATLAB workspace as a cell array.

Simulation/Prediction

`Simulation`: The algorithm uses only measured input data to simulate the response of the model.

`Prediction`: Specifies the forward-prediction horizon for computing the response `K` steps in the future, where `K` is 1, 5, or 10.

Examples

This example shows how you can use the ARMAX Estimator block in a Simulink model.

1 Generate sample input and output data.

```
y = sin([1:300]') + 0.5*randn(300,1);
```

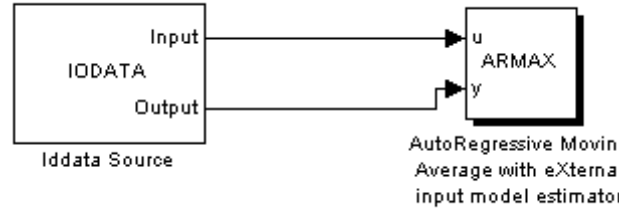
```
u = sin([1:300]') + 0.6*randn(300,1);
IODATA = iddata(y,u,0.25);
```

2 Create a new Simulink model, as follows.

Add the IDDATA Source block and specify `IODATA` in the **Iddata object** field of the IDDATA Source block parameters dialog box.

Add the ARMAX Estimator block to the model and accept default block parameter values.

Connect the Input and Output ports of the IDDATA Source block to the u and y ports of the ARMAX Estimator block, respectively.

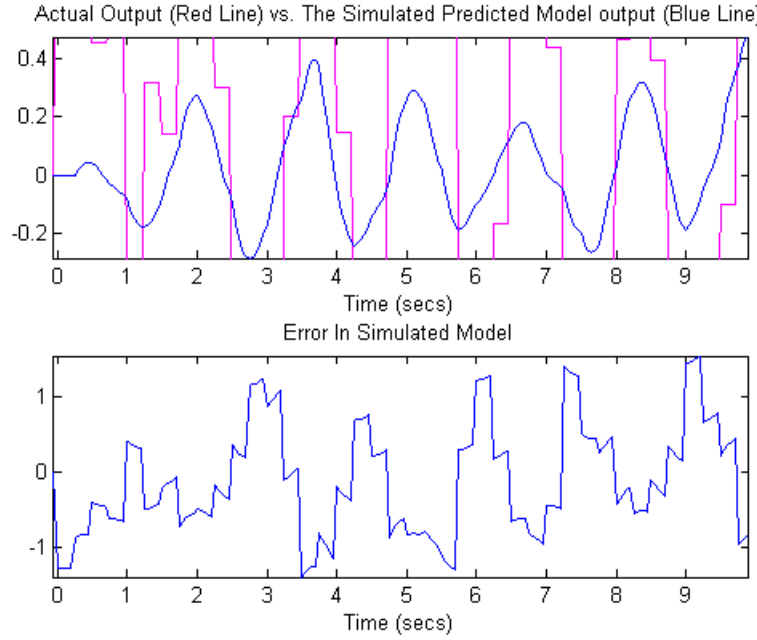


3 Run the simulation.

The estimated models display in the MATLAB Command Window every 25 samples. The following are the first two estimated transfer functions:

```
Transfer function:

num/den =

    -0.011418 z + 0.004052
   ------------------------
   z^2 - 1.9165 z + 0.91437
Noise model:

num/den =

   z^2 - 0.97202 z - 0.027981
   --------------------------
    z^2 - 1.9165 z + 0.91437
Transfer function:

num/den =

      0.027622 z + 0.042935
   -------------------------
   z^2 + 0.053184 z - 0.73983
Noise model:

num/den =

     z^2 + 1.002 z + 0.15928
   -------------------------
   z^2 + 0.053184 z - 0.73983
```

The simulation generates a plot of the measured output and the predicted output of the model.



See Also

Related Commands

`armax`

`idpoly`

Topics in the System Identification Toolbox™ User's Guide

"Identifying Input-Output Polynomial Models"

ARX Estimator

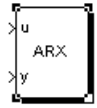
Purpose

Estimate parameters of ARX model from SISO data in Simulink® software returning `idpoly` object

Library

System Identification Toolbox

Description



AutoRegressive with eXternal input model estimator

The ARX block uses least-squares analysis to estimate the parameters of an ARX model and returns the estimated model as an `idpoly` object.

For information about the default algorithm settings used for model estimation, see the `Algorithm Properties` reference page.

Each estimation generates a figure with the following plots:

- Actual (measured) output versus the simulated or predicted model output.
- Error in simulated model, which is the difference between the measure output and the model output.

Model Definition

The ARX model is defined, as follows:

$$y(t) + a_1 y(t-1) + \cdots + a_{n_a} y(t-n_a) = b_1 u(t-1) + \cdots + b_{n_b} u(t-n_k-n_b+1) + e(t)$$

where

- *y(t)* is the output at time t.
- $a_1 \cdots a_n$ and $b_1 \cdots b_n$ are the parameters to be estimated.
- n_a is the number of poles of the system.
- $n_b - 1$ is the number of zeros of the system.
- n_k is the number of input samples that occur before the inputs that affect the current output.
- $y(t-1) \ldots y(t-n_a)$ are the previous outputs on which the current output depends.

- $u(t-n_k)\ldots u(t-n_k-n_b+1)$ are the previous inputs on which the current output depends.
- $e(t), e(t-1), \cdots e(t-n_c)$ is a white-noise disturbance value.

The ARX model can also be written in a compact way using the following notation:

$$A(q)y(t) = B(q)u(t-n_k) + e(t)$$

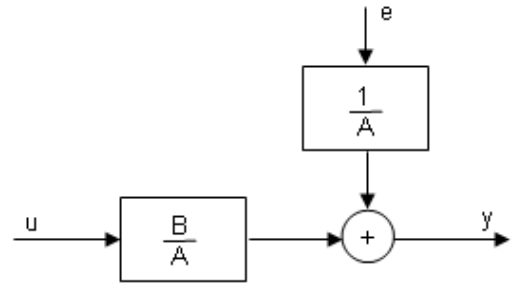
where

$$A(q) = 1 + a_1 q^{-1} + \cdots + a_{n_a} q^{-n_a}$$

$$B(q) = b_1 + b_2 q^{-1} + \cdots + b_{n_b} q^{-n_b+1}$$

and q^{-1} is the backward shift operator, defined by $q^{-1}u(t) = u(t-1)$.

The following block diagram shows the ARX model structure.



Input

The block accepts two inputs, corresponding to the measured input-output data for estimating the model.

First input: Input signal.

Second input: Output signal.

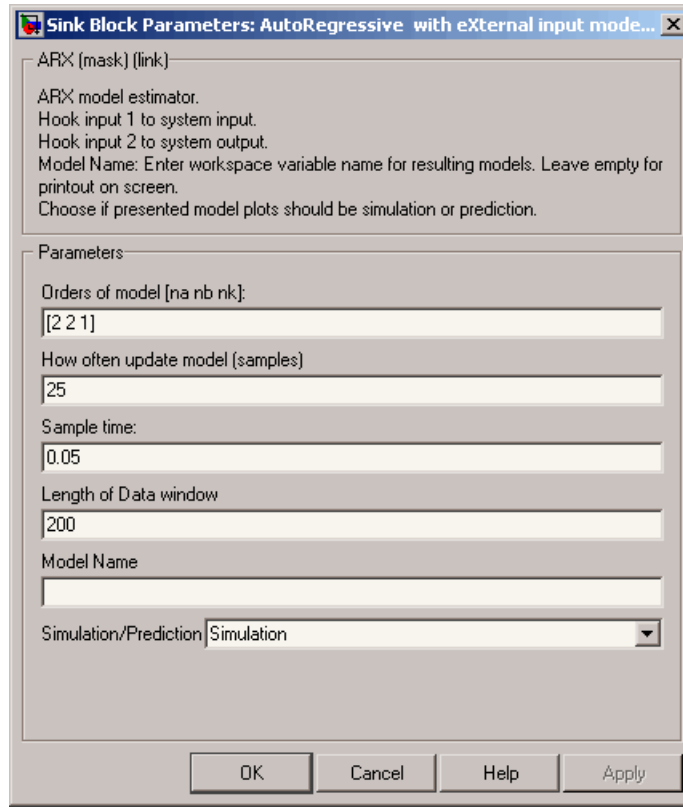
Output

The ARX Estimator block outputs a sequence of multiple models (`idpoly` objects), estimated at regular intervals during the simulation.

The **Data window** field in the block parameter dialog box specifies the number of data samples to use for estimation, as the simulation progresses.

The output format depends on whether you specify the **Model Name** in the block parameter dialog box.

Dialog Box



Orders of model [na nb nk]

Integers n_a, n_b, and n_k specify the number of A and B model parameters and the input-output delay, respectively.

How often update model [samples]

Number of input data samples that specify the interval after which to estimate a new model.

Default: 25

Sample time

Sampling time for the model.

Note If you use a fixed step-size solver, the fixed step size must be consistent with this sample time.

Length of Data window

Number of past data samples used to estimate each model. A longer data window should be used for higher-order models. Too small a value might cause poor estimation results, and too large a value leads to slower computation.

Default: 200.

Model Name

Name of the model.

Whether you specify the model name determines the output format of the resulting models, as follows:

- If you do not specify a model name, the estimated models display in the MATLAB® Command Window in a transfer-function format.
- If you specify a model name, the resulting models are output to the MATLAB workspace as a cell array.

Simulation/Prediction

`Simulation`: The algorithm uses only measured input data to simulate the response of the model.

`Prediction`: Specifies the forward-prediction horizon for computing the response `K` steps in the future, where `K` is 1, 5, or 10.

Examples

This example shows how you can use the ARX Estimator block in a Simulink model.

1 Generate sample input and output data.

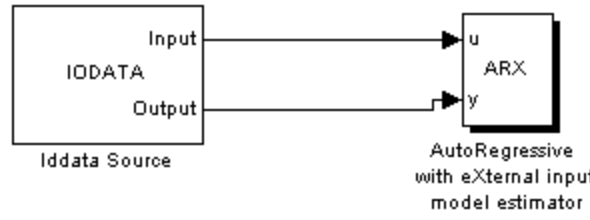
```
y = sin([1:300]') + 0.5*randn(300,1);
u = sin([1:300]') + 0.6*randn(300,1);
IODATA = iddata(y,u,0.25);
```

2 Create a new Simulink model, as follows.

Add the IDDATA Source block and specify `IODATA` in the **Iddata object** field of the IDDATA Source block parameters dialog box.

Add the ARX Estimator block to the model and accept default block parameter values.

Connect the Input and Output ports of the IDDATA Source block to the u and y ports of the ARX Estimator block, respectively.



3 Run the simulation.

The estimated models display in the MATLAB Command Window every 25 samples. The following are the first two estimated transfer functions:

```
Transfer function:

num/den =

     -0.0098875 z + 0.043902
   --------------------------
   z^2 - 0.94346 z + 0.078936
Noise model:

num/den =

               z^2
   --------------------------
   z^2 - 0.94346 z + 0.078936
Transfer function:

num/den =

     -0.001273 z + 0.010989
   --------------------------
   z^2 - 0.92699 z + 0.086175
Noise model:

num/den =

               z^2
   --------------------------
   z^2 - 0.92699 z + 0.086175
```

See Also

Related Commands

`arx`

`idpoly`

Topics in the System Identification Toolbox™ User's Guide

"Identifying Input-Output Polynomial Models"

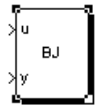
Purpose

Estimate parameters of Box-Jenkins model from SISO data in Simulink® software returning `idpoly` object

Library

System Identification Toolbox

Description



Box-Jenkins model estimator

The BJ Estimator block estimates the parameters of a Box-Jenkins model, and returns the estimated model as an `idpoly` object.

For information about the default algorithm settings used for model estimation, see the `Algorithm Properties` reference page.

Each estimation generates a figure with the following plots:

- Actual (measured) output versus the simulated or predicted model output.
- Error in simulated model, which is the difference between the measure output and the model output.

Model Definition

The Box-Jenkins model is defined, as follows:

$$y(t) = \frac{B(q)}{F(q)}u(t - n_k) + \frac{C(q)}{D(q)}e(t)$$

where the coefficients of

$$B(q) = b_1 + b_2 q^{-1} + \cdots + b_{n_b} q^{-n_b+1}$$

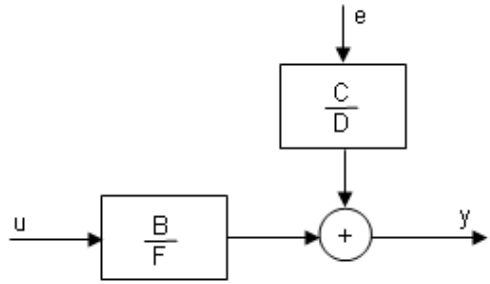
$$F(q) = 1 + f_1 q^{-1} + \cdots + f_{n_f} q^{-n_f}$$

$$C(q) = 1 + c_1 q^{-1} + \cdots + c_{n_c} q^{-n_c}$$

$$D(q) = 1 + d_1 q^{-1} + \cdots + d_{n_d} q^{-n_d}$$

are the parameters being estimated, and q^{-1} is the backward shift operator defined by $q^{-1}u(t) = u(t-1)$.

The following block diagram shows the Box-Jenkins model structure.



Input

The block accepts two inputs, corresponding to the measured input-output data for estimating the model.

First input: Input signal.

Second input: Output signal.

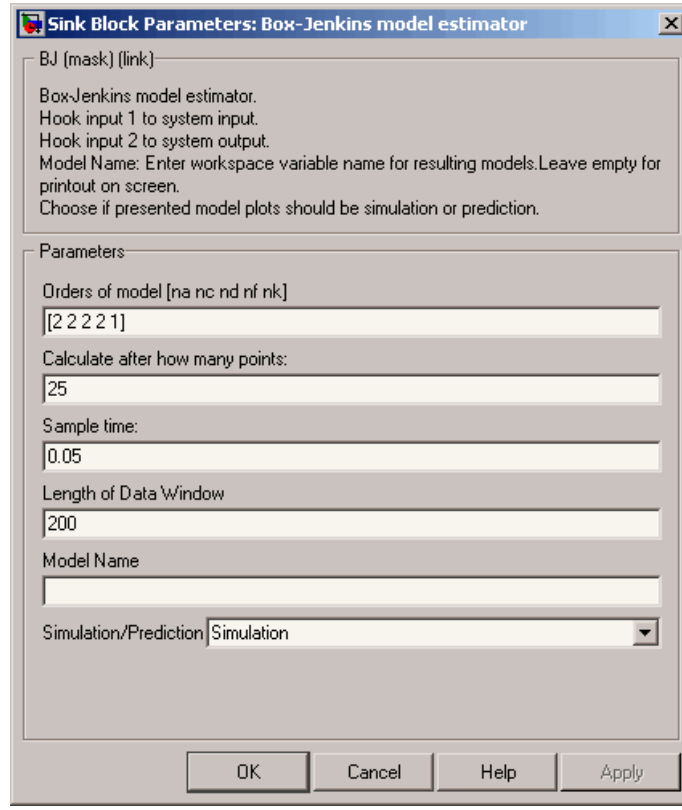
Output

The BJ Estimator block outputs a sequence of multiple models (`idpoly`), estimated at regular intervals during the simulation.

The **Data window** field in the block parameter dialog box specifies the number of data samples to use for estimation, as the simulation progresses.

The output format depends on whether you specify the **Model Name** in the block parameter dialog box.

Dialog Box



Orders of model [na nc nd nf nk]

Integers n_a, n_c, n_d, and n_f specify the number of A, C, D, and F model parameters, respectively.

Integer n_k specifies the input-output delay.

Calculate after how many points

Number of input data samples that specify the interval after which to estimate a new model.

Default: 25

Sample time
Sampling time for the model.

Note If you use a fixed step-size solver, the fixed step size must be consistent with this sample time.

Length of Data Window
Number of past data samples used to estimate each model. A longer data window should be used for higher-order models. Too small a value might cause poor estimation results, and too large a value leads to slower computation.

Default: 200.

Model Name
Name of the model.

Whether you specify the model name determines the output format of the resulting models, as follows:

- If you do not specify a model name, the estimated models display in the MATLAB® Command Window in a transfer-function format.
- If you specify a model name, the resulting models are output to the MATLAB workspace as a cell array.

Simulation/Prediction
`Simulation`: The algorithm uses only measured input data to simulate the response of the model.

`Prediction`: Specifies the forward-prediction horizon for computing the response `K` steps in the future, where `K` is 1, 5, or 10.

Examples

This example shows how you can use the BJ Estimator block in a Simulink model.

1 Generate sample input and output data.

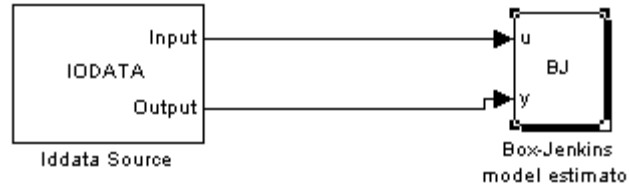
```
y = sin([1:300]') + 0.5*randn(300,1);
u = sin([1:300]') + 0.6*randn(300,1);
IODATA = iddata(y,u,0.25);
```

2 Create a new Simulink model, as follows.

Add the IDDATA Source block and specify `IODATA` in the **Iddata object** field of the IDDATA Source block parameters dialog box.

Add the BJ Estimator block to the model and accept default block parameter values.

Connect the Input and Output ports of the IDDATA Source block to the u and y ports of the BJ Estimator block, respectively.



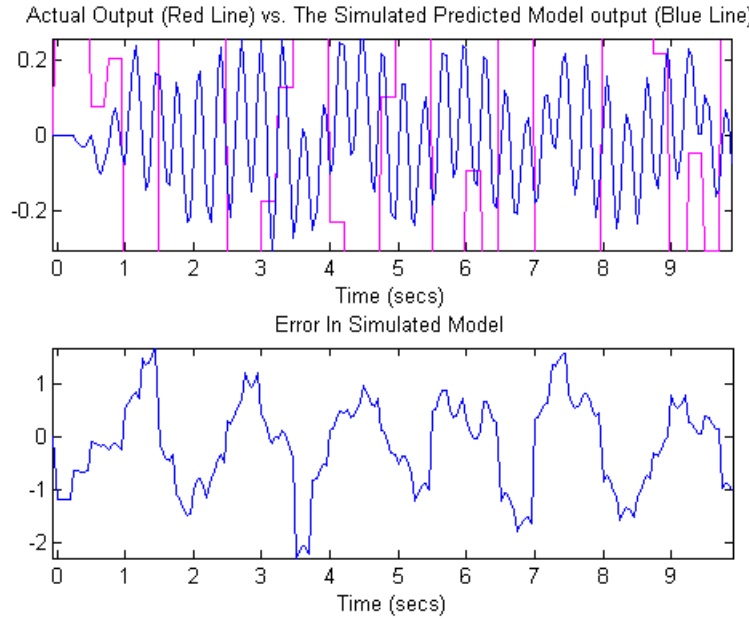
3 Run the simulation.

The estimated models display in the MATLAB Command Window every 25 samples. The following are the first two estimated transfer functions:

```
Transfer function:

num/den =

     0.047035 z - 0.041267
   ------------------------
   z^2 - 1.8992 z + 0.90897
```

```
Noise model:

num/den =

   z^2 - 0.99519 z - 0.004813
   --------------------------
    z^2 - 1.8992 z + 0.90897
Transfer function:

num/den =

     0.11524 z - 0.084277
   -----------------------
   z^2 - 1.8274 z + 0.87837
Noise model:

num/den =

    z^2 - 1.025 z + 0.13325
   -----------------------
   z^2 - 1.8274 z + 0.87837
```

The simulation generates a plot of the measured output and the predicted output of the model.



See Also

Related Commands

`bj`

`idpoly`

Topics in the System Identification Toolbox™ User's Guide

"Identifying Input-Output Polynomial Models"

IDDATA Sink

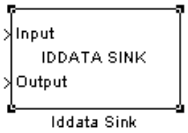
Purpose

Export `iddata` object to MATLAB® workspace

Library

System Identification Toolbox

Description



The IDDATA Sink block exports an `iddata` object to the MATLAB workspace.

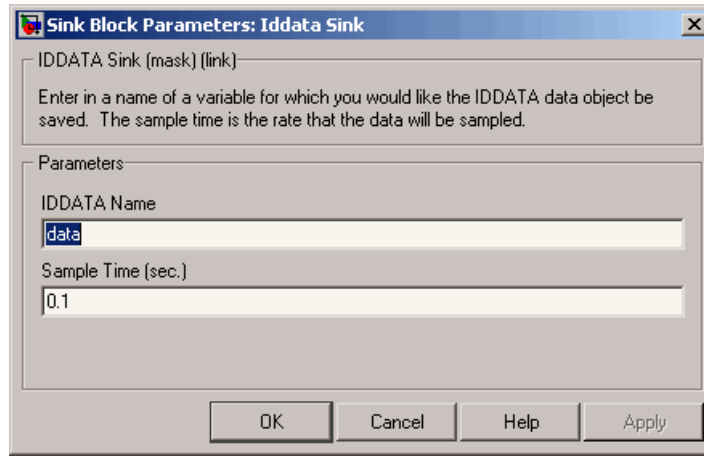
Input

The first block input is the input of specified `iddata` object in the MATLAB workspace. Similarly, the second block input is the output of the specified `iddata` object.

Output

None.

Dialog Box



IDDATA Name
Name of the `iddata` object in the MATLAB workspace.

Sample Time (sec.)
Time interval (in seconds) between successive data samples.

See Also

IDDATA Source

IDDATA Source

Purpose

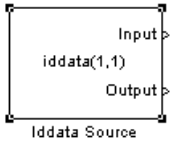
Import `iddata` object from MATLAB® workspace

Library

System Identification Toolbox

Description

The IDDATA Source block imports an `iddata` object from the MATLAB workspace.



Input

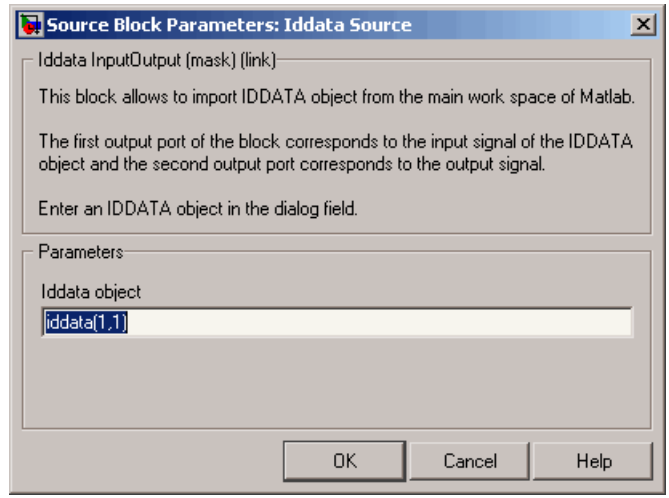
None.

Output

The first block output is the input signal of the `iddata` object imported from the MATLAB workspace.

The second block output is the output signal of this `iddata` object.

Dialog Box



Iddata object

Name of the `iddata` object exported to the MATLAB workspace.

The `iddata` object must contain only one experiment. For a multiple-experiment object, use `getexp(data,kexp)` to specify the experiment number `kexp`.

See Also

IDDATA Sink

IDMODEL Model

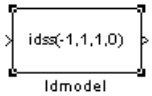
Purpose

Simulate `idmodel` object in Simulink® software

Library

System Identification Toolbox

Description



The IDMODEL block simulates a linear model in the MATLAB® workspace.

Note For simulating nonlinear models, use the IDNLGREY, IDNLARX, or IDNLHW Model blocks.

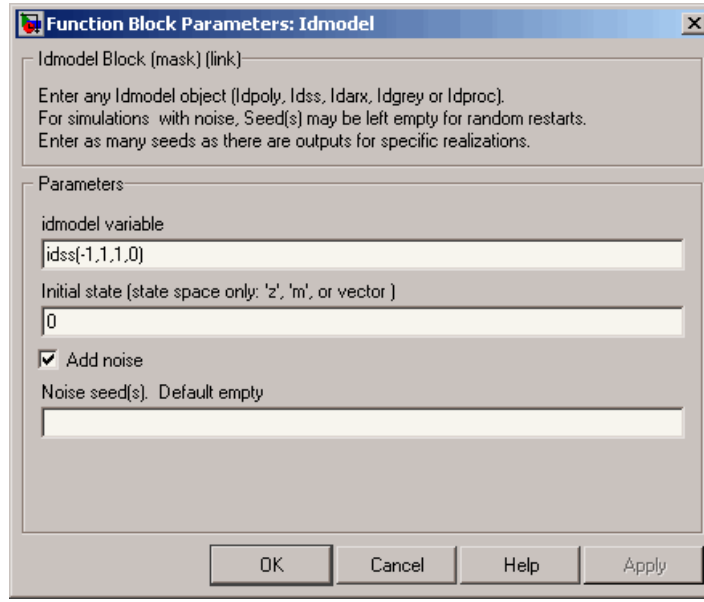
Input

Input signal to the model.

Output

Simulated output from the model.

Dialog Box



idmodel variable

Name of `idmodel` variable in the MATLAB workspace, which can be an `idpoly`, `idss`, `idgrey`, `idarx` or `idproc` model object that you estimated using System Identification Toolbox™ software.

This block supports both continuous-time and discrete-time models with or without input-output delays.

Initial state (state space only)

Initial state vector.

(Default) `'z'` (in quotation marks) specifies zero, which corresponds to a simulation starting from a state of rest.

Enter `'m'` to use the internal initial states of the model, stored by the `idss` or `idgrey` model property `X0`.

To specify your own initial states, enter one of the following:

- Enter known initial states for a state-space model directly, as a numerical vector.

  Note For non-state-space models, the initial conditions are `0`.

- Compute the initial states using the `findstates(idmodel)` command, which performs a simulation that provides the best fit to measured input-output data.

  For example, if the data set `z` and model `m` are already in the MATLAB workspace, you can enter `X0`, such that:

  ```
  X0 = findstates(m,z1)
  ```

Add noise

Select when you need to add noise, as specified by the model property `model.NoiseVariance` and the matrices or polynomials that determine the color of the additive noise.

For continuous-time models, the ideal variance of the noise term at any moment is infinite. In reality, you see a band-limited noise that takes into account the natural time constants of the system. You can interpret the resulting simulated output as filtered using a low-pass filter with a pass-band that does not distort the dynamics from the input.

Noise seed(s)

(Use when you select the **Add noise** check box.)

Enter an integer that specifies a seed that forces the simulation to add the same noise to the output every time you simulate the model. For more information about using seeds with functions that generate random number sequences, see the `rand` reference page in MATLAB documentation, for example.

See Also

`findstates(idmodel)`

`idmodel`

Purpose

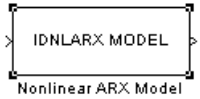
Simulate nonlinear ARX model in Simulink® software

Library

System Identification Toolbox

Description

The IDNLARX Model block simulates a nonlinear ARX (`idnlarx`) model for time-domain input and output data.



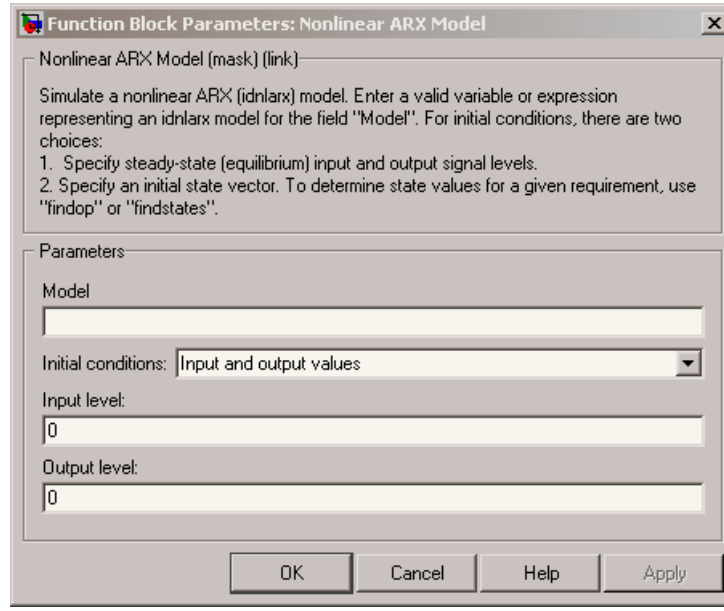
Input

Input signal to the model.

Output

Simulated output from the model.

Dialog Box



Model

Name of `idnlarx` variable in the MATLAB® workspace.

Initial conditions

Specifies the initial states as one of the following:

- `Input and output values`: Specify the input and output levels, as follows:

  - **Input level**

    If known, enter a vector of length equal to the number of model inputs. If you enter a scalar, it is the signal value for all inputs.

  - **Output level**

    If known, enter a vector of length equal to the number of model's outputs. If you enter a scalar, it is the signal value for all outputs.

- `State values`: When selected, you must specify a vector of length equal to the number of states in the model in the **Vector of state values** field.

  If you do not know the initial states, you can estimate these states, as follows:

  - To simulate about a given input level when you do not know the corresponding output level, you can estimate the equilibrium state values using the `findop(idnlarx)` command.

    For example, to simulate a model `M` about a steady-state condition where the input is `1` and the output is unknown, you can enter `X0`, such that:

    ```
    X0 = findop(M,'steady',1,NaN)
    ```

  - To estimate the initial states that provide a best fit between measured data and the simulated response of the model for the same input, use the `findstates(idnlarx)` command.

For example, to compute initial states such that the response of the model `M` matches the output data in the data set `z`, you can enter `X0`, such that:

```
X0 = findstates(M,z,[],'sim')
```

— To continue a simulation from a previous run, use the simulated input-output values from the previous simulation to compute the initial states `X0` for the current simulation.

For example, suppose that `firstSimData` is a variable that stores the input and output values from a previous simulation. For a model `M`, you can enter `X0`, such that:

```
X0 = data2state(M,firstSimData)
```

Examples

Example 1

In this example, you estimate a nonlinear ARX model from data and compare the model output of the model to the measured output of the system.

1 Load the sample data.

```
load twotankdata
```

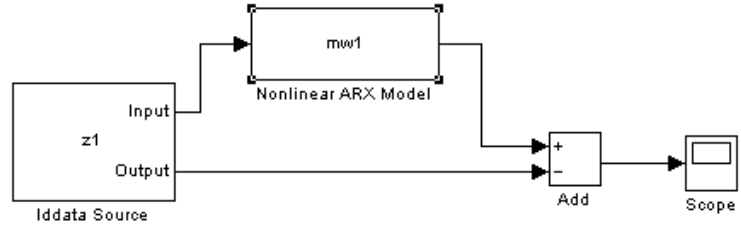
2 Create a data object from sample data.

```
z = iddata(y,u,0.2,'Name','Two tank system');
z1 = z(1:1000);
```

3 Estimate a nonlinear ARX model.

```
mw1 = nlarx(z1,[5 1 3],wavenet('NumberOfUnits',8));
```

4 Build the following Simulink model using the IDDATA Source, IDNLARX Model, and Scope blocks.



5 Double-click the IDDATA Source block and enter the following into the block parameter dialog box:

IDDATA Object: `z1`

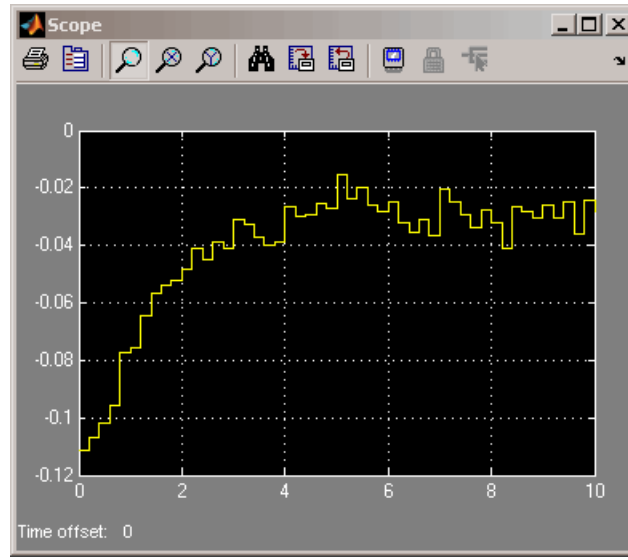
Click **OK**.

6 Double-click the IDNLARX Model block and enter the following into the block parameter dialog box:

- **Model**: `mw1`
- **Initial conditions**: Select `Input and output values` and accept the default values.

7 Run the simulation.

Click the Scope block to view the difference between the measured output and the model output. Use the **Autoscale** command to scale the axes.



In this example, the plot shows a difference between the measured and simulated responses at the start of the simulation. This difference may be reduced by estimating the initial states that best fit the measured output.

Example 2

In this example, you reduce the difference between the measured and simulated responses at the start of the simulation. To achieve this, you use the `findstates` command to estimate an initial state vector for the model from the data.

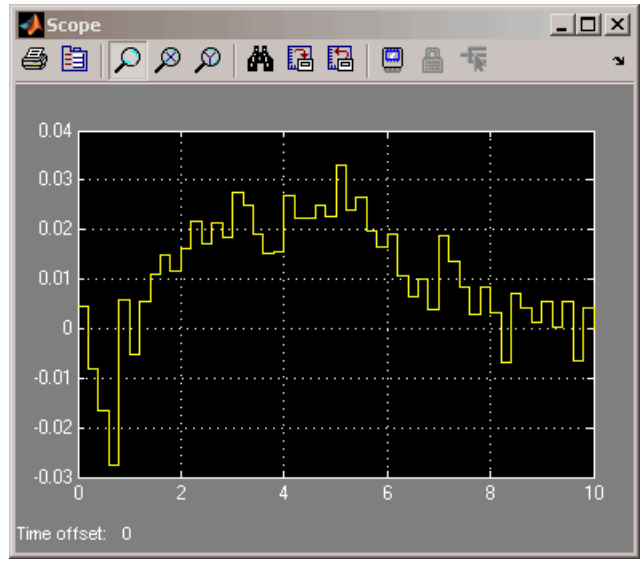
1. Estimate initial states from the data `z1`.

```
x0 = findstates(mw1,z1,[],'simulation');
```

2. Set the **Initial Conditions** to `State Values`. Enter `x0` in the corresponding field.

3. Run the simulation.

MATLAB software produces the following plot.



The difference between the measured and simulated responses at the beginning of the simulation is now reduced.

See Also

Related Commands

```
findop(idnlarx)
findstates(idnlarx)
idnlarx
```

Topics in the System Identification Toolbox™ User's Guide

"Identifying Nonlinear ARX Models"

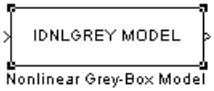
Purpose

Simulate nonlinear grey-box model in Simulink® software

Library

System Identification Toolbox

Description



Simulates systems of nonlinear grey-box (`idnlgrey`) models.

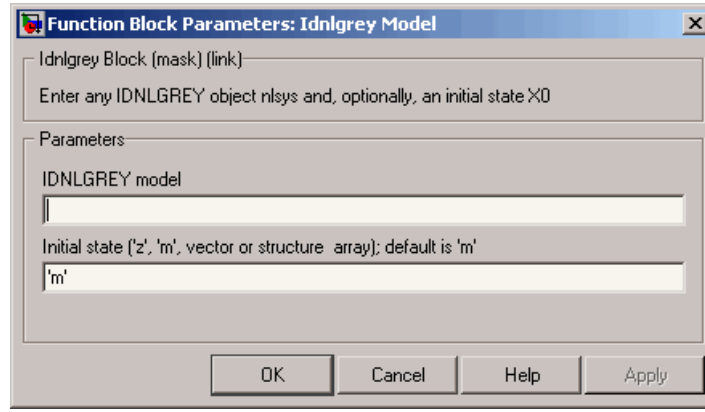
Input

Input signal to the model.

Output

Output signal from the model.

Dialog Box



IDNLGREY model

Name of `idnlgrey` variable in the MATLAB® workspace.

Initial state

Specify the initial states as one of the following:

- `'z'`: Specifies zero, which corresponds to a system starting from rest.
- 'm': Specifies the internal initial states of the model.
- Vector of size equal to the number of states in the `idnlgrey` object.
- An initial state structure array. For information about creating this structure, type `help idnlgrey/sim` in the MATLAB Command Window.

See Also

Related Commands

`idnlgrey`

Topics in the System Identification Toolbox™ User's Guide

"Estimating Nonlinear Grey-Box Models"

Purpose

Simulate Hammerstein-Wiener model in Simulink® software

Library

System Identification Toolbox

Description

The IDNLHW Model block simulates a Hammerstein-Wiener (`idnlhw`) model for time-domain input and output data.

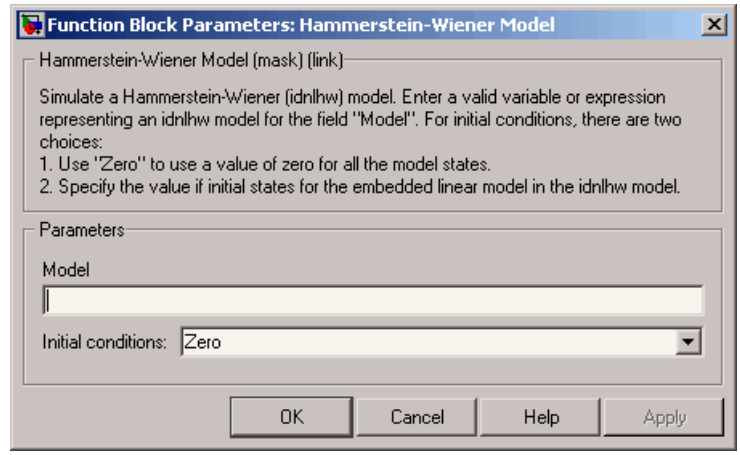
Input

Input signal to the model.

Output

Simulated output from the model.

Dialog Box



Model

Name of the `idnlhw` variable in the MATLAB® workspace.

Initial conditions

Specifies the initial states as one of the following:

- `Zero`: Specifies zero, which corresponds to a simulation starting from a state of rest.
- `State values`: When selected, you must specify a vector of length equal to the number of states in the model in the **Specify a vector of state values** field.

  If you do not know the initial states, you can estimate these states, as follows:

  — To simulate about a given input level when you do not know the corresponding steady-state output level, you can estimate the equilibrium state values using the `findop(idnlhw)` command.

  For example, to simulate a model `M` about a steady-state condition where the input is `1` and the output is unknown, you can enter `X0`, such that:

  ```
  X0 = findop(M,'steady',1,NaN)
  ```

  — To estimate the initial states such that the simulated response of the model matches specified output data for the same input, use the `findstates(idnlhw)`.

  For example, for the data set `z` and model `m`, you can enter `X0`, such that:

  ```
  X0 = findstates(m,z)
  ```

Examples

Example 1

In this example, you estimate a Hammerstein-Wiener model from data and compare the model output of the model to the measured output of the system.

1 Load the sample data.

```
load twotankdata
```

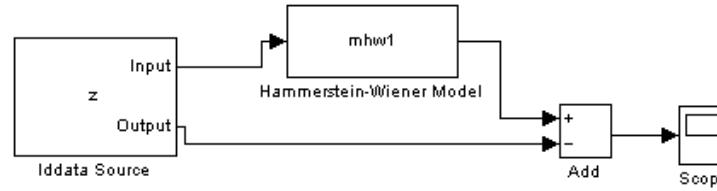
2 Create a data object from sample data.

```
z = iddata(y,u,0.2, ...
           'Name','Two tank system',...
           'Tstart',0);
```

3 Estimate a Hammerstein-Wiener model.

```
mhw1 = nlhw(z,[1 5 3],pwlinear,pwlinear);
```

4 Build the following Simulink model using the IDDATA Source, IDNLHW Model, and Scope blocks.



5 Double-click the IDDATA Source block and enter the following into the block parameter dialog box:

- **IDDATA Object**: z

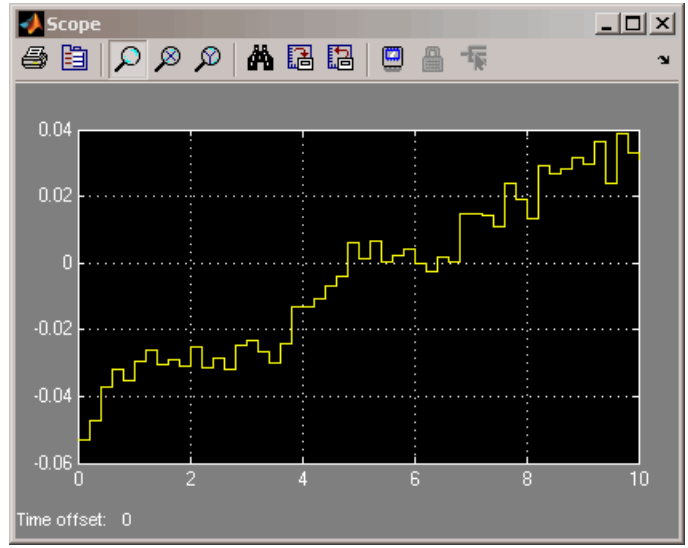
Click **OK**.

6 Double-click the IDNLHW Model block and enter the following into the block parameter dialog box:

- **Model**: mhw1
- **Initial Conditions**: Zero

7 Run the simulation.

Click the Scope block to view the difference between the measured output and the model output. Use the **Autoscale** toolbar button to scale the axes.



In this example, the plot shows a difference between the measured and simulated responses at the start of the simulation. This difference may be reduced by estimating the initial states that best fit the measured output.

Example 2

In this example, you reduce the difference between the measured and simulated responses at the start of the simulation. To achieve this, you use the `findstates` command to estimate an initial state vector for the model from the data.

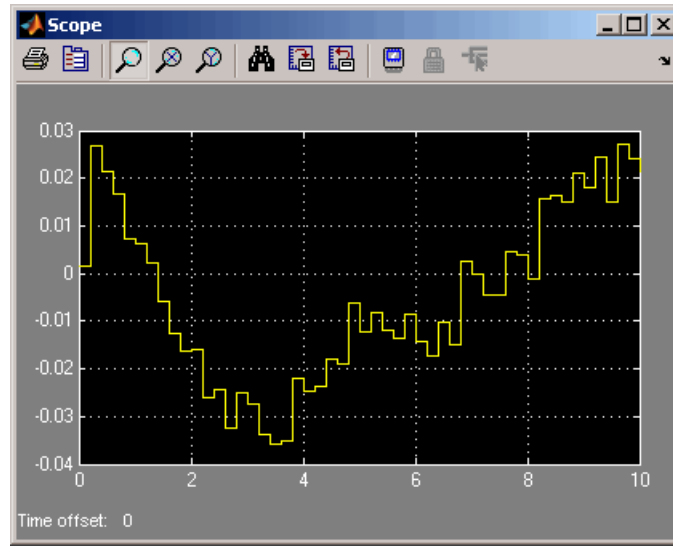
1. Estimate initial states from the data `z`:

   ```
   x0 = findstates(mhw1,z,[],'maxiter',50);
   ```

2. Set the **Initial Conditions** to `State Values`. Enter `x0` in the corresponding field.

3. Run the simulation.

MATLAB software produces the following plot.



The difference between the measured and simulated responses at the beginning of the simulation is now reduced.

See Also

Related Commands

`findop(idnlhw)`

`findstates(idnlhw)`

`idnlhw`

Topics in the System Identification Toolbox™ User's Guide

"Identifying Hammerstein-Wiener Models"

OE Estimator

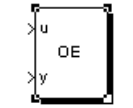
Purpose

Estimate parameters of Output-Error model from SISO data in Simulink® software returning `idpoly` object

Library

System Identification Toolbox

Description



Output-error model estimator.

The OE block estimates the parameters of an Output-Error model, and returns the estimated model as an `idpoly` object.

For information about the default algorithm settings used for model estimation, see the `Algorithm Properties` reference page.

Each estimation generates a figure with the following plots:

- Actual (measured) output versus the simulated or predicted model output.
- Error in simulated model, which is the difference between the measure output and the model output.

Model Definition

The output-error model is defined, as follows:

$$w(t) + f_1 w(t-1) + \cdots + f_{n_f} w(t-n_f) = b_1 u(t-1) + \cdots + b_{n_b} u(t-n_k-n_b+1)$$

$$y(t) = w(t) + e(t)$$

where

- w is the undisturbed output.
- $y(t)$ is the output at time t.
- $f_1 \cdots f_{n_f}$ and $b_1 \cdots b_{n_b}$ are the parameters to be estimated.
- n_f is the number of poles of the transfer function from the input to the undisturbed output.
- $n_b + 1$ is the number of zeros of the transfer function from the input to the undisturbed output.

- n_k is the number of input samples that occur before the inputs that affect the current output.
- $u(t-n_k)\ldots u(t-n_k-n_b+1)$ are the previous inputs on which the current output depends.
- *e(t)* is a white-noise disturbance value.

The OE model can also be written in a compact way using the following notation:

$$y(t)=\frac{B(q)}{F(q)}u(t-n_k)+e(t)$$

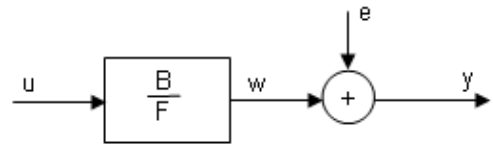
where

$$B(q)=b_1+b_2q^{-1}+\cdots+b_{n_b}q^{-n_b+1}$$

$$F(q)=1+f_1q^{-1}+\cdots f_{n_f}q^{-n_f}$$

and q^{-1} is the backward shift operator, defined by $q^{-1}u(t)=u(t-1)$.

The following block diagram shows the ARX model structure.



Input

The block accepts two inputs, corresponding to the measured input-output data for estimating the model.

First input: Input signal.

Second input: Output signal.

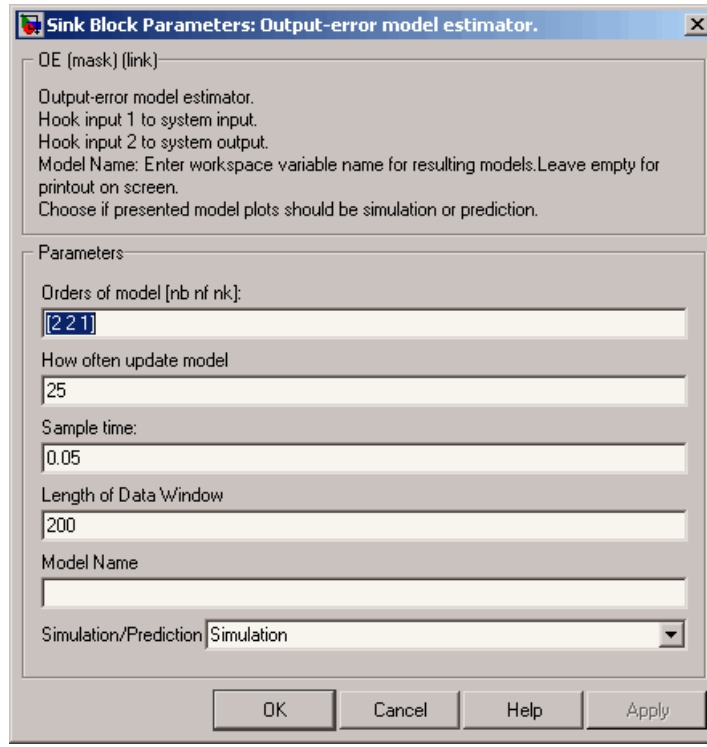
Output

The OE Estimator block outputs a sequence of multiple models (`idpoly`), estimated at regular intervals during the simulation.

The **Data window** field in the block parameter dialog box specifies the number of data samples to use for estimation, as the simulation progresses.

The output format depends on whether you specify the **Model Name** in the block parameter dialog box.

Dialog Box



Orders of model

Integers n_b, n_f, and n_k specify the number of B and F model parameters and the input-output delay, respectively.

How often update model
Number of input data samples that specify the interval after which to estimate a new model.

Default: 25

Sample time
Sampling time for the model.

Note If you use a fixed step-size solver, the fixed step size must be consistent with this sample time.

Length of Data Window
Number of past data samples used to estimate each model. A longer data window should be used for higher-order models. Too small a value might cause poor estimation results, and too large a value leads to slower computation.

Default: 200.

Model Name
Name of the model.

Whether you specify the model name determines the output format of the resulting models, as follows:

- If you do not specify a model name, the estimated models display in the MATLAB® Command Window in a transfer-function format.
- If you specify a model name, the resulting models are output to the MATLAB workspace as a cell array.

Simulation/Prediction
`Simulation`: The algorithm uses only measured input data to simulate the response of the model.

`Prediction`: Specifies the forward-prediction horizon for computing the response `K` steps in the future, where `K` is 1, 5, or 10.

Examples

This example shows how you can use the OE Estimator block in a Simulink model.

1 Generate sample input and output data.

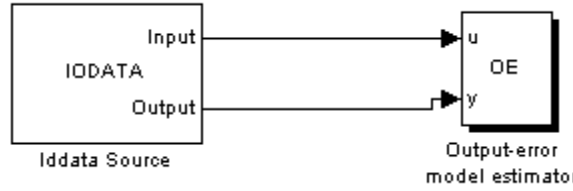
```
y = sin([1:300]') + 0.5*randn(300,1);
u = sin([1:300]') + 0.6*randn(300,1);
IODATA = iddata(y,u,0.25);
```

2 Create a new Simulink model, as follows.

Add the IDDATA Source block and specify `IODATA` in the **Iddata object** field of the IDDATA Source block parameters dialog box.

Add the OE Estimator block to the model and accept default block parameter values.

Connect the Input and Output ports of the IDDATA Source block to the u and y ports of the OE Estimator block, respectively.



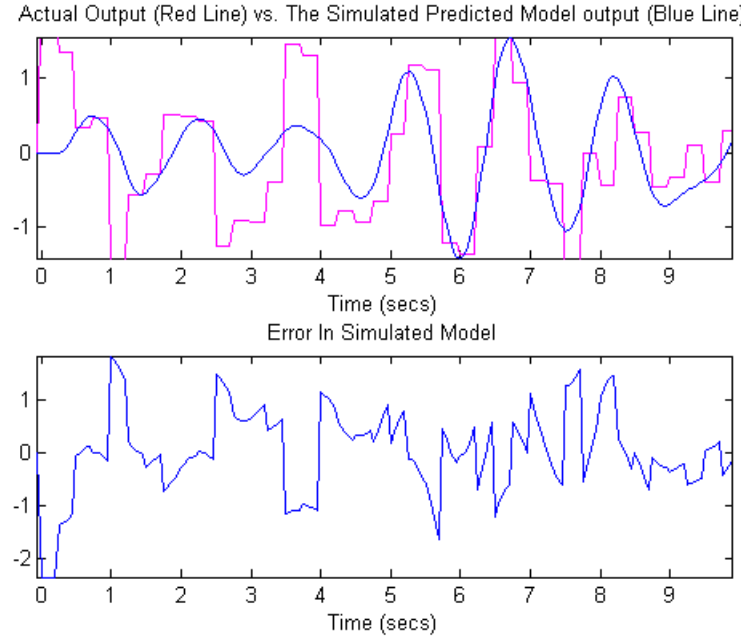
3 Run the simulation.

The estimated models display in the MATLAB Command Window every 25 samples. The following are the first two estimated models:

```
Transfer function:
```

```
num/den =

        1.2363 z - 1.3022
   --------------------------
   z^2 - 0.93997 z - 0.060004
Noise model:

num/den =

   1
   -
   1
Transfer function:

num/den =

      0.72731 z - 0.75957
   ------------------------
   z^2 - 1.1614 z + 0.22618
Noise model:

num/den =

   1
   -
   1
```

The simulation generates a plot of the measured output and the predicted output of the model.



See Also

Related Commands

`oe`

`idpoly`

Topics in the System Identification Toolbox™ User's Guide

"Identifying Input-Output Polynomial Models"

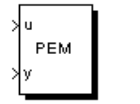
Purpose

Estimate generic input-output polynomial model parameters from SISO data using iterative prediction-error minimization method

Library

System Identification Toolbox

Description



General model estimator using Prediction Error Method

The PEM Estimator block estimates linear input-output polynomial models in Simulink® software.

For information about the default algorithm settings used for model estimation, see the `Algorithm Properties` reference page.

Each estimation generates a figure with the following plots:

- Actual (measured) output versus the simulated or predicted model output.
- Error in simulated model, which is the difference between the measure output and the model output.

Model Definition

The input-output polynomial structure is defined, as follows:

$$Ay(t) = \frac{B}{F}u(t - Nk) + \frac{C}{D}e(t)$$

where

- $y(t)$ is the output at time t.
- A, B, F, C, and D are the parameters $b_1 \cdots b_{n_b}$, $f_1 \cdots f_{n_f}$, $c_1 \cdots c_{n_c}$ and $d_1 \cdots d_{n_d}$ to be estimated.
- $e(t)$ is a white-noise disturbance.

Input

The block accepts two inputs, corresponding to the measured input-output data for estimating the model.

First input: Input signal.

Second input: Output signal.

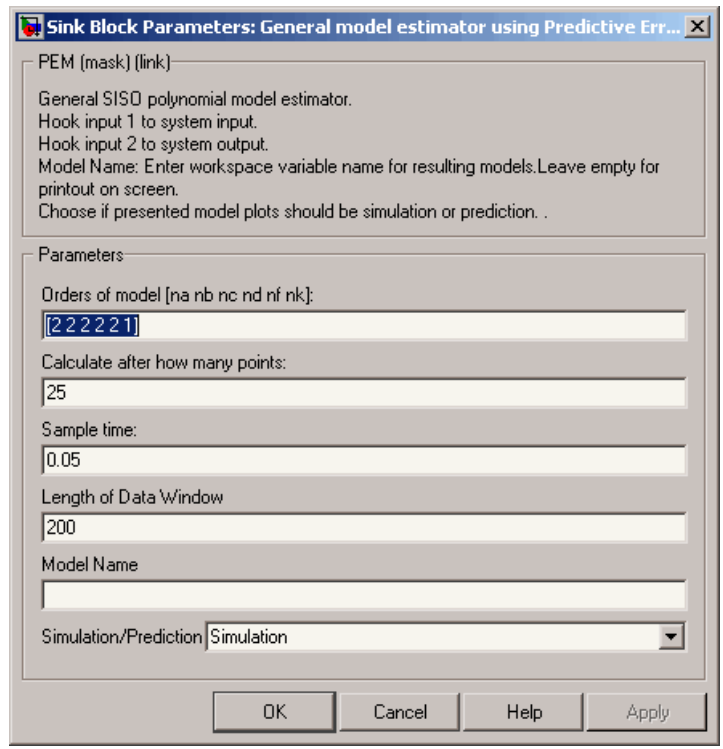
Output

The PEM Estimator block outputs a sequence of multiple models (`idpoly` objects), estimated at regular intervals during the simulation.

The **Data window** field in the block parameter dialog box specifies the number of data samples to use for estimation, as the simulation progresses.

The output format depends on whether you specify the **Model Name** in the block parameter dialog box.

Dialog Box



Orders of model [na nb nc nd nf nk]

Integers n_a, n_b, n_c, n_d, n_f, and n_k, specify the number of A, B, C, D, and F model parameters and the input-output delay, respectively.

Calculate after how many points

Number of input data samples that specify the interval after which to estimate a new model.

Default: 25

Sample time

Sampling time for the model.

Note If you use a fixed step-size solver, the fixed step size must be consistent with this sample time.

Length of Data Window

Number of past data samples used to estimate each model. A longer data window should be used for higher-order models. Too small a value might cause poor estimation results, and too large a value leads to slower computation.

Default: 200.

Model Name

Name of the model.

Whether you specify the model name determines the output format of the resulting models, as follows:

- If you do not specify a model name, the estimated models display in the MATLAB® Command Window in a transfer-function format.
- If you specify a model name, the resulting models are output to the MATLAB workspace as a cell array.

Simulation/Prediction

`Simulation`: The algorithm uses only measured input data to simulate the response of the model.

`Prediction`: Specifies the forward-prediction horizon for computing the response `K` steps in the future, where `K` is 1, 5, or 10.

Examples

This example shows how you can use the PEM Estimator block in a Simulink model.

1 Generate sample input and output data.

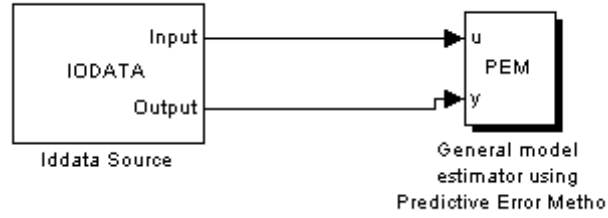
```
y = sin([1:300]') + 0.5*randn(300,1);
u = sin([1:300]') + 0.6*randn(300,1);
IODATA = iddata(y, u, 0.25);
```

2 Create a new Simulink model, as follows.

Add the IDDATA Source block and specify `IODATA` in the **Iddata object** field of the IDDATA Source block parameters dialog box.

Add the PEM Estimator blocks to the model and accept default block parameter values.

Connect the Input and Output ports of the IDDATA Source block to the u and y ports of the PEM Estimator block, respectively.

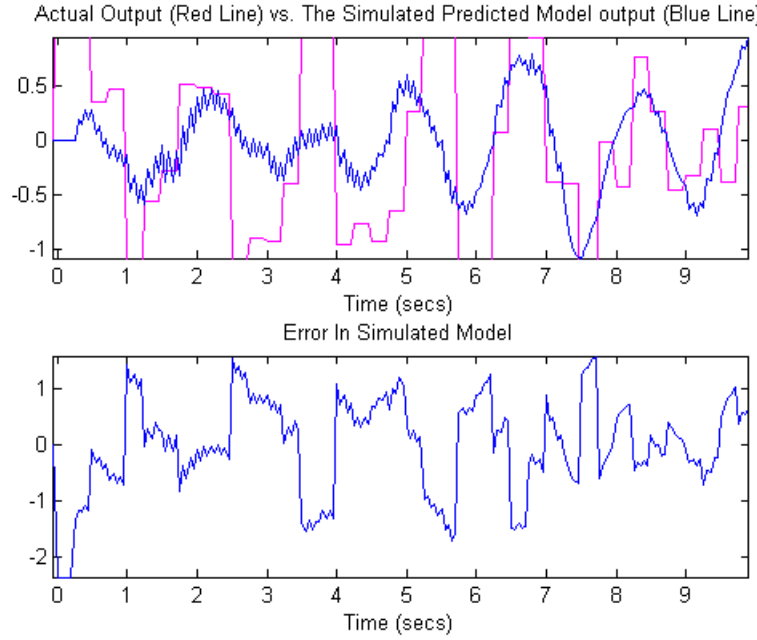


3 Run the simulation.

The estimated models display in the MATLAB Command Window every 25 samples. The following are the first two estimated transfer functions:

```
Transfer function:

num/den =

                -0.18207 z^3 - 0.24332 z^2
   ---------------------------------------------------
   z^4 - 0.84123 z^3 + 0.47495 z^2 - 1.0939 z + 0.66189
Noise model:

num/den =

              z^4 - 1.2223 z^3 + 0.40774 z^2
   ---------------------------------------------------
   z^4 - 2.1199 z^3 + 1.268 z^2 + 0.026365 z - 0.15755
Transfer function:

num/den =

                -0.01066 z^3 - 0.12172 z^2
   ---------------------------------------------------
   z^4 - 0.8102 z^3 + 0.41474 z^2 - 0.87876 z + 0.9228
Noise model:

num/den =

              z^4 - 1.6702 z^3 + 0.92533 z^2
   ---------------------------------------------------
   z^4 - 2.6141 z^3 + 2.5266 z^2 - 0.92744 z + 0.04472
```

The simulation generates a plot of the measured output and the predicted output of the model.



See Also

Related Commands

`idpoly`

`pem`

Topics in the System Identification Toolbox™ User's Guide

"Identifying Input-Output Polynomial Models"

Index

A

adaptive noise canceling 2-340
advanced algorithm options 2-16

E

estimating models
    `Focus` 2-11

F

fixed parameter 2-14
`Focus` 2-11

G

Gauss-Newton direction 2-16

L

Levenberg-Marquardt 2-16
`LimitError` 2-15

M

`MaxIter` 2-15
`MaxSize` 2-13

R

robust criterion
    `LimitError` 2-15

S

`SearchMethod` 2-16

T

`Tolerance` 2-15