

# System Identification Toolbox™

Reference

**R2012b**

*Lennart Ljung*

MATLAB®  
& SIMULINK®

## How to Contact MathWorks



[www.mathworks.com](http://www.mathworks.com) Web  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab) Newsgroup  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html) Technical Support



[suggest@mathworks.com](mailto:suggest@mathworks.com) Product enhancement suggestions  
[bugs@mathworks.com](mailto:bugs@mathworks.com) Bug reports  
[doc@mathworks.com](mailto:doc@mathworks.com) Documentation error reports  
[service@mathworks.com](mailto:service@mathworks.com) Order status, license renewals, passcodes  
[info@mathworks.com](mailto: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 1988–2012 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](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

### Patents

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://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)
October 2008	Online only	Revised for Version 7.2.1 (Release 2008b)
March 2009	Online only	Revised for Version 7.3 (Release 2009a)
September 2009	Online only	Revised for Version 7.3.1 (Release 2009b)
March 2010	Online only	Revised for Version 7.4 (Release 2010a)
September 2010	Online only	Revised for Version 7.4.1 (Release 2010b)
April 2011	Online only	Revised for Version 7.4.2 (Release 2011a)
September 2011	Online only	Revised for Version 7.4.3 (Release 2011b)
April 2012	Online only	Revised for Version 8.0 (Release 2012a)
September 2012	Online only	Revised for Version 8.1 (Release 2012b)



## Function Reference

**1**

---

Data Import and Processing .....	1-3
Linear Model Identification .....	1-5
Nonlinear Black-Box Model Identification .....	1-9
ODE Parameter Estimation .....	1-12
Recursive Model Identification .....	1-13
Model Analysis .....	1-14
Simulation and Prediction .....	1-19
System Identification Tool GUI .....	1-20

## Functions – Alphabetical List

**2**

## Block Reference

**3**

---

Data Import and Processing .....	3-2
Linear Model Identification .....	3-3

Simulation ..... 3-4

**Blocks — Alphabetical List**

**4**

**Index**

# Function Reference

---

Data Import and Processing (p. 1-3)	Represent, process, analyze, and manipulate data
Linear Model Identification (p. 1-5)	Estimate time response, frequency response, transfer function, input-output polynomial, and state-space models from time and frequency domain data
Nonlinear Black-Box Model Identification (p. 1-9)	Estimate nonlinear ARX and Hammerstein-Wiener models
ODE Parameter Estimation (p. 1-12)	Estimate parameters of linear and nonlinear ordinary differential or difference equations (grey-box models)
Recursive Model Identification (p. 1-13)	Recursively estimate input-output linear models, such as AR, ARX, ARMAX, Box-Jenkins, and Output-Error models
Model Analysis (p. 1-14)	Validate and analyze models by comparing model output, computing parameter confidence intervals and prediction errors, and getting advice on estimated models

Simulation and Prediction (p. 1-19)

Simulate and predict linear and nonlinear model output, and estimate initial states

System Identification Tool GUI  
(p. 1-20)

Start System Identification Toolbox™ GUI and customize preferences



## Data Import and Processing

<code>absorbDelay</code>	Replace time delays by poles at $z = 0$ or phase shift
<code>advice</code>	Analysis and recommendations for data or estimated linear models
<code>detrend</code>	Subtract offset or trend from data signals
<code>diff</code>	Difference signals in <code>iddata</code> objects
<code>feedback</code>	Identify possible feedback data
<code>getexp</code>	Specific experiments from multiple-experiment data set
<code>getTrend</code>	Data offset and trend information
<code>iddata</code>	Time- or frequency-domain data
<code>idfilt</code>	Filter data using user-defined passbands, general filters, or Butterworth filters
<code>idfrd</code>	Frequency-response data or model
<code>idresamp</code>	Resample time-domain data by decimation or interpolation
<code>ifft</code>	Transform <code>iddata</code> objects from frequency to time domain
<code>merge (iddata)</code>	Merge data sets into <code>iddata</code> object
<code>misdata</code>	Reconstruct missing input and output data
<code>nkshift</code>	Shift data sequences
<code>pexcit</code>	Level of excitation of input signals
<code>realdata</code>	Determine whether <code>iddata</code> is based on real-valued signals

<code>resample</code>	Resample time-domain data by decimation or interpolation (requires Signal Processing Toolbox™ software)
<code>retrend</code>	Add offsets or trends to data signals
<code>TrendInfo</code>	Offset and linear trend slope values for detrending data

## Linear Model Identification

<code>advice</code>	Analysis and recommendations for data or estimated linear models
<code>ar</code>	Estimate parameters of AR model for scalar time series
<code>armax</code>	Estimate parameters of ARMAX model using time-domain data
<code>armaxOptions</code>	Option set for <code>armax</code>
<code>arOptions</code>	Option set for <code>ar</code>
<code>arx</code>	Estimate parameters of ARX or AR model using least squares
<code>arxOptions</code>	Option set for <code>ar</code>
<code>arxstruc</code>	Compute and compare loss functions for single-output ARX models
<code>bj</code>	Estimate Box-Jenkins polynomial model using time domain data
<code>bjOptions</code>	Option set for <code>bj</code>
<code>c2d</code>	Convert model from continuous to discrete time
<code>c2dOptions</code>	Create options for continuous- to discrete-time conversions
<code>canon</code>	State-space canonical realization
<code>chgFreqUnit</code>	Change frequency units of frequency-response data model
<code>chgTimeUnit</code>	Change time units of dynamic system
<code>cra</code>	Estimate impulse response using prewhitened-based correlation analysis
<code>d2c</code>	Convert model from discrete to continuous time

<code>d2cOptions</code>	Create option set for discrete- to continuous-time conversions
<code>d2d</code>	Resample discrete-time model
<code>d2dOptions</code>	Create option set for discrete-time resampling
<code>delayest</code>	Estimate time delay (dead time) from data
<code>etfe</code>	Estimate empirical transfer functions and periodograms
<code>feedback</code>	Identify possible feedback data
<code>fft</code>	Transform <code>iddata</code> object to frequency domain data
<code>findstates(idParametric)</code>	Estimate initial states of identified linear state-space model from data
<code>findstatesOptions</code>	Option set for <code>findstates</code>
<code>idfrd</code>	Frequency-response data or model
<code>idpar</code>	Create parameter for initial states and input level estimation
<code>idpoly</code>	Polynomial model with identifiable parameters
<code>idproc</code>	Continuous-time process model with identifiable parameters
<code>idss</code>	State-space model with identifiable parameters
<code>idssdata</code>	State-space data of identified system
<code>idtf</code>	Transfer function model with identifiable parameters
<code>impulseest</code>	Nonparameteric impulse response estimation
<code>init</code>	Set or randomize initial parameter values

<code>iv4</code>	ARX model estimation using four-stage instrumental variable method.
<code>iv4Options</code>	Option set for <code>iv4</code>
<code>ivar</code>	AR model estimation using instrumental variable method
<code>ivstruc</code>	Loss functions for sets of ARX model structures
<code>ivx</code>	ARX model estimation using instrumental variable method with arbitrary instruments
<code>merge</code>	Merge estimated models
<code>n4sid</code>	Estimate state-space model using a subspace method.
<code>n4sidOptions</code>	Option set for <code>n4sid</code>
<code>nuderst</code>	Set step size for numerical differentiation
<code>oe</code>	Estimate Output-Error polynomial model using time or frequency domain data
<code>oeOptions</code>	Option set for <code>oe</code>
<code>pem</code>	Prediction error estimate of linear or nonlinear model
<code>pexcit</code>	Level of excitation of input signals
<code>polydata</code>	Access polynomial coefficients and uncertainties of identified model
<code>polyest</code>	Estimate polynomial model using time or frequency domain data
<code>polyestOptions</code>	Option set for <code>polyest</code>
<code>procest</code>	Estimate process model using time or frequency data
<code>procestOptions</code>	Options set for <code>procest</code>

<code>segment</code>	Segment data and estimate models for each segment
<code>selstruc</code>	Select model order for single-output ARX models
<code>spa</code>	Estimate frequency response with fixed frequency resolution using spectral analysis
<code>spafdr</code>	Estimate frequency response and spectrum using spectral analysis with frequency-dependent resolution
<code>ss2ss</code>	State coordinate transformation for state-space model
<code>ssest</code>	Estimate state-space model using time or frequency domain data
<code>ssestOptions</code>	Option set for <code>ssest</code>
<code>struc</code>	Generate model-order combinations for single-output ARX model estimation
<code>tfdata</code>	Access transfer function data
<code>tfest</code>	Transfer function estimation
<code>tfestOptions</code>	Options set for <code>tfest</code>

## Nonlinear Black-Box Model Identification

<code>addreg</code>	Add custom regressors to nonlinear ARX model
<code>customnet</code>	Custom nonlinearity estimator for nonlinear ARX and Hammerstein-Wiener models
<code>customreg</code>	Custom regressor for nonlinear ARX models
<code>data2state(idnlarx)</code>	Map past input/output data to current states of nonlinear ARX model
<code>deadzone</code>	Class representing dead-zone nonlinearity estimator for Hammerstein-Wiener models
<code>evaluate</code>	Value of nonlinearity estimator at given input
<code>findop(idnlarx)</code>	Compute operating point for nonlinear ARX model
<code>findop(idnlhw)</code>	Compute operating point for Hammerstein-Wiener model
<code>findstates(idnlarx)</code>	Estimate initial states of nonlinear ARX model from data
<code>findstates(idnlgrey)</code>	Estimate initial states of nonlinear grey-box model from data
<code>findstates(idnlhw)</code>	Estimate initial states of nonlinear Hammerstein-Wiener model from data
<code>getDelayInfo</code>	Get input/output delay information for <code>idnlarx</code> model structure
<code>getreg</code>	Regressor expressions and numerical values in nonlinear ARX model
<code>idnlarx</code>	Nonlinear ARX model

<code>idnlhw</code>	Hammerstein-Wiener model
<code>idnlmodel</code>	Superclass for nonlinear models
<code>init</code>	Set or randomize initial parameter values
<code>linapp</code>	Linear approximation of nonlinear ARX and Hammerstein-Wiener models for given input
<code>linear</code>	Class representing linear nonlinearity estimator for nonlinear ARX models
<code>linearize(idnlarx)</code>	Linearize nonlinear ARX model
<code>linearize(idnlhw)</code>	Linearize Hammerstein-Wiener model
<code>neuralnet</code>	Class representing neural network nonlinearity estimator for nonlinear ARX models
<code>nlarx</code>	Estimate nonlinear ARX model
<code>n1hw</code>	Estimate Hammerstein-Wiener model
<code>operspec(idnlarx)</code>	Construct operating point specification object for <code>idnlarx</code> model
<code>operspec(idnlhw)</code>	Construct operating point specification object for <code>idnlhw</code> model
<code>pem</code>	Prediction error estimate of linear or nonlinear model
<code>poly1d</code>	Class representing single-variable polynomial nonlinear estimator for Hammerstein-Wiener models
<code>polyreg</code>	Powers and products of standard regressors



---

<code>pwnlinear</code>	Class representing piecewise-linear nonlinear estimator for Hammerstein-Wiener models
<code>saturation</code>	Class representing saturation nonlinearity estimator for Hammerstein-Wiener models
<code>sigmoidnet</code>	Class representing sigmoid network nonlinearity estimator for nonlinear ARX and Hammerstein-Wiener models
<code>treepartition</code>	Class representing binary-tree nonlinearity estimator for nonlinear ARX models
<code>unitgain</code>	Specify absence of nonlinearities for specific input or output channels in Hammerstein-Wiener models
<code>wavenet</code>	Class representing wavelet network nonlinearity estimator for nonlinear ARX and Hammerstein-Wiener models

## ODE Parameter Estimation

<code>getinit</code>	Values of <code>idnlgrey</code> model initial states
<code>getpar</code>	Parameter values and properties of <code>idnlgrey</code> model parameters
<code>greyest</code>	Linear grey box model estimation
<code>greyestOptions</code>	Option set for <code>greyest</code>
<code>idgrey</code>	Linear ODE (grey-box model) with identifiable parameters
<code>idnlgrey</code>	Nonlinear ODE (grey-box model) with unknown parameters
<code>idnlmodel</code>	Superclass for nonlinear models
<code>init</code>	Set or randomize initial parameter values
<code>pem</code>	Prediction error estimate of linear or nonlinear model
<code>setinit</code>	Set initial states of <code>idnlgrey</code> model object
<code>setpar</code>	Set initial parameter values of <code>idnlgrey</code> model object

## Recursive Model Identification

rarmax	Estimate recursively parameters of ARMAX or ARMA models
rarx	Estimate parameters of ARX or AR models recursively
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
rp1r	Estimate general input-output models using recursive pseudolinear regression method

## Model Analysis

<code>aic</code>	Akaike Information Criterion for estimated model
<code>bandwidth</code>	Frequency response bandwidth
<code>blkdiag</code>	Block-diagonal concatenation of models
<code>bode</code>	Bode plot of frequency response, magnitude and phase of frequency response
<code>bodemag</code>	Bode magnitude response of LTI models
<code>bodeoptions</code>	Create list of Bode plot options
<code>bodeplot</code>	Plot Bode frequency response with additional plot customization options
<code>compare</code>	Compare model output and measured output
<code>compareOptions</code>	Option set for compare
<code>damp</code>	Natural frequency; damping ratio
<code>db2mag</code>	Convert decibels (dB) to magnitude
<code>dcgain</code>	Low-frequency (DC) gain of LTI system
<code>fcats</code>	Concatenate FRD models along frequency dimension
<code>fdel</code>	Delete specified data from frequency response data (FRD) models
<code>forecast</code>	Forecast linear system response into future
<code>forecastOptions</code>	Option set for forecast
<code>fpe</code>	Akaike Final Prediction Error for estimated model

<code>frdata</code>	Access data for frequency response data (FRD) object
<code>freqresp</code>	Frequency response over grid
<code>fselect</code>	Select frequency points or range in FRD model
<code>get</code>	Access model property values
<code>getcov</code>	Parameter covariance information in either raw or factored form
<code>getoptions</code>	Return <code>@PlotOptions</code> handle or plot options property
<code>getpvec</code>	Model parameters and associated uncertainty data
<code>goodnessOfFit</code>	Goodness of fit between test and reference data
<code>identpref</code>	Set System Identification Toolbox preferences
<code>idssdata</code>	State-space data of identified system
<code>impulse</code>	Impulse response plot of dynamic system; impulse response data
<code>impulseplot</code>	Plot impulse response and return plot handle
<code>interp</code>	Interpolate FRD model
<code>iopzmap</code>	Plot pole-zero map for I/O pairs of model
<code>iopzplot</code>	Plot pole-zero map for I/O pairs and return plot handle
<code>isct</code>	Determine if dynamic system model is in continuous time
<code>isdt</code>	Determine if dynamic system model is in discrete time

<code>isreal</code>	Determine whether model parameters or data values are real
<code>issiso</code>	Determine if dynamic system model is single-input/single-output (SISO)
<code>isstable</code>	Determine whether system is stable
<code>lsim</code>	Simulate time response of dynamic system to arbitrary inputs
<code>lsiminfo</code>	Compute linear response characteristics
<code>lsimplot</code>	Simulate response of dynamic system to arbitrary inputs and return plot handle
<code>mag2db</code>	Convert magnitude to decibels (dB)
<code>ndims</code>	Query number of dimensions of dynamic system model or model array
<code>noise2meas</code>	Noise component of model
<code>noisecnv</code>	Transform <code>idmodel</code> object with noise channels to model with measured channels only
<code>norm</code>	Norm of linear model
<code>nparams</code>	Number of model parameters
<code>nyquist</code>	Nyquist plot
<code>nyquistoptions</code>	List of Nyquist plot options
<code>nyquistplot</code>	Nyquist plot with additional plot customization options
<code>order</code>	Query model order
<code>pe</code>	Prediction error for an identified model
<code>peOptions</code>	Option set for <code>pe</code>

<code>plot</code>	Plot iddata or model objects
<code>pole</code>	Compute poles of dynamic system
<code>polydata</code>	Access polynomial coefficients and uncertainties of identified model
<code>predict</code>	K-step ahead prediction
<code>predictOptions</code>	Option set for <code>predict</code>
<code>present</code>	Display model information, including estimated uncertainty
<code>pzmap</code>	Pole-zero plot of dynamic system
<code>pzoptions</code>	Create list of pole/zero plot options
<code>pzplot</code>	Pole-zero map of dynamic system model with plot customization options
<code>resid</code>	Compute and test model residuals (prediction errors)
<code>rsample</code>	Random sampling of linear identified systems
<code>selstruc</code>	Select model order for single-output ARX models
<code>set</code>	Set or modify model properties
<code>setcov</code>	Set parameter covariance data in identified model
<code>setoptions</code>	Set plot options for response plot
<code>setPolyFormat</code>	Specify format for B and F polynomials of multi-input polynomial model for backward compatibility
<code>setpvec</code>	Modify value of model parameters
<code>showConfidence</code>	Display confidence regions on response plots for identified models

<code>sim</code>	Simulate response of identified models to arbitrary inputs
<code>sim(idnlarx)</code>	Simulate nonlinear ARX model
<code>sim(idnlgrey)</code>	Simulate nonlinear ODE model
<code>sim(idnlhw)</code>	Simulate Hammerstein-Wiener model
<code>simOptions</code>	Option set for <code>sim</code>
<code>simsd</code>	Simulate linear models with uncertainty using Monte Carlo method
<code>simsdOptions</code>	Option set for <code>simsd</code>
<code>size</code>	Query output/input/array dimensions of input-output model and number of frequencies of FRD model
<code>ssdata</code>	Access state-space model data
<code>stack</code>	Build model array by stacking models or model arrays along array dimensions
<code>step</code>	Step response plot of dynamic system
<code>stepinfo</code>	Rise time, settling time, and other step response characteristics
<code>stepplot</code>	Plot step response and return plot handle
<code>strseq</code>	Create sequence of indexed strings
<code>tfdata</code>	Access transfer function data
<code>timeoptions</code>	Create list of time plot options
<code>zero</code>	Zeros and gain of SISO dynamic system
<code>zpkdata</code>	Access zero-pole-gain data



## Simulation and Prediction

<code>idinput</code>	Generate input signals
<code>predict</code>	K-step ahead prediction
<code>predictOptions</code>	Option set for <code>predict</code>
<code>sim</code>	Simulate response of identified models to arbitrary inputs
<code>sim(idnlarx)</code>	Simulate nonlinear ARX model
<code>sim(idnlgrey)</code>	Simulate nonlinear ODE model
<code>sim(idnlhw)</code>	Simulate Hammerstein-Wiener model
<code>simOptions</code>	Option set for <code>sim</code>
<code>simsd</code>	Simulate linear models with uncertainty using Monte Carlo method
<code>simsdOptions</code>	Option set for <code>simsd</code>

## **System Identification Tool GUI**

`ident`

Open System Identification Tool GUI

`midprefs`

Set folder for storing `idprefs.mat` containing GUI startup information

# Functions – Alphabetical List

---

# absorbDelay

---

**Purpose** Replace time delays by poles at  $z = 0$  or phase shift

**Syntax**  
`sysnd = absorbDelay(sysd)`  
`[sysnd,G] = absorbDelay(sysd)`

**Description** `sysnd = absorbDelay(sysd)` absorbs all time delays of the dynamic system model `sysd` into the system dynamics or the frequency response data.

For discrete-time models (other than frequency response data models), a delay of  $k$  sampling periods is replaced by  $k$  poles at  $z = 0$ . For continuous-time models (other than frequency response data models), time delays have no exact representation with a finite number of poles and zeros. Therefore, use `pade` to compute a rational approximation of the time delay.

For frequency response data models in both continuous and discrete time, `absorbDelay` absorbs all time delays into the frequency response data as a phase shift.

`[sysnd,G] = absorbDelay(sysd)` returns the matrix `G` that maps the initial states of the `ss` model `sysd` to the initial states of the `sysnd`.

## Examples

### Example 1

Create a discrete-time transfer function that has a time delay and absorb the time delay into the system dynamics as poles at  $z = 0$ .

```
z = tf('z',-1);  
sysd = (-.4*z - .1)/(z^2 + 1.05*z + .08);  
sysd.InputDelay = 3
```

These commands produce the result:

Transfer function:  
                  -0.4 z - 0.1  
 $z^{(-3)} * \frac{\text{-----}}{z^2 + 1.05 z + 0.08}$

Sampling time: unspecified

The display of `sysd` represents the `InputDelay` as a factor of  $z^{-3}$ , separate from the system poles that appear in the transfer function denominator.

Absorb the delay into the system dynamics.

```
sysnd = absorbDelay(sysd)
```

The display of `sysnd` shows that the factor of  $z^{-3}$  has been absorbed as additional poles in the denominator.

```
Transfer function:
      -0.4 z - 0.1
-----
z^5 + 1.05 z^4 + 0.08 z^3
```

Sampling time: unspecified

Additionally, `sysnd` has no input delay:

```
sysnd.InputDelay
```

```
ans =
```

```
0
```

## Example 2

Convert "nk" into regular coefficients of a polynomial model.

Consider the discrete-time polynomial model:

```
m = idpoly(1,[0 0 0 2 3]);
```

The value of the B polynomial, `m.b`, has 3 leading zeros. Two of these zeros are treated as input-output delays. Consequently:

```
sys = tf(m)
```

# absorbDelay

---

creates a transfer function such that the numerator is [0 2 3] and the IO delay is 2. In order to treat the leading zeros as regular B coefficients, use absorbDelay:

```
m2 = absorbDelay(m);  
sys2 = tf(m2);
```

sys2's numerator is [0 0 0 2 3] and IO delay is 0. The model m2 treats the leading zeros as regular coefficients by freeing their values. m2.Structure.b.Free(1:2) is TRUE while m.Structure.b.Free(1:2) is FALSE.

## See Also

hasdelay | pade | totaldelay

---

<b>Purpose</b>	Analysis and recommendations for data or estimated linear models
<b>Syntax</b>	<code>advice(data)</code> <code>advice(model,data)</code>
<b>Description</b>	<p><code>advice(data)</code> displays the following information about the data in the MATLAB® Command Window:</p> <ul style="list-style-type: none"><li>• What are the excitation levels of the signals and how does this affect the model orders? See also <code>pexcit</code>.</li><li>• Does it make sense to remove constant offsets and linear trends from the data? See also <code>detrend</code>.</li><li>• Is there an indication of output feedback in the data? See also <code>feedback</code>.</li><li>• Would a nonlinear ARX model perform better than a linear ARX model?</li></ul> <p><code>advice(model,data)</code> displays the following information about the estimated linear model in the MATLAB Command Window:</p> <ul style="list-style-type: none"><li>• Does the model capture essential dynamics of the system and the disturbance characteristics?</li><li>• Is the model order higher than necessary?</li><li>• Is there potential output feedback in the validation data?</li></ul>
<b>Input Arguments</b>	<p><b>data</b> Specify <code>data</code> as an <code>iddata</code> object.</p> <p><b>model</b> Specify <code>model</code> as an <code>idtf</code>, <code>idgrey</code>, <code>idpoly</code>, <code>idproc</code>, or <code>idss</code> model object.</p>
<b>See Also</b>	<code>detrend</code>   <code>feedback</code>   <code>iddata</code>   <code>pexcit</code>

# addreg

---

**Purpose** Add custom regressors to nonlinear ARX model

**Syntax**  
`m = addreg(model,regressors)`  
`m = addreg(model,regressors,output)`

**Description** `m = addreg(model,regressors)` adds custom regressors to a nonlinear ARX model by appending the `CustomRegressors` `model` property. `model` and `m` are `idnlarx` objects. For single-output models, `regressors` is an object array of regressors you create using `customreg` or `polyreg`, or a cell array of string expressions. For multiple-output models, `regressors` is 1-by-ny cell array of `customreg` objects or 1-by-ny cell array of cell arrays of string expressions. `addreg` adds each element of the ny cells to the corresponding `model` output channel. If `regressors` is a single regressor, `addreg` adds this regressor to all output channels.

`m = addreg(model,regressors,output)` adds regressors `regressors` to specific output channels `output` of a multiple-output model. `output` is a scalar integer or vector of integers, where each integer is the index of a model output channel. Specify several pairs of `regressors` and `output` values to add different regressor variables to the corresponding output channels.

**Examples** Add regressors to a nonlinear ARX model as a cell array of strings:

```
% Create nonlinear ARX model with standard regressors:  
m1 = idnlarx([4 2 1], 'wavenet', 'nlr', [1:3]);  
% Create model with additional custom regressors:  
m2 = addreg(m1, {'y1(t-2)^2'; 'u1(t)*y1(t-7)'});  
% List all standard and custom regressors of m2:  
getreg(m2)
```

---

Add regressors to a nonlinear ARX model as `customreg` objects:

```
% Create nonlinear ARX model with standard regressors:  
m1 = idnlarx([4 2 1], 'wavenet', 'nlr', [1:3]);  
% Create a model based on m1 with custom regressors:
```



```
r1 = customreg(@(x)x^2, {'y1'}, 2)
r2 = customreg(@(x,y)x*y, {'u1','y1'}, [0 7])
m2 = addreg(m1,[r1 r2]);
```

## See Also

customreg | getreg | nlarx | polyreg

## How To

- “Identifying Nonlinear ARX Models”

<b>Purpose</b>	Akaike Information Criterion for estimated model
<b>Syntax</b>	<pre>am = aic(model) am = aic(model1,model2,...)</pre>
<b>Arguments</b>	<p><code>model</code> Name of an <code>idtf</code>, <code>idgrey</code>, <code>idpoly</code>, <code>idproc</code>, <code>idss</code>, <code>idnlarx</code>, <code>idnlhw</code>, or <code>idnlgrey</code> model object.</p>
<b>Description</b>	<p><code>am = aic(model)</code> returns a scalar value of the Akaike's Information Criterion (AIC) for the estimated <code>model</code>.</p> <p><code>am = aic(model1,model2,...)</code> returns a row vector containing AIC values for the estimated models <code>model1,model2,...</code>.</p>
<b>Tips</b>	Akaike's Information Criterion (AIC) provides a measure of model quality by simulating the situation where the model is tested on a different data set. After computing several different models, you can compare them using this criterion. According to Akaike's theory, the most accurate model has the smallest AIC.

---

**Note** If you use the same data set for both model estimation and validation, the fit always improves as you increase the model order and, therefore, the flexibility of the model structure.

---

Akaike's Information Criterion (AIC) is defined by the following equation:

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

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

The loss function  $V$  is defined by the following equation:

$$V = \det \left( \frac{1}{N} \sum_1^N \varepsilon(t, \theta_N) (\varepsilon(t, \theta_N))^T \right)$$

where  $\theta_N$  represents the estimated parameters.

For  $d \ll N$ :

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

---

**Note** *AIC* is approximately equal to  $\log(FPE)$ .

---

AIC is formally defined as the negative log-likelihood function  $\Lambda$ , 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  $\Lambda$ , the logarithm of the likelihood function is

$$L(\theta, \Lambda) = -\frac{1}{2} \sum_1^N \varepsilon(t, \theta)^T \Lambda^{-1} \varepsilon(t, \theta) - \frac{N}{2} \log(\det \Lambda) + \text{const}$$

Maximizing this analytically with respect to  $\Lambda$ , and then maximizing the result with respect to  $\theta$ , gives

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

## 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

fpe

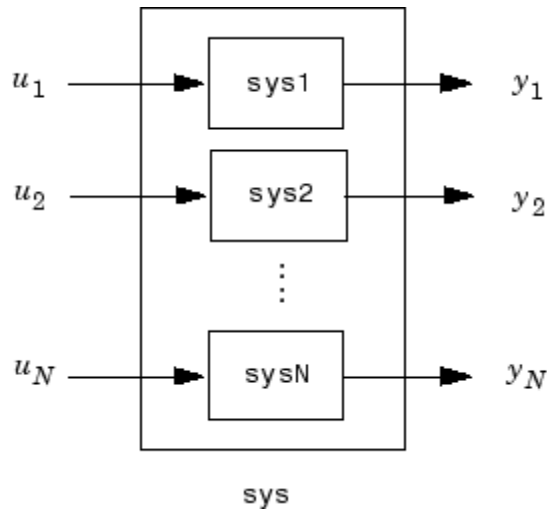
# append

**Purpose** Group models by appending their inputs and outputs

**Syntax** `sys = append(sys1,sys2,...,sysN)`

**Description** `sys = append(sys1,sys2,...,sysN)`

`append` appends the inputs and outputs of the models `sys1,...,sysN` to form the augmented model `sys` depicted below.



For systems with transfer functions  $H_1(s), \dots, H_N(s)$ , the resulting system `sys` has the block-diagonal transfer function

$$\begin{bmatrix} H_1(s) & 0 & \dots & 0 \\ 0 & H_2(s) & \dots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ 0 & \dots & 0 & H_N(s) \end{bmatrix}$$

For state-space models `sys1` and `sys2` with data  $(A_1, B_1, C_1, D_1)$  and  $(A_2, B_2, C_2, D_2)$ , `append(sys1,sys2)` produces the following state-space model:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} B_1 & 0 \\ 0 & B_2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} C_1 & 0 \\ 0 & C_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

## Arguments

The input arguments `sys1`, ..., `sysN` can be model objects `s` of any type. Regular matrices are also accepted as a representation of static gains, but there should be at least one model in the input list. The models should be either all continuous, or all discrete with the same sample time. When appending models of different types, the resulting type is determined by the precedence rules (see “Precedence Rules That Determine Model Type” for details).

There is no limitation on the number of inputs.

## Examples

The commands

```
sys1 = tf(1,[1 0]);
sys2 = ss(1,2,3,4);
sys = append(sys1,10,sys2)
```

produce the state-space model

```
a =
      x1  x2
x1    0   0
x2    0   1

b =
      u1  u2  u3
x1    1   0   0
x2    0   0   2

c =
      x1  x2
y1    1   0
```

# append

---

```
y2  0  0
y3  0  3

d =
      u1  u2  u3
y1  0   0   0
y2  0  10   0
y3  0   0   4
```

Continuous-time model.

## See Also

`connect` | `feedback` | `parallel` | `series`

**Purpose**

Estimate parameters of AR model for scalar time series

**Syntax**

```
m = ar(y,n)
[m,ref1] = ar(y,n,approach>window)
m= ar(y,n,Name,Value)
m= ar(y,n, __ ,opt)
```

**Description**


---

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

---

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

`[m,ref1] = ar(y,n,approach>window)` returns an `idpoly` model `m` and the variable `ref1`. For the two lattice-based approaches, 'burg' and 'gl', `ref1` stores the reflection coefficients in the first row, and the corresponding loss function values in the second row. The first column of `ref1` is the zeroth-order model, and the (2,1) element of `ref1` is the norm of the time series itself.

`m= ar(y,n,Name,Value)` specifies model structure attributes using one or more `Name,Value` pair arguments.

`m= ar(y,n, __ ,opt)` specifies the estimations options using `opt`.

**Input 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**

One of the following text strings, specifying the algorithm for computing the least squares AR model:

- '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**

One of the following text strings, specifying how to use information about the data outside the measured time interval (past and future values):

- '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.

**opt**

Estimation options.



`opt` is an options set that specifies the following:

- data offsets
- covariance handling
- estimation approach
- estimation window

Use `arOptions` to create the options set.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

#### **ts**

Positive scalar that specifies the sample time. Use when you specify `Y` as double vector rather than an `IDDATA` object.

#### **IntegrateNoise**

Boolean value that specifies whether the noise source contains an

integrator or not. Use it to create "ARI" structure models:  $Ay = \frac{e}{(1-z^{-1})}$

**Default:** false

## **Output Arguments**

#### **m**

An `idpoly` model.

#### **ref1**

An 2-by-2 array. The first row stores the reflection coefficients, and the second row stores the corresponding loss function values. The first

column of `ref1` is the zeroth-order model, and the (2,1) element of `ref1` is the norm of the time series itself.

## 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)
```

Estimate an ARI model.

```
load iddata9 z9
Ts = z9.Ts;
y = cumsum(z9.y);
model = ar(y, 4, 'ls', 'Ts', Ts, 'IntegrateNoise', true)
compare(y,model,5) % 5 step ahead prediction
```

Use option set to choose 'ls' estimation approach and to specify that covariance matrix should not be estimated.

```
y = rand(100,1);
opt = arOptions('Approach', 'ls', 'EstCovar', false);
model = ar(y, N, opt);
```

## Algorithms

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.

<b>Method</b>	<b>Approach and Windowing</b>
Modified Covariance Method	(Default) Forward-backward approach and no windowing.
Correlation Method	Yule-Walker approach, which corresponds to least squares plus pre- and postwindowing.
Covariance Method	Least squares approach with no windowing. <code>arx</code> uses this routine.

## References

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

## See Also

`arOptions` | `idpoly` | `arx` | `etfe` | `ivar` | `pem` | `spa` | `forecast`

# armax

---

**Purpose** Estimate parameters of ARMAX model using time-domain data

**Syntax**

```
sys = armax(data,[na nb nc nk])
sys = armax(data,[na nb nc nk],Name,Value)
sys = armax(data,init_sys)
sys = armax(data, __ ,opt)
```

## Description

---

**Note** armax supports only time-domain data. For frequency-domain data, use `oe`.

---

`sys = armax(data,[na nb nc nk])` returns an `idpoly` model, `sys`, with estimated parameters and covariance (parameter uncertainties). Estimates the parameters using the prediction-error method and specified polynomial orders.

`sys = armax(data,[na nb nc nk],Name,Value)` returns an `idpoly` model, `sys`, with additional options specified by one or more `Name,Value` pair arguments.

`sys = armax(data,init_sys)` estimates a polynomial model using the ARMAX structure polynomial model `init_sys` to configure the initial parameterization.

`sys = armax(data, __ ,opt)` specifies estimation options using the option set `opt`.

**Tips**

- Use the `IntegrateNoise` property to add integrators to the noise source.

## Input Arguments

### **data**

Estimation data.

Specify `data` as an `iddata` object containing the time-domain input-output data.

You cannot use frequency-domain data for estimating ARMAX models.

**[na nb nc nk]**

Polynomial orders.

[na nb nc nk] define the polynomial orders of an “ARMAX Model” on page 2-22.

- na — Order of the polynomial  $A(q)$ .  
Specify na as an  $N_y$ -by- $N_y$  matrix of nonnegative integers.  $N_y$  is the number of outputs.
- nb — Order of the polynomial  $B(q) + 1$ .  
nb is an  $N_y$ -by- $N_u$  matrix of nonnegative integers.  $N_y$  is the number of outputs and  $N_u$  is the number of inputs.
- nc — Order of the polynomial  $C(q)$ .  
nc is a column vector of nonnegative integers of length  $N_y$ .  $N_y$  is the number of outputs.
- nk — Input-output delay expressed as fixed leading zeros of the  $B$  polynomial.  
Specify nk as an  $N_y$ -by- $N_u$  matrix of nonnegative integers.  $N_y$  is the number of outputs and  $N_u$  is the number of inputs.

**init\_sys**

Linear polynomial model that configures the initial parameterization of `sys`.

`init_sys` must be an ARMAX model. You may obtain `init_sys` by either performing an estimation using measured data, or by direct construction.

Use the `Structure` property of `init_sys` to configure initial guesses and constraints for  $A(q)$ ,  $B(q)$ , and  $C(q)$ .

To specify an initial guess for, say, the  $A(q)$  term of `init_sys`, set `init_sys.Structure.a.Value` as the initial guess.

To specify constraints for, say, the  $B(q)$  term of `init_sys`:

- set `init_sys.Structure.b.Minimum` to the minimum  $B(q)$  coefficient values
- set `init_sys.Structure.b.Maximum` to the maximum  $B(q)$  coefficient values
- set `init_sys.Structure.b.Free` to indicate which  $B(q)$  coefficients are free for estimation

You can similarly specify the initial guess and constraints for the other polynomials.

If `opt` is not specified, and `init_sys` was created by estimation, then the estimation options from `init_sys.Report.OptionsUsed` are used.

## **opt**

Estimation options.

`opt` is an options set that specifies estimation options, including:

- estimation objective
- handling of initial conditions
- numerical search method to be used in estimation

Use `armaxOptions` to create the options set.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

## **InputDelay**

Input delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. Specify input delays in integer multiples of the sampling period  $T_s$ . For example, `InputDelay = 3` means a delay of three sampling periods.

For a system with  $N_u$  inputs, set `InputDelay` to an  $N_u$ -by-1 vector, where each entry is a numerical value representing the input delay for the corresponding input channel. You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0 for all input channels

### **ioDelay**

Transport delays. `ioDelay` is a numeric array specifying a separate transport delay for each input/output pair.

Specify transport delays as integers denoting delay of a multiple of the sampling period  $T_s$ .

For a MIMO system with  $N_y$  outputs and  $N_u$  inputs, set `ioDelay` to a  $N_y$ -by- $N_u$  array, where each entry is a numerical value representing the transport delay for the corresponding input/output pair. You can also set `ioDelay` to a scalar value to apply the same delay to all input/output pairs. Useful as a replacement for the `nk` order, you can factor out  $\max(nk-1, 0)$  lags as the `ioDelay` value.

**Default:** 0 for all input/output pairs

### **IntegrateNoise**

Logical vector specifying integrators in the noise channel.

`IntegrateNoise` is a logical vector of length  $N_y$ , where  $N_y$  is the number of outputs.

Setting `IntegrateNoise` to true for a particular output results in the model:

$$A(q)y(t) = B(q)u(t - nk) + \frac{C(q)}{1 - q^{-1}}e(t)$$

Where,  $\frac{1}{1 - q^{-1}}$  is the integrator in the noise channel,  $e(t)$ .

Use `IntegrateNoise` to create an ARIMA model.

For example,

```
load iddata9 z9;
z9.y = cumsum(z9.y); %integrated data
sys = armax(z9,[4 1], 'IntegrateNoise', true);
compare(z9,sys,10) %10-step ahead prediction
```

**Default:** `false(Ny, 1)` ( $N_y$  is the number of outputs.)

## Output Arguments

### **sys**

Identified ARMAX structure polynomial model.

`sys` is a discrete-time `idpoly` model, which encapsulates the estimated  $A$ ,  $B$  and  $C$  polynomials and the parameter covariance information.

## Definitions

### **ARMAX Model**

The ARMAX model structure is

$$y(t) + a_1y(t-1) + \dots + a_{n_a}y(t-n_a) = b_1u(t-n_k) + \dots + b_{n_b}u(t-n_k-n_b+1) + c_1e(t-1) + \dots + c_{n_c}e(t-n_c) + e(t)$$

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.
- $y(t-1)\dots y(t-n_a)$  — Previous outputs on which the current output depends.
- $u(t-n_k)\dots u(t-n_k-n_b+1)$  — Previous and delayed inputs on which the current output depends.
- $e(t-1)\dots e(t-n_c)$  — White-noise disturbance value.

The parameters  $n_a$ ,  $n_b$ , and  $n_c$  are the orders of the ARMAX model, and  $n_k$  is the delay.  $q$  is the delay operator. Specifically,

$$A(q) = 1 + a_1q^{-1} + \dots + a_{n_a}q^{-n_a}$$

$$B(q) = b_1 + b_2q^{-1} + \dots + b_{n_b}q^{-n_b+1}$$

$$C(q) = 1 + c_1q^{-1} + \dots + c_{n_c}q^{-n_c}$$

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]`

### ARIMAX Model

An ARIMAX model structure is similar to ARMAX, except that it contains an integrator in the noise source  $e(t)$ :

$$A(q)y(t) = B(q)u(t - nk) + \frac{1}{(1 - q^{-1})}e(t)$$

If there are no inputs, this reduces to an ARIMA model:

$$A(q)y(t) = \frac{1}{(1 - q^{-1})}e(t)$$

## Examples

### Specifying Estimation Options

Estimate an ARMAX model from measured data and specify the estimation options.

Estimate an ARMAX model with simulation focus, using 'lm' as the search method and maximum number of search iterations set to 10.

```
load twotankdata
z = iddata(y,u,0.2);
opt = armaxOptions;
opt.Focus = 'simulation';
opt.SearchMethod = 'lm';
opt.SearchOption.MaxIter = 10;
opt.Display = 'on';
sys = armax(z, [2 2 2 1], opt)
```

The termination conditions for measured component of the model shown in the progress viewer is that the maximum number of iterations were reached. To improve results, re-estimate the model using a greater value for MaxIter, or we can continue iterations on the previously estimated model as follows:

```
sys2 = armax(z, sys);
compare(z, sys, sys2)
```

where sys2 refines the parameters of sys to improve the fit to data.

## Estimate an ARIMA Model

Estimate an ARIMA Model from measured data.

Estimate a 4th order ARIMA model for univariate time series data.

```
load iddata9
z9.y = cumsum(z9.y); % integrated data
model = armax(z9, [4 1], 'IntegrateNoise', true);
compare(z9, model, 10) % 10-step ahead prediction
```

## Estimate ARMAX Models Iteratively

Estimate ARMAX models of varying orders iteratively from measured data.

Estimate ARMAX models of orders varying between 1 and 4 for dryer data

```
load dryer2
z = iddata(y2,u2,0.08,'Tstart',0);
na = 2:4; nc = 1:2; nk = 0:2;
models = cell(1,18);
ct = 1;
for i = 1:3
    na_ = na(i);
    nb_ = na_;
    for j = 1:2
        nc_ = nc(j);
        for k = 1:3
            nk_ = nk(k);
            models{ct} = armax(z, [na_, nb_, nc_, nk_]);
            ct = ct+1;
        end
    end
end
```

Stack the estimated models and compare their simulated responses to estimation data  $z$ .

```
models = stack(1,models{:});  
compare(z,models)
```

## Algorithms

An iterative search algorithm minimizes a robustified quadratic prediction error criterion. The iterations are terminated either when the maximum number of iterations is reached, or when the expected improvement is less than the specified tolerance, or when a lower value of the criterion cannot be found. You can get information about the stopping criteria using `sys.Report.Termination`.

Use the `armaxOptions` option set to create and configure options affecting the estimation results. In particular, set the search algorithm attributes, such as `MaxIter` and `Tolerance`, using the `'SearchOption'` property.

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

The cutoff value for the robustification is based on the `Advanced.ErrorThreshold` estimation option and on the estimated standard deviation of the residuals from the initial parameter estimate. It is not recalculated during the minimization. By default, no robustification is performed; the default value of `ErrorThreshold` option is 0.

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 estimation option `'Display'` is `'On'` or `'Full'`. With `'Display' = '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 (cost) are given and the

Gauss-Newton vector and its norm are also displayed. With 'Display' = '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.

**Alternatives**

armax does not support continuous-time model estimation. Use `tfest` to estimate a continuous-time transfer function model, or `ssest` to estimate a continuous-time state-space model.

**See Also**

`armaxOptions` | `arx` | `bj` | `oe` | `polyest` | `ssest` | `tfest` | `idpoly` | `iddata` | `idfrd` | `forecast`

# armaxOptions

---

**Purpose** Option set for armax

**Syntax**  
`opt = armaxOptions`  
`opt = armaxOptions(Name,Value)`

**Description** `opt = armaxOptions` creates the default options set for armax.  
`opt = armaxOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### InitialCondition

Specify handling of initial conditions during estimation.

`InitialCondition` takes one of the following values:

- 'zero' — Initial conditions are set to 0.
- 'estimate' — Initial conditions, along with the model parameters, are treated as estimation parameters.
- 'backcast' — Initial conditions are estimated using a backcasting (backward filtering) process.
- 'auto' — The software chooses the initial condition handling method based on the estimation data.

**Default:** 'auto'

### Focus

Defines how the errors  $e$  between the measured and the modeled outputs are weighed at specific frequencies during the minimization of the prediction error.

Higher weighting at specific frequencies emphasizes the requirement for a good fit at these frequencies.

**Focus** can take the following values:

- '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.

This method provides a stable model.

- 'prediction' — 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.
- 'stability' — Same as 'prediction', but with model stability enforced.
- Passbands — Row vector or matrix containing frequency values that define desired passbands. For example:

```
[w1,wh]
[w11,w1h;w21,w2h;w31,w3h;...]
```

where  $w1$  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.

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

- A single-input-single-output (SISO) linear system.
- The {A,B,C,D} format, which specifies the state-space matrices of the filter.
- The {numerator, denominator} format, 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. To obtain a good model fit for a specific frequency range, you must choose the filter with a passband in this range. The estimation result is the same if you first prefilter the data using `idfilt`.

- Weighting vector — 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.

**Default:** 'prediction'

## **EstCovar**

Controls whether parameter covariance data is generated or not.

If `EstCovar` is `true`, then use `getcov` to fetch the covariance matrix from the estimated model.

**Default:** `true`

## **Display**

Specifies whether estimation progress should be displayed.

`Display` requires one of the following strings:

- 'on' — Information on model structure and estimation results are displayed in a progress viewer window
- 'off' — No progress or results information is displayed



**Default:** 'off'

### **InputOffset**

Removes offset from time domain input data during estimation.

Specify as a column vector of length  $Nu$ , where  $Nu$  is the number of inputs.

Use [] to indicate no offset.

For multiexperiment data, specify **InputOffset** as a  $Nu$ -by- $Ne$  matrix.  $Nu$  is the number of inputs, and  $Ne$  is the number of experiments.

Each entry specified by **InputOffset** is subtracted from the corresponding input data.

**Default:** []

### **OutputOffset**

Removes offset from time domain output data during estimation.

Specify as a column vector of length  $Ny$ , where  $Ny$  is the number of outputs.

Use [] to indicate no offset.

For multiexperiment data, specify **OutputOffset** as a  $Ny$ -by- $Ne$  matrix.  $Ny$  is the number of outputs, and  $Ne$  is the number of experiments. Each entry specified by **OutputOffset** is subtracted from the corresponding output data.

**Default:** []

### **SearchMethod**

Search method used for iterative parameter estimation.

**SearchMethod** is a string that can take the following values:

- gn — The subspace Gauss-Newton direction. Singular values of the Jacobian matrix less than  $GnPinvConst * \epsilon * \max(\text{size}(J)) * \text{norm}(J)$

are discarded when computing the search direction.  $J$  is the Jacobian matrix. The Hessian matrix is approximated by  $J^T J$ . If there is no improvement in this direction, the function tries the gradient direction.

- `gna` — An adaptive version of subspace Gauss-Newton approach, suggested by Wills and Ninness [1]. Eigenvalues less than  $\text{gamma} \cdot \max(\text{sv})$  of the Hessian are neglected, where  $\text{sv}$  are the singular values of the Hessian. The Gauss-Newton direction is computed in the remaining subspace. *gamma* has the initial value `InitGnaTol` (see `Advanced` for more information) 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  $2 \cdot \text{LMStep}$  each time a search is successful without any bisections.
- `lm` — Uses the Levenberg-Marquardt method. This means that the next parameter value is  $-\text{pinv}(H+d \cdot I) \cdot \text{grad}$  from the previous one, where  $H$  is the Hessian,  $I$  is the identity matrix, and  $\text{grad}$  is the gradient.  $d$  is a number that is increased until a lower value of the criterion is found.
- `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.
- `grad` — The steepest descent gradient search method.
- `auto` — A choice among the above is made in the algorithm. The descent direction is calculated using `gn`, `gna`, `lm` and `grad` successively, in that order, at each iteration until a sufficient reduction in error is achieved.

**Default:** 'auto'

## SearchOption

`SearchOption` is an options set for the search algorithm with the fields:

SearchMethod	SearchOption
gn,gna,lm,grad and auto	<ul style="list-style-type: none"> <li>• <b>Tolerance</b> — 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 <b>Tolerance</b>, the iterations are stopped. 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. <b>Default:</b> 0.01</li> <li>• <b>MaxIter</b> — Maximum number of iterations during loss-function minimization. The iterations stop when <b>MaxIter</b> is reached or another stopping criterion is satisfied, such as <b>Tolerance</b>. Setting <b>MaxIter</b> = 0 returns the result of the startup procedure. Use <code>sys.Report.Termination.Iterations</code> to get the actual number of iterations during an estimation, where <code>sys</code> is an <code>idtf</code> model. <b>Default:</b> 20</li> <li>• <b>Advanced</b> — Search settings: <ul style="list-style-type: none"> <li>▪ <b>GnPinvConst</b> — Singular values of the Jacobian that are smaller than <code>GnPinvConst*max(size(J)*norm(J)*eps)</code> are discarded when computing the search direction and <b>SearchMethod</b> is 'gn'. <b>GnPinvConst</b> must be a positive real value. <b>Default:</b> 10000</li> <li>▪ <b>InitGnaTol</b> — Initial value of <i>gamma</i> when <b>SearchMethod</b> is 'gna'. <b>Default:</b> .0001</li> </ul> </li> </ul>

SearchMethod	SearchOption
	<ul style="list-style-type: none"><li data-bbox="583 361 1248 453">▪ <b>LMStartValue</b> — Starting value of search-direction length <math>d</math> in the Levenberg-Marquardt method. Applicable when <b>SearchMethod</b> is 'lm'. <b>Default:</b> .001</li><li data-bbox="583 522 1267 673">▪ <b>LMStep</b> — Size of the Levenberg-Marquardt step. The next value of the search-direction length <math>d</math> in the Levenberg-Marquardt method is <b>LMStep</b> times the previous one. Applicable when <b>SearchMethod</b> is 'lm'. <b>Default:</b> 2</li><li data-bbox="583 743 1233 852">▪ <b>MaxBisections</b> — Maximum number of bisections used by the line search along the search direction. <b>Default:</b> 25</li><li data-bbox="583 874 1267 1025">▪ <b>MaxFunEvals</b> — Iterations are stopped if the number of calls to the model file exceeds this value. MaxFunEvals must be a positive integer value. <b>Default:</b> Inf</li><li data-bbox="583 1048 1267 1199">▪ <b>MinParChange</b> — Smallest parameter update allowed per iteration MinParChange must be a positive, real value. <b>Default:</b> 0</li><li data-bbox="583 1222 1267 1407">▪ <b>RelImprovement</b> — Iterations are stopped if the relative improvement of the criterion function is less than <b>RelImprovement</b>. RelImprovement must be a positive integer value. <b>Default:</b> 0</li></ul>

SearchMethod	SearchOption
	<ul style="list-style-type: none"> <li>▪ <b>StepReduction</b> — Suggested parameter update is reduced by the factor <code>StepReduction</code> after each try until either <code>MaxBisections</code> tries are completed or a lower value of the criterion function is obtained.</li> </ul> <p><code>StepReduction</code> must be a positive, real value that is greater than 1.</p> <p><b>Default:</b> 2</p>
lsqnonlin	<ul style="list-style-type: none"> <li>• <b>TolFun</b> — Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values.</li> </ul> <p>The value of <code>TolFun</code> is the same as that of <code>sys.SearchOption.Advanced.TolFun</code>.</p> <p><b>Default:</b> 1e-5</p> <ul style="list-style-type: none"> <li>• <b>TolX</b> — Termination tolerance on the estimated parameter values.</li> </ul> <p>The value of <code>TolX</code> is the same as that of <code>sys.SearchOption.Advanced.TolX</code>.</p> <p><b>Default:</b> 1e-6</p> <ul style="list-style-type: none"> <li>• <b>MaxIter</b> — Maximum number of iterations during loss-function minimization. The iterations stop when <code>MaxIter</code> is reached or another stopping criterion is satisfied, such as <code>TolFun</code> etc..</li> </ul> <p>The value of <code>MaxIter</code> is the same as that of <code>sys.SearchOption.Advanced.MaxIter</code>.</p> <p><b>Default:</b> 20</p> <ul style="list-style-type: none"> <li>• <b>Advanced</b> — Options set for <code>lsqnonlin</code>.</li> </ul> <p>For more information, see “Optimization Options”.</p>

SearchMethod	SearchOption
	Use <code>optimset('lsqnonlin')</code> to create an options set for <code>lsqnonlin</code> and then modify it to specify its various options.

## Advanced

Advanced is a structure with the following fields:

- **ErrorThreshold** — Specifies when to adjust the weight of large errors from quadratic to linear.

Errors larger than **ErrorThreshold** times the estimated standard deviation have a linear weight in the 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 the chapter “Computing the Estimate” of [1].)

**ErrorThreshold** = 0 disables robustification and leads to a purely quadratic criterion. When estimating with frequency-domain data, **ErrorThreshold** is set to zero. **ErrorThreshold** = 1.6 is a good choice for estimation when data contains outliers.

**Default:** 0

- **MaxSize** — Specifies the maximum number of elements in a segment when input-output data is split into segments.

**MaxSize** must be a positive integer.

**Default:** 250000

- **StabilityThreshold** — Specifies thresholds for stability tests.

**StabilityThreshold** is a structure with the following fields:

- **s** — Specifies the location of the right-most pole to test the stability of continuous-time models. A model is considered stable when its right-most pole is to the left of **s**.

**Default:** 0

- `z` — 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 `z` from the origin.

**Default:** `1+sqrt(eps)`

- `AutoInitThreshold` — Specifies when to automatically estimate the initial conditions.

When `InitialCondition = 'Auto'`, the initial condition is estimated when the ratio of the prediction-error norm with a zero initial condition to the norm with an estimated initial condition exceeds `AutoInitialState`.

**Default:** `1.05`

## Output Arguments

### `opt`

Option set containing the specified options for `armax`.

## Examples

### Create Default Options Set for ARMAX Estimation

Create a default options set for `armax`.

```
opt = armaxOptions;
```

### Specify Options for ARMAX Estimation

Create an options set for `armax` using the `'stability'` for Focus and set the Display to `'on'`.

```
opt = armaxOptions('Focus','stability','Display','on');
```

Alternatively, use dot notation to set the values of `opt`.

```
opt = armaxOptions;
opt.Focus = 'stability';
opt.Display = 'on';
```

# armaxOptions

---

## References

[1] Wills, Adrian , B. Ninness, and S. Gibson. "On Gradient-Based Search for Multivariable System Estimates" IFAC World Congress, Prague, 2005.

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

## See Also

armax | idfilt



**Purpose** Option set for ar

**Syntax**  
opt = arOptions  
opt = arOptions(Name,Value)

**Description** opt = arOptions creates the default options set for ar.  
opt = arOptions(Name,Value) creates an option set with the options specified by one or more Name,Value pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments, where Name is the argument name and Value is the corresponding value. Name must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

### Approach

Technique used for AR model estimation.

Approach takes one of the following strings:

- 'fb' — Forward-backward approach.
- 'ls' — Least-squares method.
- 'yw' — Yule-Walker approach.
- 'burg' — Burg's method.
- 'gl' — Geometric lattice method.

**Default:** 'fb'

### Window

Data windowing technique.

Window determines how the data outside the measured time interval (past and future values) should be handled.

Window takes one of the following strings:

- 'now' — No windowing.
- 'prw' — Pre-windowing.
- 'pow' — Post-windowing.
- 'ppw' — Pre- and post-windowing.

**Default:** 'now' (except when Approach is 'yw')

### **DataOffset**

Data offset level that must be removed before estimation of parameters.

Specify **DataOffset** as a double scalar. For multi-experiment data, specify **DataOffset** as a vector of length  $N_e$ , where  $N_e$  is the number of experiments. Each entry of the vector is subtracted from the corresponding data.

**Default:** [ ] (no offsets)

### **MaxSize**

Specifies the maximum number of elements in a segment when input-output data is split into segments.

If larger matrices are needed, the software will use loops for calculations. Use this option to manage the trade-off between memory management and program execution speed. The original data matrix must be smaller than the matrix specified by **MaxSize**.

**MaxSize** must be a positive integer.

**Default:** 250000

## **Output Arguments**

### **opt**

Option set containing the specified options for ar.

**Examples****Create Default Options Set for AR Estimation**

Create a default options set for ar.

```
opt = arOptions;
```

**Specify Options for AR Estimation**

Create an options set for ar using the least squares algorithm for estimation and set the Window to 'ppw'.

```
opt = arOptions('Approach','ls','Window','ppw');
```

Alternatively, use dot notation to set the values of opt.

```
opt = arOptions;  
opt.Approach = 'ls';  
opt.Window = 'ppw';
```

**See Also** [ar](#)

**Purpose** Estimate parameters of ARX or AR model using least squares

**Syntax**

```
sys = arx(data,[na nb nk])  
sys = arx(data,[na nb nk],Name,Value)  
sys = arx(data,[na nb nk], ___,opt)
```

## Description

---

**Note** arx does not support continuous-time estimations. Use tfest instead.

---

`sys = arx(data,[na nb nk])` returns an ARX structure polynomial model, `sys`, with estimated parameters and covariances (parameter uncertainties) using the least-squares method and specified orders.

`sys = arx(data,[na nb nk],Name,Value)` estimates a polynomial model with additional options specified by one or more `Name,Value` pair arguments.

`sys = arx(data,[na nb nk], ___,opt)` specifies estimation options that configure the estimation objective, initial conditions and handle input/output data offsets.

## Input Arguments

### **data**

Estimation data.

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

### **[na nb nk]**

Polynomial orders.

[na nb nk] define the polynomial orders of an ARX model.

- `na` — Order of the polynomial  $A(q)$ .

Specify `na` as an  $N_y$ -by- $N_y$  matrix of nonnegative integers.  $N_y$  is the number of outputs.

- **nb** — Order of the polynomial  $B(q) + 1$ .  
nb is an  $N_y$ -by- $N_u$  matrix of nonnegative integers.  $N_y$  is the number of outputs and  $N_u$  is the number of inputs.
- **nk** — Input-output delay expressed as fixed leading zeros of the  $B$  polynomial.  
Specify nk as an  $N_y$ -by- $N_u$  matrix of nonnegative integers.  $N_y$  is the number of outputs and  $N_u$  is the number of inputs.

**opt**

Estimation options.

opt is an options set that specifies estimation options, including:

- input/output data offsets
- output weight

Use arxOptions to create the options set.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of **Name**, **Value** arguments, where **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

**InputDelay**

Input delays. **InputDelay** is a numeric vector specifying a time delay for each input channel. Specify input delays in integer multiples of the sampling period  $T_s$ . For example, **InputDelay** = 3 means a delay of three sampling periods.

For a system with  $N_u$  inputs, set **InputDelay** to an  $N_u$ -by-1 vector, where each entry is a numerical value representing the input delay for the corresponding input channel. You can also set **InputDelay** to a scalar value to apply the same delay to all channels.

**Default:** 0 for all input channels

**ioDelay**

Transport delays. `ioDelay` is a numeric array specifying a separate transport delay for each input/output pair.

Specify transport delays as integers denoting delay of a multiple of the sampling period  $T_s$ .

For a MIMO system with  $N_y$  outputs and  $N_u$  inputs, set `ioDelay` to a  $N_y$ -by- $N_u$  array, where each entry is a numerical value representing the transport delay for the corresponding input/output pair. You can also set `ioDelay` to a scalar value to apply the same delay to all input/output pairs. Useful as a replacement for the `nk` order, you can factor out  $\max(nk-1, 0)$  lags as the `ioDelay` value.

**Default:** 0 for all input/output pairs

**IntegrateNoise**

Logical vector specifying integrators in the noise channel.

`IntegrateNoise` is a logical vector of length  $N_y$ , where  $N_y$  is the number of outputs.

Setting `IntegrateNoise` to `true` for a particular output results in the ARIX model:

$$A(q)y(t) = B(q)u(t - nk) + \frac{1}{1 - q^{-1}}e(t)$$

Where,  $\frac{1}{1 - q^{-1}}$  is the integrator in the noise channel,  $e(t)$ .

**Default:** `false(Ny, 1)` ( $N_y$  is the number of outputs.)

**Output Arguments**

**sys**

Identified ARX structure polynomial model.

sys is a discrete-time idpoly model, which encapsulates the estimated  $A$  and  $B$  polynomials and the parameter covariance information.

## Definitions

### ARX structure

arx estimates the parameters of the ARX model structure:

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

The parameters  $n_a$  and  $n_b$  are the orders of the ARX model, and  $n_k$  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.
- $y(t-1)\dots y(t-n_a)$  — Previous outputs on which the current output depends.
- $u(t-n_k)\dots u(t-n_k-n_b+1)$  — Previous and delayed inputs on which the current output depends.
- $e(t-1)\dots e(t-n_c)$  — 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,

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

$$B(q) = b_1 + b_2q^{-1} + \dots + b_{n_b} q^{-n_b+1}$$

**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 ith element corresponds to the order and delay associated with the ith input.

$$y(t) + A_1y(t-1) + A_2y(t-2) + \dots + A_{na}y(t-na) = B_0u(t) + B_1u(t-1) + \dots + B_{nb}u(t-nb) + e(t)$$

**Multi-Output 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



```
opt = arxOptions('OutputWeight', inv(lambda))  
m = arx(data, orders, opt)
```

### 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.

Set the `InitialCondition` estimation option (see `arxOptions`) 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.

### 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]);
```

### Algorithms

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.

## See Also

`arxOptions` | `arxstruc` | `ar` | `arimax` | `bj` | `iv4` | `n4sid` | `oe` | `nlarx`

## How To

- “Using Linear Model for Nonlinear ARX Estimation”

**Purpose**

ARX parameters from multiple-output models with variance information

---

**Note** arxdata will be removed in a future release. Use polydata instead.

---

**Syntax**

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

**Arguments**

m  
 An idarx model object.

Also accepts single-output 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  $n_y$  is the number of outputs (the dimension of the vector  $y(t)$ ) and  $n_u$  is the number of inputs.

A is an  $n_y$ -by- $n_y$ -by- $(n_a+1)$  array such that

$A(:,:,k+1) = A_k$   
 $A(:,:,1) = \text{eye}(n_y)$

where  $k=0,1,\dots,n_a$ .

B is an  $n_y$ -by- $n_u$ -by- $(n_b+1)$  array with

$B(:,:,k+1) = B_k$

$A(0)$  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,  $B = []$ .

---

$[A, B, dA, dB] = \text{arxdata}(m)$  returns  $A$  and  $B$  matrices, and  $dA$  and  $dB$  as the estimated standard deviations of  $A$  and  $B$ , respectively.

## Tips

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

$$y(t) + A_1 y(t-1) + A_2 y(t-2) + \dots + A_{na} y(t-na) = B_0 u(t) + B_1 u(t-1) + \dots + B_{nb} u(t-nb) + e(t)$$

where  $A_k$  and  $B_k$  matrices have dimensions  $ny$ -by- $ny$  and  $ny$ -by- $nu$ , respectively.  $ny$  is the number of outputs (the dimension of the vector  $y(t)$ ) and  $nu$  is the number of inputs.

## See Also

`idarx` | `idpoly`

**Purpose** Option set for ar

**Syntax**  
opt = arxOptions  
opt = arxOptions(Name,Value)

**Description** opt = arxOptions creates the default options set for arx.  
opt = arxOptions(Name,Value) creates an option set with the options specified by one or more Name,Value pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments, where Name is the argument name and Value is the corresponding value. Name must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

### InitialCondition

Specify handling of initial conditions during estimation.

InitialCondition takes one of the following values:

- 'zero' — Initial conditions are set to 0.
- 'estimate' — Initial conditions, along with the model parameters, are treated as estimation parameters.
- 'backcast' — Initial conditions are estimated using a backcasting (backward filtering) process.
- 'auto' — The software chooses the initial condition handling method based on the estimation data.

**Default:** 'auto'

### Focus

Defines how the errors  $e$  between the measured and the modeled outputs are weighed at specific frequencies during the minimization of the prediction error.

Higher weighting at specific frequencies emphasizes the requirement for a good fit at these frequencies.

**Focus** can take the following values:

- '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.

This method provides a stable model.

- 'prediction' — 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.
- 'stability' — Same as 'prediction', but with model stability enforced.
- Passbands — Row vector or matrix containing frequency values that define desired passbands. For example:

```
[w1,wh]  
[w11,w1h;w21,w2h;w31,w3h;...]
```

where  $w1$  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.

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

- A single-input-single-output (SISO) linear system.
- The {A,B,C,D} format, which specifies the state-space matrices of the filter.
- The {numerator, denominator} format, 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. To obtain a good model fit for a specific frequency range, you must choose the filter with a passband in this range. The estimation result is the same if you first prefilter the data using `idfilt`.

- Weighting vector — 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.

**Default:** 'prediction'

### **EstCovar**

Controls whether parameter covariance data is generated or not.

If `EstCovar` is `true`, then use `getcov` to fetch the covariance matrix from the estimated model.

**Default:** `true`

### **Display**

Specifies whether estimation progress should be displayed.

`Display` requires one of the following strings:

- 'on' — Information on model structure and estimation results are displayed in a progress viewer window
- 'off' — No progress or results information is displayed

**Default:** 'off'

## **InputOffset**

Removes offset from time domain input data during estimation.

Specify as a column vector of length  $Nu$ , where  $Nu$  is the number of inputs.

Use [] to indicate no offset.

For multiexperiment data, specify **InputOffset** as a  $Nu$ -by- $Ne$  matrix.  $Nu$  is the number of inputs, and  $Ne$  is the number of experiments.

Each entry specified by **InputOffset** is subtracted from the corresponding input data.

**Default:** []

## **OutputOffset**

Removes offset from time domain output data during estimation.

Specify as a column vector of length  $Ny$ , where  $Ny$  is the number of outputs.

Use [] to indicate no offset.

For multiexperiment data, specify **OutputOffset** as a  $Ny$ -by- $Ne$  matrix.  $Ny$  is the number of outputs, and  $Ne$  is the number of experiments. Each entry specified by **OutputOffset** is subtracted from the corresponding output data.

**Default:** []

## **OutputWeight**

Weight of prediction errors in multi-output estimation.

Specify **OutputWeight** as a positive semi-definite symmetric matrix ( $W$ ). The software will minimize the trace of the weighted prediction error matrix  $\text{trace}(E' * E * W)$ . Where  $E$  is the matrix of prediction errors,



with one column for each output, and  $W$  is the positive semi-definite symmetric matrix of size equal to the number of outputs. Use  $W$  to specify the relative importance of outputs in multiple-input multiple-output models, or the reliability of corresponding data.

This option is relevant only for multi-output models.

### Advanced

Advanced is a structure with the following fields:

- **MaxSize** — Specifies the maximum number of elements in a segment when input-output data is split into segments.

MaxSize must be a positive integer.

**Default:** 250000

- **StabilityThreshold** — Specifies thresholds for stability tests.

StabilityThreshold is a structure with the following fields:

- **s** — Specifies the location of the right-most pole to test the stability of continuous-time models. A model is considered stable when its right-most pole is to the left of  $s$ .

**Default:** 0

- **z** — 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  $z$  from the origin.

**Default:**  $1 + \sqrt{\text{eps}}$

## Output Arguments

**opt**

Option set containing the specified options for arx.

## Examples

### Create Default Options Set for ARX Estimation

Create a default options set for arx.

```
opt = arxOptions;
```

## Specify Options for ARX Estimation

Create an options set for arx using zero initial conditions for estimation and set Display to 'on'.

```
opt = arxOptions('InitialCondition','zero','Display','on');
```

Alternatively, use dot notation to set the values of opt.

```
opt = arxOptions;  
opt.InitialCondition = 'zero';  
opt.Display = 'on';
```

## See Also

arx | idfilt

<b>Purpose</b>	Compute and compare loss functions for single-output ARX models
<b>Syntax</b>	<pre>V = arxstruc(ze,zv,NN) V = arxstruc(ze,zv,NN,maxsize)</pre>
<b>Arguments</b>	<p><b>ze</b> Estimation data set can be <code>iddata</code> or <code>idfrd</code> object.</p> <p><b>zv</b> Validation data set can be <code>iddata</code> or <code>idfrd</code> object.</p> <p><b>NN</b> Matrix defines the number of different ARX-model structures. Each row of <b>NN</b> is of the form:</p> <pre>nn = [na nb nk]</pre> <p><b>maxsize</b> Specifies the maximum number of elements in a segment when input-output data is split into segments.</p> <p>If larger matrices are needed, the software will use loops for calculations. Use this option to manage the trade-off between memory management and program execution speed. The original data matrix must be smaller than the matrix specified by <b>maxsize</b>.</p> <p><b>maxsize</b> must be a positive integer.</p>

## 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.

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`.

## Tips

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

This example uses the simulation data from a second-order `idpoly` model with additive noise. The data is split into two parts, where one part is the estimation data and the other is the validation data. You select the best model by comparing the output of models with orders ranging between 1 and 5 with the validating data. All models have an input-to-output delay of 1.

```
% Create an ARX model for generating data:
A = [1 -1.5 0.7]; B = [0 1 0.5];
m0 = idpoly(A,B);
% Generate a random input signal:
u = iddata([],idinput(400,'rbs'));
e = iddata([],0.1*randn(400,1));
% Simulate the output signal from the model m0:
y = sim(m0, [u e]);
z = [y,u]; % analysis data
NN = struc(1:5,1:5,1);
V = arxstruc(z(1:200),z(201:400),NN);
nn = selstruc(V,0);
```

```
m = arx(z,nn);
```

```
arx | idpoly | ivstruc | selstruc | struc
```

# bandwidth

---

**Purpose** Frequency response bandwidth

**Syntax**  
`fb = bandwidth(sys)`  
`fb = bandwidth(sys,dbdrop)`

**Description** `fb = bandwidth(sys)` computes the bandwidth `fb` of the SISO dynamic system model `sys`, defined as the first frequency where the gain drops below 70.79 percent (-3 dB) of its DC value. The frequency `fb` is expressed in `rad/TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`.

For FRD models, `bandwidth` uses the first frequency point to approximate the DC gain.

`fb = bandwidth(sys,dbdrop)` further specifies the critical gain drop in dB. The default value is -3 dB, or a 70.79 percent drop.

If `sys` is an `S1-by...-by- $S_p$`  array of models, `bandwidth` returns an array of the same size such that

`fb(j1,...,jp) = bandwidth(sys(:, :, j1, ..., jp))`

**See Also** `dcgain` | `issiso`

---

<b>Purpose</b>	Estimate Box-Jenkins polynomial model using time domain data
<b>Syntax</b>	<pre>sys = bj(data, [nb nc nd nf nk]) sys = bj(data,[nb nc nd nf nk], Name,Value) sys = bj(data, init_sys) sys = bj(data, ___, opt)</pre>
<b>Description</b>	<p><code>sys = bj(data, [nb nc nd nf nk])</code> estimates a Box-Jenkins polynomial model, <code>sys</code>, using the time domain data, <code>data</code>. <code>[nb nc nd nf nk]</code> define the orders of the polynomials used for estimation.</p> <p><code>sys = bj(data,[nb nc nd nf nk], Name,Value)</code> estimates a polynomial model with additional options specified by one or more <code>Name,Value</code> pair arguments.</p> <p><code>sys = bj(data, init_sys)</code> estimates a Box-Jenkins polynomial using the polynomial model <code>init_sys</code> to configure the initial parameterization of <code>sys</code>.</p> <p><code>sys = bj(data, ___, opt)</code> estimates a Box-Jenkins polynomial using the option set, <code>opt</code>, to specify estimation behavior.</p>
<b>Input Arguments</b>	<p><b>data</b> Estimation data. <code>data</code> is an <code>iddata</code> object containing the input and output signal values.</p> <p><b>[nb nc nd nf nk]</b> A vector of matrices containing the orders and delays of the Box-Jenkins model. Matrixes must contain nonnegative integers.</p> <ul style="list-style-type: none"><li>• <code>nb</code> is the order of the B polynomial plus 1 (<math>N_y</math>-by-<math>N_u</math> matrix)</li><li>• <code>nc</code> is the order of the C polynomial plus 1 (<math>N_y</math>-by-1 matrix)</li><li>• <code>nd</code> is the order of the D polynomial plus 1 (<math>N_y</math>-by-1 matrix)</li><li>• <code>nf</code> is the order of the F polynomial plus 1 (<math>N_y</math>-by-<math>N_u</math> matrix)</li></ul>

- `nk` is the input delay (in number of samples,  $N_y$ -by- $N_u$  matrix) where  $N_u$  is the number of inputs and  $N_y$  is the number of outputs.

**opt**

Estimation options.

`opt` is an options set that configures, among others, the following:

- estimation objective
- initial conditions
- numerical search method to be used in estimation

Use `bjOptions` to create the options set.

**init\_sys**

Polynomial model that configures the initial parameterization of `sys`.

`init_sys` must be an `idpoly` model with the Box-Jenkins structure that has only  $B$ ,  $C$ ,  $D$  and  $F$  polynomials active. `bj` uses the parameters and constraints defined in `init_sys` as the initial guess for estimating `sys`.

Use the `Structure` property of `init_sys` to configure initial guesses and constraints for  $B(q)$ ,  $F(q)$ ,  $C(q)$  and  $D(q)$ .

To specify an initial guess for, say, the  $C(q)$  term of `init_sys`, set `init_sys.Structure.c.Value` as the initial guess.

To specify constraints for, say, the  $B(q)$  term of `init_sys`:

- set `init_sys.Structure.b.Minimum` to the minimum  $B(q)$  coefficient values
- set `init_sys.Structure.b.Maximum` to the maximum  $B(q)$  coefficient values
- set `init_sys.Structure.b.Free` to indicate which  $B(q)$  coefficients are free for estimation



You can similarly specify the initial guess and constraints for the other polynomials.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **InputDelay**

Input delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. Specify input delays in integer multiples of the sampling period  $T_s$ . For example, `InputDelay = 3` means a delay of three sampling periods.

For a system with  $N_u$  inputs, set `InputDelay` to an  $N_u$ -by-1 vector, where each entry is a numerical value representing the input delay for the corresponding input channel. You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0 for all input channels

### **ioDelay**

Transport delays. `ioDelay` is a numeric array specifying a separate transport delay for each input/output pair.

Specify transport delays as integers denoting delay of a multiple of the sampling period  $T_s$ .

For a MIMO system with  $N_y$  outputs and  $N_u$  inputs, set `ioDelay` to a  $N_y$ -by- $N_u$  array, where each entry is a numerical value representing the transport delay for the corresponding input/output pair. You can also set `ioDelay` to a scalar value to apply the same delay to all input/output pairs.

**Default:** 0 for all input/output pairs

**IntegrateNoise**

Logical specifying integrators in the noise channel.

`IntegrateNoise` is a logical vector of length  $N_y$ , where  $N_y$  is the number of outputs.

Setting `IntegrateNoise` to `true` for a particular output results in the model:

$$y(t) = \frac{B(q)}{F(q)}u(t - nk) + \frac{C(q)}{D(q)} \frac{e(t)}{1 - q^{-1}}$$

Where,  $\frac{1}{1 - q^{-1}}$  is the integrator in the noise channel,  $e(t)$ .

**Default:** `false(Ny, 1)` ( $N_y$  is the number of outputs)

**Output Arguments**

**sys**

Identified polynomial model of Box-Jenkins structure.

`sys` is a discrete-time `idpoly` model which encapsulates the identified polynomial model.

**Definitions**

**Box-Jenkins Model Structure**

The general Box-Jenkins model structure is:

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

where  $nu$  is the number of input channels.

The orders of Box-Jenkins model are defined as follows:

$$nb: B(q) = b_1 + b_2q^{-1} + \dots + b_{nb}q^{-nb+1}$$

$$nc: C(q) = 1 + c_1q^{-1} + \dots + c_{nc}q^{-nc}$$

$$nd: D(q) = 1 + d_1q^{-1} + \dots + d_{nd}q^{-nd}$$

$$nf: F(q) = 1 + f_1q^{-1} + \dots + f_{nf}q^{-nf}$$

## Examples

### Identify SISO Box-Jenkins Model

Estimate the parameters of a single-input, single-output Box-Jenkins model from measured data.

```
load iddata1 z1;
nb = 2;
nc = 2;
nd = 2;
nf = 2;
nk = 1;
sys = bj(z1,[nb nc nd nf nk])
```

sys is a discrete-time `idpoly` model with estimated coefficients. The order of sys is as described by nb, nc, nd, nf, and nk.

Use `getpvec` to obtain the estimated parameters and `getcov` to obtain the covariance associated with the estimated parameters.

### Estimate a Multi-Input, Single-Output Box-Jenkins Model

Estimate the parameters of a multi-input, single-output Box-Jenkins model from measured data.

```
load iddata8;
nb = [2 1 1];
nc = 1;
nd = 1;
nf = [2 1 2];
nk = [5 10 15];
sys = bj(z8,[nb nc nd nf nk]);
```

sys estimates the parameters of a model with three inputs and one output. Each of the inputs has a delay associated with it.

### **Configure Estimation Options**

Estimate the parameters of a single-input, single-output Box-Jenkins model while configuring some estimation options.

Generate estimation data.

```
B = [0 1 0.5];  
C = [1 -1 0.2];  
D = [1 1.5 0.7];  
F = [1 -1.5 0.7];  
sys0 = idpoly(1,B,C,D,F,0.1);  
e = iddata([],randn(200,1));  
u = iddata([],idinput(200));  
y = sim(sys0,[u e]);  
data = [y u];
```

data is a single-input, single-output data set created by simulating a known model.

Estimate initial Box-Jenkins model.

```
nb = 2;  
nc = 2;  
nd = 2;  
nf = 2;  
nk = 1;  
init_sys = bj(data,[2 2 2 2 1]);
```

Create an estimation option set to refine the parameters of the estimated model.

```
opt = bjOptions;  
opt.Display = 'on';  
opt.SearchOption.MaxIter = 50;
```

`opt` is an estimation option set that configures the estimation to iterate 50 times at most and display the estimation progress.

Reestimate the model parameters using the estimation option set.

```
sys = bj(data,init_sys,opt)
```

`sys` is estimated using `init_sys` for the initial parameterization for the polynomial coefficients.

To view the estimation result, enter `sys.Report` at the MATLAB command prompt.

### **Estimate MIMO Box-Jenkins Model**

Estimate a multi-input, multi-output Box-Jenkins model from estimated data.

Load measured data.

```
load iddata1 z1
load iddata2 z2
data = [z1, z2(1:300)];
```

`data` contains the measured data for two inputs and two outputs.

Estimate the model.

```
nb = [2 2; 3 4];
nc = [2;2];
nd = [2;2];
nf = [1 0; 2 2];
nk = [1 1; 0 0];
sys = bj(data, [nb nc nd nf nk])
```

The polynomial order coefficients contain one row for each output.

`sys` is a discrete-time `idpoly` model with two inputs and two outputs.

### **Alternatives**

To estimate a continuous-time model, use:

- `tfest` — returns a transfer function model
- `ssest` — returns a state-space model
- `bj` to first estimate a discrete-time model and convert it a continuous-time model using `d2c`.

## References

[1] Ljung, L. *System Identification: Theory for the User*, Upper Saddle River, NJ, Prentice-Hal PTR, 1999.

## See Also

`bjoptions` | `tfest` | `arx` | `armax` | `iv4` | `ssest` | `oe` |  
`polyest` | `idpoly` | `iddata` | `d2c` | `forecast` | `sim` | `compare`

## Purpose

Option set for bj

## Syntax

```
opt = bjOptions
opt = bjOptions(Name,Value)
```

## Description

`opt = bjOptions` creates the default options set for bj.

`opt = bjOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### InitialCondition

Specify how initial conditions are handled during estimation.

`InitialCondition` takes one of the following values:

- 'zero' — The initial conditions are set to zero.
- 'estimate' — The initial conditions are treated as independent estimation parameters.
- 'backcast' — The initial conditions are estimated using the best least squares fit.
- 'auto' — The software chooses the method to handle initial conditions based on the estimation data.

**Default:** 'auto'

### Focus

Defines how the errors  $e$  between the measured and the modeled outputs are weighed at specific frequencies during the minimization of the prediction error.

Higher weighting at specific frequencies emphasizes the requirement for a good fit at these frequencies.

**Focus** can take the following values:

- '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.

This method provides a stable model.

- prediction — Automatically calculates the weighting function as a product of the input spectrum and the inverse of the noise model. The weighting function 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. Use 'stability' when you want to ensure a stable model.
- 'stability' — Same as 'prediction', but with model stability enforced.
- Passbands — Row vector or matrix containing frequency values that define desired passbands. For example:

```
[w1,wh]  
[w11,w1h;w21,w2h;w31,w3h;...]
```

where  $w1$  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.

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



- A single-input-single-output (SISO) linear system.
- The {A,B,C,D} format, which specifies the state-space matrices of the filter.
- The {numerator, denominator} format, 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. To obtain a good model fit for a specific frequency range, you must choose the filter with a passband in this range. The estimation result is the same if you first prefilter the data using `idfilt`.

- Weighting vector — 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.

**Default:** 'prediction'

### **EstCovar**

Controls whether parameter covariance data is generated or not.

If `EstCovar` is `true`, then use `getcov` to fetch the covariance matrix from the estimated model.

**Default:** `true`

### **Display**

Specifies whether estimation progress should be displayed.

`Display` requires one of the following strings:

- 'on' — Information on model structure and estimation results are displayed in a progress viewer window
- 'off' — No progress or results information is displayed

**Default:** 'off'

## **InputOffset**

Removes offset from time domain input data during estimation.

Specify as a column vector of length  $Nu$ , where  $Nu$  is the number of inputs.

Use [] to indicate no offset.

For multiexperiment data, specify **InputOffset** as a  $Nu$ -by- $Ne$  matrix.  $Nu$  is the number of inputs, and  $Ne$  is the number of experiments.

Each entry specified by **InputOffset** is subtracted from the corresponding input data.

**Default:** []

## **OutputOffset**

Removes offset from time domain output data during estimation.

Specify as a column vector of length  $Ny$ , where  $Ny$  is the number of outputs.

Use [] to indicate no offset.

For multiexperiment data, specify **OutputOffset** as a  $Ny$ -by- $Ne$  matrix.  $Ny$  is the number of outputs, and  $Ne$  is the number of experiments. Each entry specified by **OutputOffset** is subtracted from the corresponding output data.

**Default:** []

## **SearchMethod**

Search method used for iterative parameter estimation.

**SearchMethod** is a string that can take the following values:

- gn — The subspace Gauss-Newton direction.

- `gna` — An adaptive version of subspace Gauss-Newton approach, suggested by Wills and Ninness [1].
- `lm` — Uses the Levenberg-Marquardt method.
- `lsqnonlin` — Uses the trust region reflective algorithm. Requires the Optimization Toolbox software.
- `grad` — The steepest descent gradient search method.
- `auto` — A choice among the above is made in the algorithm. The descent direction is calculated using `gn`, `gna`, `lm` and `grad` successively, in that order, at each iteration until a sufficient reduction in error is achieved.

**Default:** 'auto'

### SearchOption

`SearchOption` is an options set for the search algorithm with the fields:

SearchMethod	SearchOption
gn,gna,lm,grad and auto	<ul style="list-style-type: none"> <li>• <b>Tolerance</b> — 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 <b>Tolerance</b>, the iterations are stopped. 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.  <b>Default:</b> 0.01</li> <li>• <b>MaxIter</b> — Maximum number of iterations during loss-function minimization. The iterations stop when <b>MaxIter</b> is reached or another stopping criterion is satisfied, such as <b>Tolerance</b>.  Setting <b>MaxIter</b> = 0 returns the result of the startup procedure.</li> </ul>

## SearchMethodOption

Use `sys.Report.Termination.Iterations` to get the actual number of iterations during an estimation, where `sys` is an `idtf` model.

**Default:** 20

- Advanced — Search settings:
  - `GnPinvConst` — Singular values of the Jacobian that are smaller than `GnPinvConst*max(size(J)*norm(J)*eps)` are discarded when computing the search direction and `SearchMethod` is `'gn'`.

`GnPinvConst` must be a positive real value.

**Default:** 10000

- `InitGnaTol` — Initial value of *gamma* when `SearchMethod` is `'gna'`.

**Default:** .0001

- `LMStartValue` — Starting value of search-direction length *d* in the Levenberg-Marquardt method. Applicable when `SearchMethod` is `'lm'`.

**Default:** .001

- `LMStep` — 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. Applicable when `SearchMethod` is `'lm'`.

**Default:** 2

- `MaxBisections` — Maximum number of bisections used by the line search along the search direction.

**Default:** 25

SearchMethod	SearchOption
	<ul style="list-style-type: none"> <li data-bbox="630 361 1313 418"> <p>▪ <b>MaxFunEvals</b> — Iterations are stopped if the number of calls to the model file exceeds this value.</p> <p>MaxFunEvals must be a positive integer value.</p> <p><b>Default:</b> Inf</p> </li> <li data-bbox="630 534 1313 591"> <p>▪ <b>MinParChange</b> — Smallest parameter update allowed per iteration</p> <p>MinParChange must be a positive, real value.</p> <p><b>Default:</b> 0</p> </li> <li data-bbox="630 708 1313 765"> <p>▪ <b>RelImprovement</b> — Iterations are stopped if the relative improvement of the criterion function is less than RelImprovement.</p> <p>RelImprovement must be a positive integer value.</p> <p><b>Default:</b> 0</p> </li> <li data-bbox="630 881 1313 939"> <p>▪ <b>StepReduction</b> — 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.</p> <p>StepReduction must be a positive, real value that is greater than 1.</p> <p><b>Default:</b> 2</p> </li> </ul>
lsqnonlin	<ul style="list-style-type: none"> <li data-bbox="630 1194 1313 1251"> <p>▪ <b>TolFun</b> — Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values.</p> <p>The value of TolFun is the same as that of <code>sys.SearchOption.Advanced.TolFun</code>.</p> <p><b>Default:</b> 1e-5</p> </li> </ul>

SearchMethod	SearchOption
	<ul style="list-style-type: none"><li>• <b>TolX</b> — Termination tolerance on the estimated parameter values. The value of TolX is the same as that of <code>sys.SearchOption.Advanced.TolX</code>. <b>Default:</b> 1e-6</li><li>• <b>MaxIter</b> — Maximum number of iterations during loss-function minimization. The iterations stop when MaxIter is reached or another stopping criterion is satisfied, such as TolFun etc.. The value of MaxIter is the same as that of <code>sys.SearchOption.Advanced.MaxIter</code>. <b>Default:</b> 20</li><li>• <b>Advanced</b> — Options set for <code>lsqnonlin</code>. For more information, see “Optimization Options”. Use <code>optimset('lsqnonlin')</code> to create an options set for <code>lsqnonlin</code> and then modify it to specify its various options.</li></ul>

## Advanced

Advanced is a structure with the following fields:

- **ErrorThreshold** — Specifies when to adjust the weight of large errors from quadratic to linear.

Errors larger than `ErrorThreshold` times the estimated standard deviation have a linear weight in the 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 the chapter “Computing the Estimate” of [2]. `ErrorThreshold = 0` disables robustification and leads to a purely quadratic criterion. When estimating with frequency-domain data,

ErrorThreshold is set to zero. ErrorThreshold = 1.6 is a good choice for estimation when data contains outliers.

**Default:** 0

- **MaxSize** — Specifies the maximum number of elements in a segment when input-output data is split into segments.

MaxSize must be a positive integer.

**Default:** 250000

- **StabilityThreshold** — Specifies thresholds for stability tests.

StabilityThreshold is a structure with the following fields:

- **s** — Specifies the location of the right-most pole to test the stability of continuous-time models. A model is considered stable when its right-most pole is to the left of s.

**Default:** 0

- **z** — 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 z from the origin.

**Default:** 1+sqrt(eps)

- **AutoInitThreshold** — Specifies when to automatically estimate the initial conditions.

When InitialCondition = 'Auto', the initial condition is estimated when the ratio of the prediction-error norm with a zero initial condition to the norm with an estimated initial condition exceeds AutoInitialState.

**Default:** 1.05

## Output Arguments

**opt**

Option set containing the specified options for bj.

## References

[1] Wills, Adrian , B. Ninness, and S. Gibson. "On Gradient-Based Search for Multivariable System Estimates" IFAC World Congress, Prague, 2005.

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

## Examples

### Create Default Options Set for Box-Jenkins Estimation

Create a default options set for bj.

```
opt = bjOptions;
```

### Specify Options for Box-Jenkins Estimation

Create an options set for bj using zero initial conditions for estimation and set Display to 'on'.

```
opt = bjOptions('InitialCondition','zero','Display','on');
```

Alternatively, use dot notation to set the values of opt.

```
opt = bjOptions;  
opt.InitialCondition = 'zero';  
opt.Display = 'on';
```

## See Also

bj | idfilt



**Purpose** Block-diagonal concatenation of models

**Syntax** `sys = blkdiag(sys1,sys2,...,sysN)`

**Description** `sys = blkdiag(sys1,sys2,...,sysN)` produces the aggregate system

$$\begin{bmatrix} \text{sys1} & 0 & \dots & 0 \\ 0 & \text{sys2} & \dots & \vdots \\ \vdots & \dots & \dots & 0 \\ 0 & \dots & 0 & \text{sysN} \end{bmatrix}$$

blkdiag is equivalent to append.

**Examples** The commands

```
sys1 = tf(1,[1 0]);
sys2 = ss(1,2,3,4);
sys = blkdiag(sys1,10,sys2)
```

produce the state-space model

```
a =
      x1  x2
x1    0   0
x2    0   1

b =
      u1  u2  u3
x1    1   0   0
x2    0   0   2

c =
      x1  x2
y1    1   0
y2    0   0
y3    0   3
```

# blkdiag

---

```
d =
      u1  u2  u3
y1    0   0   0
y2    0  10   0
y3    0   0   4
```

Continuous-time model.

## See Also

[append](#) | [series](#) | [parallel](#) | [feedback](#)

**Purpose**

Bode plot of frequency response, magnitude and phase of frequency response

**Syntax**

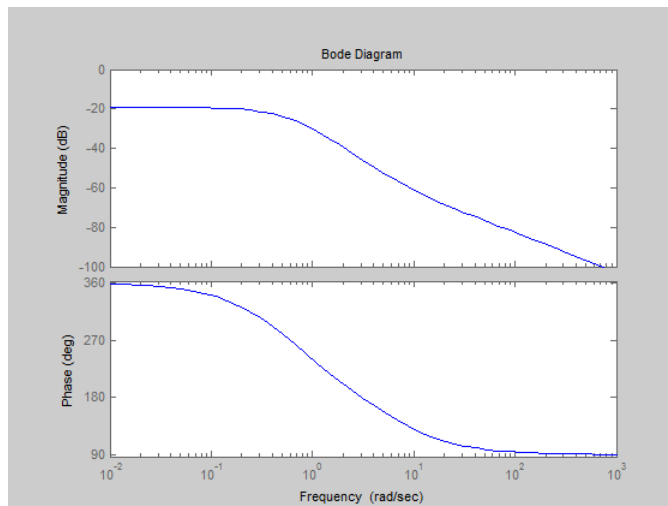
```
bode(sys)
bode(sys1,...,sysN)
bode(sys1,PlotStyle1,...,sysN,PlotStyleN)
bode(...,w)
[mag,phase] = bode(sys,w)
[mag,phase,wout] = bode(sys)
[mag,phase,wout,sdmag,sdphase] = bode(sys)
```

**Description**

`bode(sys)` creates a Bode plot of the frequency response of a dynamic system model `sys`. The plot displays the magnitude (in dB) and phase (in degrees) of the system response as a function of frequency.

When `sys` is a multi-input, multi-output (MIMO) model, `bode` produces an array of Bode plots, each plot showing the frequency response of one I/O pair.

`bode` automatically determines the plot frequency range based on system dynamics.



# bode

---

`bode(sys1, ..., sysN)` plots the frequency response of multiple dynamic systems in a single figure. All systems must have the same number of inputs and outputs.

`bode(sys1, PlotStyle1, ..., sysN, PlotStyleN)` plots system responses using the color, linestyle, and markers specified by the `PlotStyle` strings.

`bode(..., w)` plots system responses at frequencies determined by `w`.

- If `w` is a cell array `{wmin, wmax}`, `bode(sys, w)` plots the system response at frequency values in the range `{wmin, wmax}`.
- If `w` is a vector of frequencies, `bode(sys, w)` plots the system response at each of the frequencies specified in `w`.

`[mag, phase] = bode(sys, w)` returns magnitudes `mag` in absolute units and phase values `phase` in degrees. The response values in `mag` and `phase` correspond to the frequencies specified by `w` as follows:

- If `w` is a cell array `{wmin, wmax}`, `[mag, phase] = bode(sys, w)` returns the system response at frequency values in the range `{wmin, wmax}`.
- If `w` is a vector of frequencies, `[mag, phase] = bode(sys, w)` returns the system response at each of the frequencies specified in `w`.

`[mag, phase, wout] = bode(sys)` returns magnitudes, phase values, and frequency values `wout` corresponding to `bode(sys)`.

`[mag, phase, wout, sdmag, sdphase] = bode(sys)` additionally returns the estimated standard deviation of the magnitude and phase values when `sys` is an identified model and `[]` otherwise.

## Input Arguments

### **sys**

Dynamic system model, such as a Numeric LTI model, or an array of such models.

### **PlotStyle**

Line style, marker, and color of both the line and marker, specified as a one-, two-, or three-part string enclosed in single quotes ( ' ' ). The elements of the string can appear in any order. The string can specify only the line style, the marker, or the color.

For more information about configuring the `PlotStyle` string, see “Colors, Line Styles, and Markers” in the MATLAB documentation.

### **w**

Input frequency values, specified as a row vector or a two-element cell array.

Possible values of `w`:

- Two-element cell array `{wmin,wmax}`, where `wmin` is the minimum frequency value and `wmax` is the maximum frequency value.
- Row vector of frequency values.

For example, use `logspace` to generate a row vector with logarithmically-spaced frequency values.

Specify frequency values in radians per `TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`.

## **Output Arguments**

### **mag**

Bode magnitude of the system response in absolute units, returned as a 3-D array with dimensions (number of outputs) × (number of inputs) × (number of frequency points).

- For a single-input, single-output (SISO) `sys`, `mag(1,1,k)` gives the magnitude of the response at the `k`th frequency.
- For MIMO systems, `mag(i,j,k)` gives the magnitude of the response from the `j`th input to the `i`th output.

You can convert the magnitude from absolute units to decibels using:

$$\text{magdb} = 20 \cdot \log_{10}(\text{mag})$$

## **phase**

Phase of the system response in degrees, returned as a 3-D array with dimensions are (number of outputs)  $\times$  (number of inputs)  $\times$  (number of frequency points).

- For SISO `sys`, `phase(1,1,k)` gives the phase of the response at the `k`th frequency.
- For MIMO systems, `phase(i,j,k)` gives the phase of the response from the `j`th input to the `i`th output.

## **wout**

Response frequencies, returned as a row vector of frequency points. Frequency values are in radians per `TimeUnit`, where `TimeUnit` is the value of the `TimeUnit` property of `sys`.

## **sdmag**

Estimated standard deviation of the magnitude. `sdmag` has the same dimensions as `mag`.

If `sys` is not an identified LTI model, `sdmag` is `[]`.

## **sdphase**

Estimated standard deviation of the phase. `sdphase` has the same dimensions as `phase`.

If `sys` is not an identified LTI model, `sdphase` is `[]`.

## **Examples**

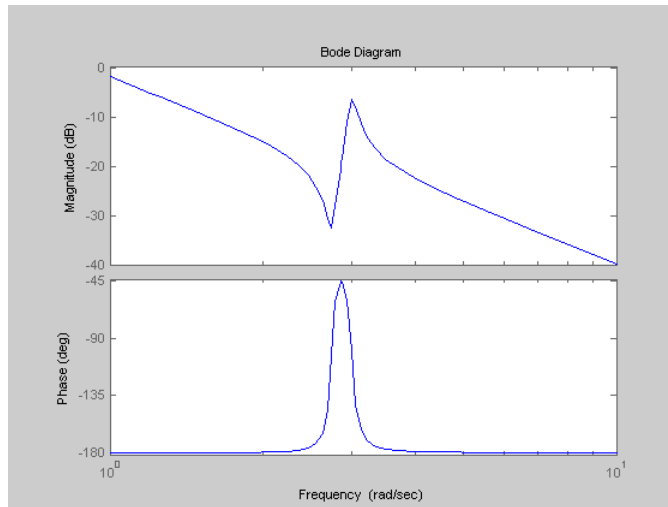
### **Bode Plot of Dynamic System**

Create Bode plot of the dynamic system:

$$H(s) = \frac{s^2 + 0.1s + 7.5}{s^4 + 0.12s^3 + 9s^2}$$

$H(s)$  is a continuous-time SISO system.

```
H = tf([1 0.1 7.5],[1 0.12 9 0 0]);  
bode(H)
```



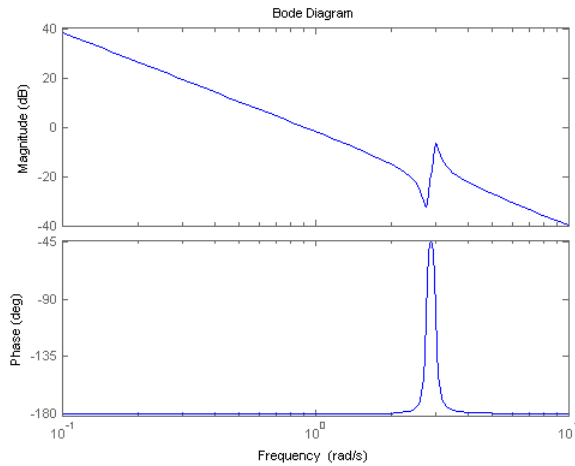
bode automatically selects the plot range based on the system dynamics.

### Bode Plot at Specified Frequencies

Create Bode plot over a specified frequency range. Use this approach when you want to focus on the dynamics in a particular range of frequencies.

```
H = tf([1 0.1 7.5],[1 0.12 9 0 0]);  
bode(H, {0.1, 10})
```

The cell array {0.1, 10} specifies the minimum and maximum frequency values in the Bode plot.



Alternatively, you can specify a vector of frequencies to use for evaluating and plotting the frequency response.

```
w = logspace(-1,1,50);  
bode(H,w)
```

`logspace` defines a logarithmically spaced frequency vector in the range of 0.1-10 rad/s.

---

## Compare Bode Plots of Several Dynamic Systems

Compare the frequency response of a continuous-time system to an equivalent discretized system on the same Bode plot.

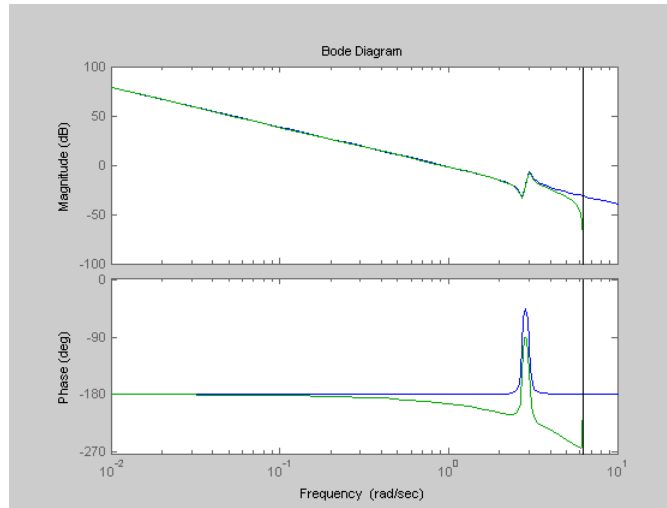
**1** Create continuous-time and discrete-time dynamic systems.

```
H = tf([1 0.1 7.5],[1 0.12 9 0 0]);  
Hd = c2d(H,0.5,'zoh');
```

**2** Create Bode plot that includes both systems.



bode(H,Hd)



### Bode Plot with Specified Line and Marker Attributes

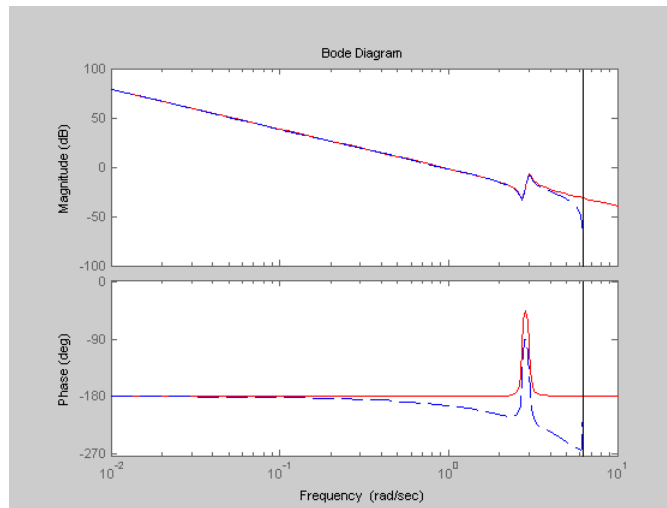
Specify the color, linestyle, or marker for each system in a Bode plot using the `PlotStyle` input arguments.

```
H = tf([1 0.1 7.5],[1 0.12 9 0 0]);
Hd = c2d(H,0.5,'zoh');
```

H and Hd are two different systems.

```
bode(H,'r',Hd,'b--')
```

The string `'r'` specifies a solid red line for the response of H. The string `'b--'` specifies a dashed blue line for the response of Hd.



## Obtain Magnitude and Phase Data

Compute the magnitude and phase of the frequency response of a dynamic system.

```
H = tf([1 0.1 7.5],[1 0.12 9 0 0]);  
[mag phase wout] = bode(H);
```

Because H is a SISO model, the first two dimensions of mag and phase are both 1. The third dimension is the number of frequencies in wout.

## Bode Plot of Identified Model

Compare the frequency response of a parametric model, identified from input/output data, to a non-parametric model identified using the same data.

1 Identify parametric and non-parametric models based on data.

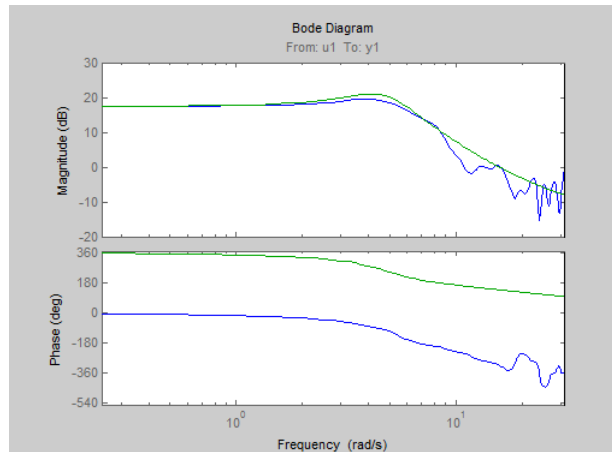
```
load iddata2 z2;
```

```
w = linspace(0,10*pi,128);
sys_np = spa(z2,[],w);
sys_p = tfest(z2,2);
```

`sys_np` is a non-parametric identified model. `sys_p` is a parametric identified model.

- 2 Create a Bode plot that includes both systems.

```
bode(sys_np,sys_p,w);
```



### Obtain Magnitude and Phase Standard Deviation Data of Identified Model

Compute the standard deviation of the magnitude and phase of an identified model. Use this data to create a  $3\sigma$  plot of the response uncertainty.

- 1 Identify a transfer function model based on data. Obtain the standard deviation data for the magnitude and phase of the frequency response.

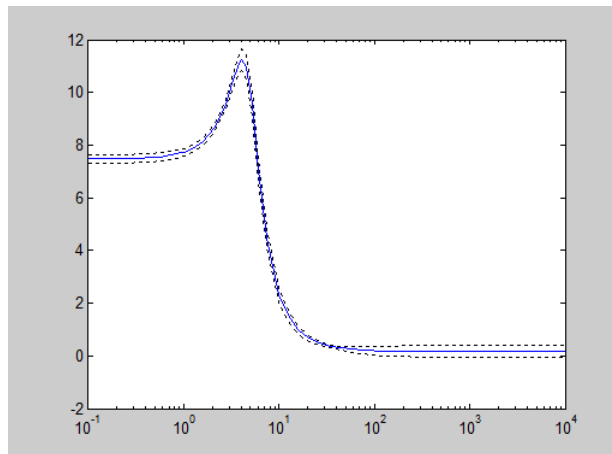
```
load iddata2 z2;  
sys_p = tfest(z2,2);  
w = linspace(0,10*pi,128);  
[mag,ph,w,sdmag,sdphase] = bode(sys_p,w);
```

sys\_p is an identified transfer function model.

sdmag and sdphase contain the standard deviation data for the magnitude and phase of the frequency response, respectively.

- 2 Create a 3 $\sigma$  plot corresponding to the confidence region.

```
mag = squeeze(mag);  
sdmag = squeeze(sdmag);  
semilogx(w,mag,'b',w,mag+3*sdmag,'k:',w,mag-3*sdmag,'k:');
```



## Algorithms

bode computes the frequency response using these steps:

- 1 Computes the zero-pole-gain (zpk) representation of the dynamic system.
- 2 Evaluates the gain and phase of the frequency response based on the zero, pole, and gain data for each input/output channel of the system.

- a For continuous-time systems, `bode` evaluates the frequency response on the imaginary axis  $s = j\omega$  and considers only positive frequencies.
- b For discrete-time systems, `bode` evaluates the frequency response on the unit circle. To facilitate interpretation, the command parameterizes the upper half of the unit circle as

$$z = e^{j\omega T_s}, \quad 0 \leq \omega \leq \omega_N = \frac{\pi}{T_s},$$

where  $T_s$  is the sampling time.  $\omega_N$  is the *Nyquist frequency*. The equivalent continuous-time frequency  $\omega$  is then used as the  $x$ -axis variable. Because  $H(e^{j\omega T_s})$  is periodic and has a period  $2\omega_N$ , `bode` plots the response only up to the Nyquist frequency  $\omega_N$ . If you do not specify a sampling time, `bode` uses  $T_s = 1$ .

**Alternatives**

Use `bodeplot` when you need additional plot customization options.

**See Also**

`bodeplot` | `freqresp` | `nichols` | `nyquist` | `spectrum`

**How To**

- “Dynamic System Models”

# bodemag

---

**Purpose** Bode magnitude response of LTI models

**Syntax**

```
bodemag(sys)
bodemag(sys, {wmin, wmax})
bodemag(sys, w)
bodemag(sys1, sys2, ..., sysN, w)
bodemag(sys1, 'r', sys2, 'y--', sys3, 'gx')
```

**Description** `bodemag(sys)` plots the magnitude of the frequency response of the dynamic system model `sys` (Bode plot without the phase diagram). The frequency range and number of points are chosen automatically.

`bodemag(sys, {wmin, wmax})` draws the magnitude plot for frequencies between `wmin` and `wmax` (in `rad/TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`).

`bodemag(sys, w)` uses the user-supplied vector `W` of frequencies, in `rad/TimeUnit`, at which the frequency response is to be evaluated.

`bodemag(sys1, sys2, ..., sysN, w)` shows the frequency response magnitude of several models `sys1, sys2, ..., sysN` on a single plot. The frequency vector `w` is optional. You can also specify a color, line style, and marker for each model, as in

```
bodemag(sys1, 'r', sys2, 'y--', sys3, 'gx')
```

**See Also** `bode` | `ltiview`

**Purpose** Create list of Bode plot options

**Syntax**  
 P = bodeoptions  
 P = bodeoptions('cstprefs')

**Description** P = bodeoptions returns a list of available options for Bode plots with default values set. You can use these options to customize the Bode plot appearance using the command line.

P = bodeoptions('cstprefs') initializes the plot options with the options you selected in the Control System Toolbox Preferences Editor. For more information about the editor, see “Toolbox Preferences Editor” in the User’s Guide documentation.

The following table summarizes the Bode plot options.

Option	Description
Title, XLabel, YLabel	Label text and style
TickLabel	Tick label style
Grid	Show or hide the grid Specified as one of the following strings: 'off'   'on' <b>Default:</b> 'off'
XlimMode, YlimMode	Limit modes
Xlim, Ylim	Axes limits
IOGrouping	Grouping of input-output pairs Specified as one of the following strings: 'none'   'inputs'   'output'   'all' <b>Default:</b> 'none'
InputLabel, OutputLabel	Input and output label styles
InputVisible, OutputVisible	Visibility of input and output channels

# bodeoptions

---

Option	Description
ConfidenceRegionNumber	Number of standard deviations to use to plotting the response confidence region (identified models only). <b>Default:</b> 1.
FreqUnits	Frequency units, specified as one of the following strings: <ul style="list-style-type: none"><li>• 'Hz'</li><li>• 'rad/second'</li><li>• 'rpm'</li><li>• 'kHz'</li><li>• 'MHz'</li><li>• 'GHz'</li><li>• 'rad/nanosecond'</li><li>• 'rad/microsecond'</li><li>• 'rad/millisecond'</li><li>• 'rad/minute'</li><li>• 'rad/hour'</li><li>• 'rad/day'</li><li>• 'rad/week'</li><li>• 'rad/month'</li><li>• 'rad/year'</li><li>• 'cycles/nanosecond'</li><li>• 'cycles/microsecond'</li><li>• 'cycles/millisecond'</li><li>• 'cycles/hour'</li><li>• 'cycles/day'</li></ul>



Option	Description
	<ul style="list-style-type: none"> <li>• 'cycles/week'</li> <li>• 'cycles/month'</li> <li>• 'cycles/year'</li> </ul> <p><b>Default:</b> 'rad/s'</p> <p>You can also specify 'auto' which uses frequency units rad/TimeUnit relative to system time units specified in the TimeUnit property. For multiple systems with different time units, the units of the first system are used.</p>
FreqScale	<p>Frequency scale</p> <p>Specified as one of the following strings: 'linear'   'log'</p> <p><b>Default:</b> 'log'</p>
MagUnits	<p>Magnitude units</p> <p>Specified as one of the following strings: 'dB'   'abs'</p> <p><b>Default:</b> 'dB'</p>
MagScale	<p>Magnitude scale</p> <p>Specified as one of the following strings: 'linear'   'log'</p> <p><b>Default:</b> 'linear'</p>
MagVisible	<p>Magnitude plot visibility</p> <p>Specified as one of the following strings: 'on'   'off'</p> <p><b>Default:</b> 'on'</p>
MagLowerLimMode	<p>Enables a lower magnitude limit</p> <p>Specified as one of the following strings: 'auto'   'manual'</p> <p><b>Default:</b> 'auto'</p>
MagLowerLim	<p>Specifies the lower magnitude limit</p>
PhaseUnits	<p>Phase units</p> <p>Specified as one of the following strings: 'deg'   'rad'</p> <p><b>Default:</b> 'deg'</p>

# bodeoptions

---

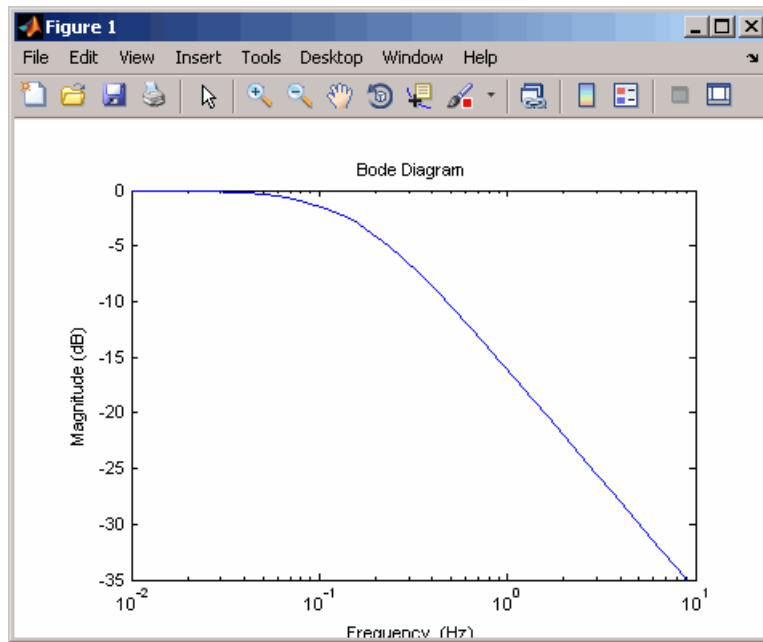
Option	Description
PhaseVisible	Phase plot visibility Specified as one of the following strings: 'on'   'off' <b>Default:</b> 'on'
PhaseWrapping	Enables phase wrapping Specified as one of the following strings: 'on'   'off' <b>Default:</b> 'off'
PhaseMatching	Enables phase matching Specified as one of the following strings: 'on'   'off' <b>Default:</b> 'off'
PhaseMatchingFreq	Frequency for matching phase
PhaseMatchingValue	The value to which phase responses are matched closely

## Examples

In this example, set phase visibility and frequency units in the Bode plot options.

```
P = bodeoptions; % Set phase visibility to off and frequency units to Hz in options
P.PhaseVisible = 'off';
P.FreqUnits = 'Hz'; % Create plot with the options specified by P
h = bodeplot(tf(1,[1,1]),P);
```

The following plot is created, with the phase plot visibility turned off and the frequency units in Hz.



## See Also

[bode](#) | [bodeplot](#) | [getoptions](#) | [setoptions](#) | [showConfidence](#)

# bodeplot

---

**Purpose** Plot Bode frequency response with additional plot customization options

**Syntax**

```
h = bodeplot(sys)
bodeplot(sys)
bodeplot(sys1,sys2,...)
bodeplot(AX,...)
bodeplot(..., plotoptions)
bodeplot(sys,w)
```

**Description** `h = bodeplot(sys)` plot the Bode magnitude and phase of the dynamic system model `sys` and returns the plot handle `h` to the plot. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands.

`bodeplot(sys)` draws the Bode plot of the model `sys`. The frequency range and number of points are chosen automatically.

`bodeplot(sys1,sys2,...)` graphs the Bode response of multiple models `sys1,sys2,...` on a single plot. You can specify a color, line style, and marker for each model, as in

```
bodeplot(sys1, 'r', sys2, 'y--', sys3, 'gx')
```

`bodeplot(AX,...)` plots into the axes with handle `AX`.

`bodeplot(..., plotoptions)` plots the Bode response with the options specified in `plotoptions`. Type

```
help bodeoptions
```

for a list of available plot options. See “Example 2” on page 2-99 for an example of phase matching using the `PhaseMatchingFreq` and `PhaseMatchingValue` options.

`bodeplot(sys,w)` draws the Bode plot for frequencies specified by `w`. When `w = {wmin,wmax}`, the Bode plot is drawn for frequencies between `wmin` and `wmax` (in `rad/TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`).

When `w` is a user-supplied vector `w` of frequencies, in rad/TimeUnit, the Bode response is drawn for the specified frequencies.

See `logspace` to generate logarithmically spaced frequency vectors.

## **Tips**

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

## **Examples**

### **Example 1**

Use the plot handle to change options in a Bode plot.

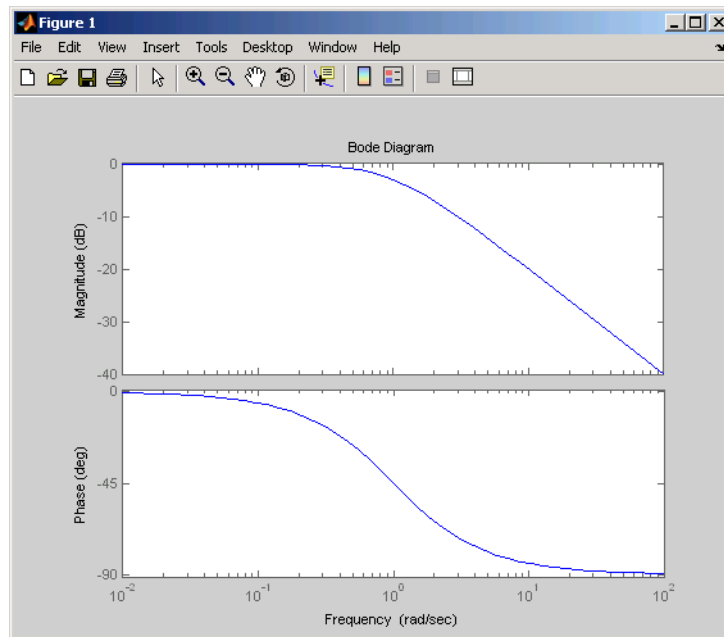
```
sys = rss(5);  
h = bodeplot(sys);  
% Change units to Hz and make phase plot invisible  
setoptions(h, 'FreqUnits', 'Hz', 'PhaseVisible', 'off');
```

### **Example 2**

The properties `PhaseMatchingFreq` and `PhaseMatchingValue` are parameters you can use to specify the phase at a specified frequency. For example, enter the following commands.

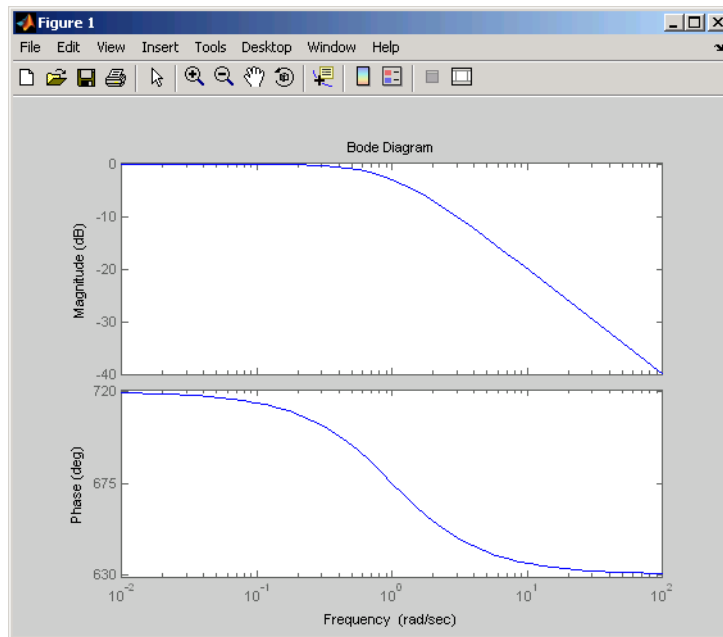
```
sys = tf(1,[1 1]);  
h = bodeplot(sys) % This displays a Bode plot.
```

# bodeplot



Use this code to match a phase of 750 degrees to 1 rad/s.

```
p = getoptions(h);  
p.PhaseMatching = 'on';  
p.PhaseMatchingFreq = 1;  
p.PhaseMatchingValue = 750; % Set the phase to 750 degrees at 1  
% rad/s.  
setoptions(h,p); % Update the Bode plot.
```



The first bode plot has a phase of -45 degrees at a frequency of 1 rad/s. Setting the phase matching options so that at 1 rad/s the phase is near 750 degrees yields the second Bode plot. Note that, however, the phase can only be  $-45 + N \cdot 360$ , where  $N$  is an integer, and so the plot is set to the nearest allowable phase, namely 675 degrees (or  $2 \cdot 360 - 45 = 675$ ).

### Example 3

Compare the frequency responses of identified state-space models of order 2 and 6 along with their 2 std confidence regions.

```
load iddata1
sys1 = n4sid(z1, 2) % discrete-time IDSS model of order 2
sys2 = n4sid(z1, 6) % discrete-time IDSS model of order 6
```

Both models produce about 76% fit to data. However, sys2 shows higher uncertainty in its frequency response, especially close to Nyquist frequency as shown by the plot:

# bodeplot

---

```
w = linspace(8,10*pi,256);  
h = bodeplot(sys1,sys2,w);  
setoptions(h, 'PhaseMatching', 'on', 'ConfidenceRegionNumberSD', 2);
```

Use the context menu by right-clicking **Characteristics > Confidence Region** to turn on the confidence region characteristic.

## Example 4

Compare the frequency response of a parametric model, identified from input/output data, to a nonparametric model identified using the same data.

- 1 Identify parametric and non-parametric models based on data.

```
load iddata2 z2;  
w = linspace(0,10*pi,128);  
sys_np = spa(z2,[],w);  
sys_p = tfest(z2,2);
```

`spa` and `tfest` require System Identification Toolbox software. `sys_np` is a non-parametric identified model. `sys_p` is a parametric identified model.

- 2 Create a Bode plot that includes both systems.

```
opt = bodeoptions; opt.PhaseMatching = 'on';  
bodeplot(sys_np,sys_p,w, opt);
```

## See Also

[bode](#) | [bodeoptions](#) | [getoptions](#) | [setoptions](#) | [showConfidence](#)



**Purpose**

Convert model from continuous to discrete time

**Syntax**

```
sysd = c2d(sys,Ts)
sysd = c2d(sys,Ts,method)
sysd = c2d(sys,Ts,opts)
[sysd,G] = c2d(sys,Ts,method)
[sysd,G] = c2d(sys,Ts,opts)
```

**Description**

`sysd = c2d(sys,Ts)` discretizes the continuous-time dynamic system model `sys` using zero-order hold on the inputs and a sample time of `Ts` seconds.

`sysd = c2d(sys,Ts,method)` discretizes `sys` using the specified discretization method `method`.

`sysd = c2d(sys,Ts,opts)` discretizes `sys` using the option set `opts`, specified using the `c2dOptions` command.

`[sysd,G] = c2d(sys,Ts,method)` returns a matrix, `G` that maps the continuous initial conditions  $x_0$  and  $u_0$  of the state-space model `sys` to the discrete-time initial state vector  $x[0]$ . `method` is optional. To specify additional discretization options, use `[sysd,G] = c2d(sys,Ts,opts)`.

**Tips**

- Use the syntax `sysd = c2d(sys,Ts,method)` to discretize `sys` using the default options for `method`. To specify additional discretization options, use the syntax `sysd = c2d(sys,Ts,opts)`.
- To specify the `tustin` method with frequency prewarping (formerly known as the 'prewarp' method), use the `PrewarpFrequency` option of `c2dOptions`.

**Input Arguments****sys**

Continuous-time dynamic system model (except frequency response data models). `sys` can represent a SISO or MIMO system, except that the 'matched' discretization method supports SISO systems only.

`sys` can have input/output or internal time delays; however, the `'matched'` and `'impulse'` methods do not support state-space models with internal time delays.

The following identified linear systems cannot be discretized directly:

- `idgrey` models with `FcnType` is `'c'`. Convert to `idss` model first.
- `idproc` models. Convert to `idtf` or `idpoly` model first.

For the syntax `[sysd,G] = c2d(sys,Ts,opts)`, `sys` must be a state-space model.

## **Ts**

Sample time.

## **method**

String specifying a discretization method:

- `'zoh'` — Zero-order hold (default). Assumes the control inputs are piecewise constant over the sampling period `Ts`.
- `'foh'` — Triangle approximation (modified first-order hold). Assumes the control inputs are piecewise linear over the sampling period `Ts`.
- `'impulse'` — Impulse invariant discretization.
- `'tustin'` — Bilinear (Tustin) method.
- `'matched'` — Zero-pole matching method.

For more information about discretization methods, see “Continuous-Discrete Conversion Methods”.

## **opts**

Discretization options. Create `opts` using `c2dOptions`.

## **Output Arguments**

### **sysd**

Discrete-time model of the same type as the input system `sys`.

When `sys` is an identified (IDLTI) model, `sysd`:

- Includes both measured and noise components of `sys`. The innovations variance  $\lambda$  of the continuous-time identified model `sys`, stored in its `NoiseVariance` property, is interpreted as the intensity of the spectral density of the noise spectrum. The noise variance in `sysd` is thus  $\lambda/T_s$ .
- Does not include the estimated parameter covariance of `sys`. If you want to translate the covariance while discretizing the model, use `translatecov`.

## G

Matrix relating continuous-time initial conditions  $x_0$  and  $u_0$  of the state-space model `sys` to the discrete-time initial state vector  $x[0]$ , as follows:

$$x[0] = G \cdot \begin{bmatrix} x_0 \\ u_0 \end{bmatrix}$$

For state-space models with time delays, `c2d` pads the matrix `G` with zeroes to account for additional states introduced by discretizing those delays. See “Continuous-Discrete Conversion Methods” for a discussion of modeling time delays in discretized systems.

## Examples

Discretize the continuous-time transfer function:

$$H(s) = \frac{s-1}{s^2 + 4s + 5}$$

with input delay  $T_d = 0.35$  second. To discretize this system using the triangle (first-order hold) approximation with sample time  $T_s = 0.1$  second, type

```
H = tf([1 -1], [1 4 5], 'inputdelay', 0.35);
Hd = c2d(H, 0.1, 'foh'); % discretize with FOH method and
                        % 0.1 second sample time
```

Transfer function:

$$0.0115 z^3 + 0.0456 z^2 - 0.0562 z - 0.009104$$

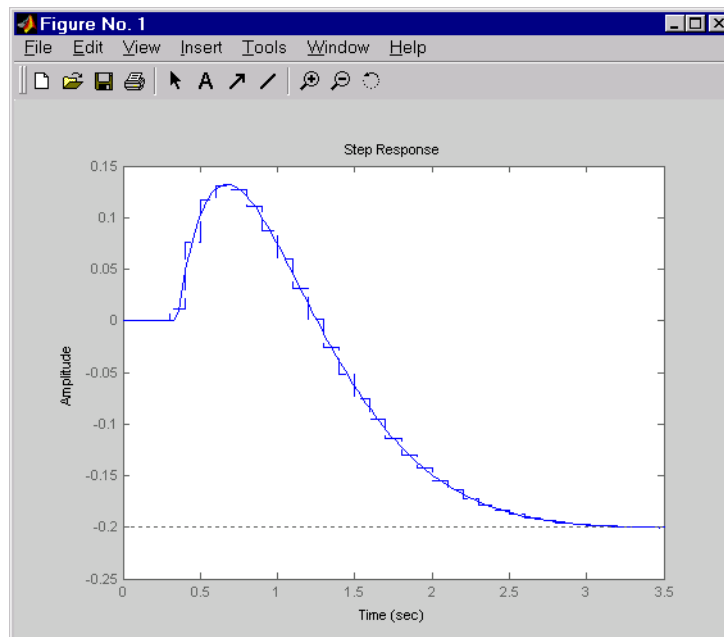
-----

$$z^6 - 1.629 z^5 + 0.6703 z^4$$

Sampling time: 0.1

The next command compares the continuous and discretized step responses.

```
step(H, '-', Hd, '--')
```



Discretize the delayed transfer function

$$H(s) = e^{-0.25s} \frac{10}{s^2 + 3s + 10}$$

using zero-order hold on the input, and a 10-Hz sampling rate.

```
h = tf(10,[1 3 10],'iodelay',0.25); % create transfer function
hd = c2d(h, 0.1) % zoh is the default method
```

These commands produce the discrete-time transfer function

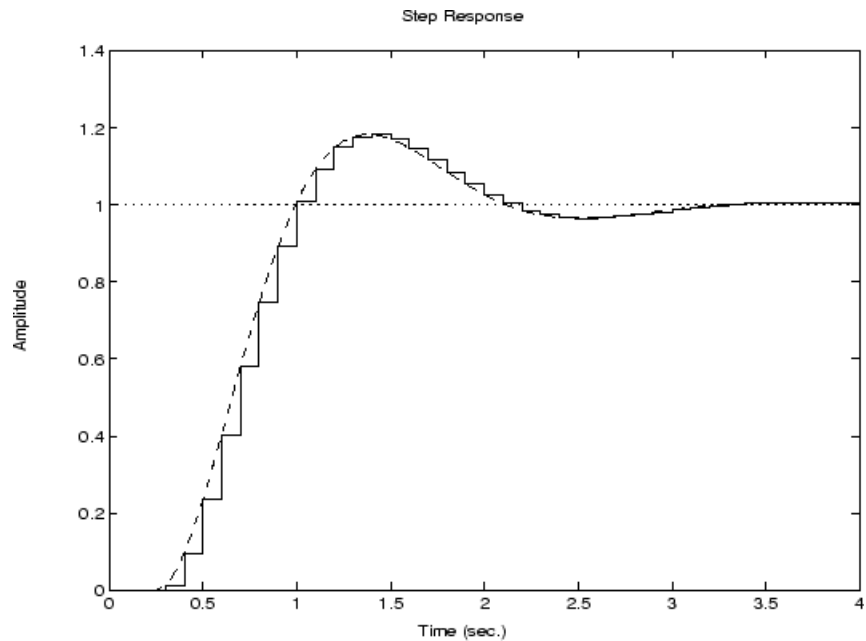
```
Transfer function:
      0.01187 z^2 + 0.06408 z + 0.009721
z^(-3) * -----
      z^2 - 1.655 z + 0.7408
```

```
Sampling time: 0.1
```

In this example, the discretized model `hd` has a delay of three sampling periods. The discretization algorithm absorbs the residual half-period delay into the coefficients of `hd`.

Compare the step responses of the continuous and discretized models using

```
step(h, '-',hd, '-')
```



Discretize a state-space model with time delay, using a Thiran filter to model fractional delays:

```
sys = ss(tf([1, 2], [1, 4, 2])); % create a state-space model
sys.InputDelay = 2.7           % add input delay
```

This command creates a continuous-time state-space model with two states, as the output shows:

```
a =
      x1  x2
x1  -4  -2
x2   1   0
```

```
b =
      u1
```

```

x1  2
x2  0

c =
      x1  x2
y1  0.5  1

d =
      u1
y1  0

```

Input delays (listed by channel): 2.7

Continuous-time model.

Use `c2dOptions` to create a set of discretization options, and discretize the model. This example uses the Tustin discretization method.

```

opt = c2dOptions('Method', 'tustin', 'FractDelayApproxOrder', 3);
sysd1 = c2d(sys, 1, opt)    % 1s sampling time

```

These commands yield the result

```

a =
      x1      x2      x3      x4      x5
x1  -0.4286  -0.5714  -0.00265  0.06954  2.286
x2   0.2857   0.7143  -0.001325  0.03477  1.143
x3    0         0      -0.2432   0.1449  -0.1153
x4    0         0         0.25     0         0
x5    0         0         0         0.125    0

b =
      u1
x1  0.002058
x2  0.001029
x3    8
x4    0
x5    0

```

```
c =
      x1      x2      x3      x4      x5
y1    0.2857  0.7143 -0.001325  0.03477  1.143
```

```
d =
      u1
y1  0.001029
```

```
Sampling time: 1
Discrete-time model.
```

The discretized model now contains three additional states  $x_3$ ,  $x_4$ , and  $x_5$  corresponding to a third-order Thiran filter. Since the time delay divided by the sampling time is 2.7, the third-order Thiran filter (`FractDelayApproxOrder = 3`) can approximate the entire time delay.

---

Discretize an identified, continuous-time transfer function and compare its performance against a directly estimated discrete-time model

Estimate a continuous-time transfer function and discretize it.

```
load iddata1
sys1c = tfest(z1, 2);
sys1d = c2d(sys1c, 0.1, 'zoh');
```

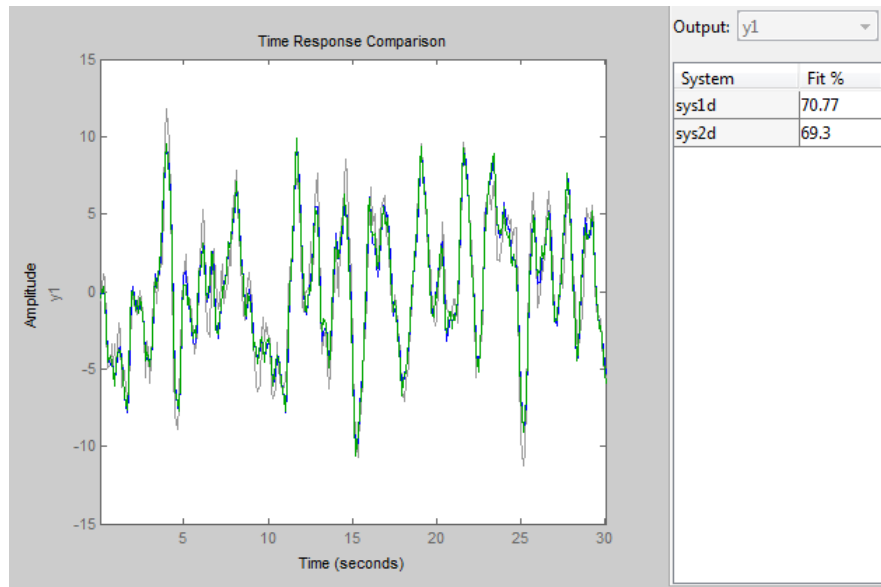
Estimate a second order discrete-time transfer function.

```
sys2d = tfest(z1, 2, 'Ts', 0.1);
```

Compare the two models.

```
compare(z1, sys1d, sys2d)
```





The two systems are virtually identical.

Discretize an identified state-space model to build a one-step ahead predictor of its response.

```
load iddata2
sysc = ssest(z2, 4);
sysd = c2d(sysc, 0.1, 'zoh');
[A,B,C,D,K] = idssdata(sysd);
Predictor = ss(A-K*C, [K B-K*D], C, [0 D], 0.1);
```

The Predictor is a two input model which uses the measured output and input signals (`[z1.y z1.u]`) to compute the 1-step predicted response of `sysc`.

## Algorithms

For information about the algorithms for each c2d conversion method, see “Continuous-Discrete Conversion Methods”.

**See Also**

[c2dOptions](#) | [d2c](#) | [d2d](#) | [thiran](#) | [translatecov](#)

**How To**

- “Dynamic System Models”
- “Discretize a Compensator”
- “Continuous-Discrete Conversion Methods”

**Purpose** Create option set for continuous- to discrete-time conversions

**Syntax**

```
opts = c2dOptions
opts = c2dOptions('OptionName',
    OptionValue)
```

**Description** `opts = c2dOptions` returns the default options for `c2d`.  
`opts = c2dOptions('OptionName', OptionValue)` accepts one or more comma-separated name/value pairs that specify options for the `c2d` command. Specify *OptionName* inside single quotes.

## Input Arguments

### Name-Value Pair Arguments

#### Method

Discretization method, specified as one of the following values:

'zoh'	Zero-order hold, where <code>c2d</code> assumes the control inputs are piecewise constant over the sampling period $T_s$ .
'foh'	Triangle approximation (modified first-order hold), where <code>c2d</code> assumes the control inputs are piecewise linear over the sampling period $T_s$ . (See [1], p. 228.)
'impulse'	Impulse-invariant discretization.
'tustin'	Bilinear (Tustin) approximation. By default, <code>c2d</code> discretizes with no prewarp and rounds any fractional time delays to the nearest multiple of the sample time. To include prewarp, use the <code>PrewarpFrequency</code> option. To approximate fractional time delays, use the <code>FractDelayApproxOrder</code> option.
'matched'	Zero-pole matching method. (See [1], p. 224.) By default, <code>c2d</code> rounds any fractional time delays to the nearest multiple of the sample time. To approximate fractional time delays, use the <code>FractDelayApproxOrder</code> option.

**Default:** 'zoh'

## **PrewarpFrequency**

Prewarp frequency for 'tustin' method, specified in rad/TimeUnit, where TimeUnit is the time units, specified in the TimeUnit property, of the discretized system. Takes positive scalar values. A value of 0 corresponds to the standard 'tustin' method without prewarp.

**Default:** 0

## **FractDelayApproxOrder**

Maximum order of the Thiran filter used to approximate fractional delays in the 'tustin' and 'matched' methods. Takes integer values. A value of 0 means that c2d rounds fractional delays to the nearest integer multiple of the sample time.

**Default:** 0

## **Examples**

Discretize two models using identical discretization options.

```
% generate two arbitrary continuous-time state-space models
sys1 = rss(3, 2, 2);
sys2 = rss(4, 4, 1);
```

Use c2dOptions to create a set of discretization options.

```
opt = c2dOptions('Method', 'tustin', 'PrewarpFrequency', 3.4);
```

Then, discretize both models using the option set.

```
dsys1 = c2d(sys1, 0.1, opt); % 0.1s sampling time
dsys2 = c2d(sys2, 0.2, opt); % 0.2s sampling time
```

The c2dOptions option set does not include the sampling time Ts. You can use the same discretization options to discretize systems using a different sampling time.

## References

[1] Franklin, G.F., Powell, D.J., and Workman, M.L., *Digital Control of Dynamic Systems* (3rd Edition), Prentice Hall, 1997.

## See Also

c2d

**Purpose** State-space canonical realization

**Syntax**  
`csys = canon(sys,type)`  
`[csys,T]= canon(sys,type)`  
`csys = canon(sys,'modal',condt)`

**Description** `csys = canon(sys,type)` transforms the linear model `sys` into a canonical state-space model `csys`. The argument `type` specifies whether `csys` is in modal or companion form.

`[csys,T]= canon(sys,type)` also returns the state-coordinate transformation `T` that relates the states of the state-space model `sys` to the states of `csys`.

`csys = canon(sys,'modal',condt)` specifies an upper bound `condt` on the condition number of the block-diagonalizing transformation.

## Input Arguments

**sys**  
Any linear dynamic system model, except for `frd` models.

**type**  
String specifying the type of canonical form of `csys`. `type` can take one of the two following values:

- 'modal' — convert `sys` to modal form.
- 'companion' — convert `sys` to companion form.

**condt**  
Positive scalar value specifying an upper bound on the condition number of the block-diagonalizing transformation that converts `sys` to `csys`. This argument is available only when `type` is 'modal'.

Increase `condt` to reduce the size of the eigenvalue clusters in the  $A$  matrix of `csys`. Setting `condt = Inf` diagonalizes  $A$ .

**Default:** 1e8

## Output Arguments

### **csys**

State-space (ss) model. **csys** is a state-space realization of **sys** in the canonical form specified by **type**.

### **T**

Matrix specifying the transformation between the state vector  $x$  of the state-space model **sys** and the state vector  $x_c$  of **csys**:

$$x_c = Tx$$

This argument is available only when **sys** is state-space model.

## Definitions

### **Modal Form**

In modal form,  $A$  is a block-diagonal matrix. The block size is typically 1-by-1 for real eigenvalues and 2-by-2 for complex eigenvalues. However, if there are repeated eigenvalues or clusters of nearby eigenvalues, the block size can be larger.

For example, for a system with eigenvalues  $(\lambda_1, \sigma \pm j\omega, \lambda_2)$ , the modal  $A$  matrix is of the form

$$\begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \sigma & \omega & 0 \\ 0 & -\omega & \sigma & 0 \\ 0 & 0 & 0 & \lambda_2 \end{bmatrix}$$

### **Companion Form**

In the companion realization, the characteristic polynomial of the system appears explicitly in the rightmost column of the  $A$  matrix. For a system with characteristic polynomial

$$p(s) = s^n + \alpha_1 s^{n-1} + \dots + \alpha_{n-1} s + \alpha_n$$

the corresponding companion  $A$  matrix is

$$A = \begin{bmatrix} 0 & 0 & \dots & \dots & 0 & -\alpha_n \\ 1 & 0 & 0 & \dots & 0 & -\alpha_n - 1 \\ 0 & 1 & 0 & \dots & \vdots & \vdots \\ \vdots & 0 & \dots & \dots & \vdots & \vdots \\ 0 & \dots & \dots & 1 & 0 & -\alpha_2 \\ 0 & \dots & \dots & 0 & 1 & -\alpha_1 \end{bmatrix}$$

The companion transformation requires that the system be controllable from the first input. The companion form is poorly conditioned for most state-space computations; avoid using it when possible.

## Examples

This example uses `canon` to convert a system having doubled poles and clusters of close poles to modal canonical form.

Consider the system  $G$  having the following transfer function:

$$G(s) = 100 \frac{(s-1)(s+1)}{s(s+10)(s+10.0001)(s-(1+i))^2(s-(1-i))^2}$$

To create a linear model of this system and convert it to modal canonical form, enter:

```
G = zpke([1 -1],[0 -10 -10.0001 1+1i 1-1i 1+1i 1-1i],100);
Gc = canon(G, 'modal');
```

The system  $G$  has a pair of nearby poles at  $s = -10$  and  $s = -10.0001$ .  $G$  also has two complex poles of multiplicity 2 at  $s = 1 + i$  and  $s = 1 - i$ . As a result, the modal form, has a block of size 2 for the two poles near  $s = -10$ , and a block of size 4 for the complex eigenvalues. To see this, enter the following command:

```
Gc.A
```

This command returns the result:



ans =

```

0      0      0      0      0      0      0
0      1.0000  1.0000  0      0      0      0
0      -1.0000  1.0000  2.0548  0      0      0
0      0      0      1.0000  1.0000  0      0
0      0      0      -1.0000  1.0000  0      0
0      0      0      0      0      -10.0000  8.0573
0      0      0      0      0      0      -10.0000

```

To separate the two poles near  $s = -10$ , you can increase the value of `condt`. For example, entering the commands:

```
Gc2 = canon(G, 'modal', 1e10);
Gc2.A
```

returns the result:

ans =

```

0      0      0      0      0      0      0
0      1.0000  1.0000  0      0      0      0
0      -1.0000  1.0000  2.0548  0      0      0
0      0      0      1.0000  1.0000  0      0
0      0      0      -1.0000  1.0000  0      0
0      0      0      0      0      -10.0000  8.0573
0      0      0      0      0      0      -10.0000

```

The  $A$  matrix of `Gc2` includes separate diagonal elements for the poles near  $s = -10$ . The cost of increasing the maximum condition number of  $A$  is that the  $B$  matrix includes some large values.

```
format shortE
Gc2.B
```

ans =

```
3.2000e-001
```

```
-6.5691e-003  
5.4046e-002  
-1.9502e-001  
1.0637e+000  
3.2533e+005  
3.2533e+005
```

---

This example estimates a state-space model that is freely parameterized and convert to companion form after estimation.

```
load icEngine.mat  
z = iddata(y,u,0.04);  
FreeModel = n4sid(z,4,'InputDelay',2);  
CanonicalModel = canon(FreeModel, 'companion')
```

Obtain the covariance of the resulting form by running a zero-iteration update to model parameters.

```
opt = ssestOptions; opt.SearchOption.MaxIter = 0;  
CanonicalModel = ssest(z, CanonicalModel, opt)
```

Compare frequency response confidence bounds of FreeModel to CanonicalModel.

```
h = bodeplot(FreeModel, CanonicalModel)
```

the bounds are identical.

## Algorithms

The `canon` command uses the `bdschur` command to convert `sys` into modal form and to compute the transformation `T`. If `sys` is not a state-space model, the algorithm first converts it to state space using `ss`.

The reduction to companion form uses a state similarity transformation based on the controllability matrix [1].

## References

[1] Kailath, T. *Linear Systems*, Prentice-Hall, 1980.

**See Also**

ctrb | ctrbf | ss2ss

# chgFreqUnit

---

**Purpose** Change frequency units of frequency-response data model

**Syntax** `sys_new = chgFreqUnit(sys,newfrequnits)`

**Description** `sys_new = chgFreqUnit(sys,newfrequnits)` changes units of the frequency points in `sys` to `newfrequnits`. Both `Frequency` and `FrequencyUnit` properties of `sys` adjust so that the frequency responses of `sys` and `sys_new` match.

**Tips**

- Use `chgFreqUnit` to change the units of frequency points without modifying system behavior.

**Input Arguments**

**sys**  
Frequency-response data (`frd`, `idfrd`, or `genfrd`) model

**newfrequnits**

New units of frequency points, specified as one of the following strings:

- 'rad/TimeUnit'
- 'cycles/TimeUnit'
- 'rad/s'
- 'Hz'
- 'kHz'
- 'MHz'
- 'GHz'
- 'rpm'

`rad/TimeUnit` and `cycles/TimeUnit` express frequency units relative to the system time units specified in the `TimeUnit` property.

**Default:** 'rad/TimeUnit'

## Output Arguments

### **sys\_new**

Frequency-response data model of the same type as `sys` with new units of frequency points. The frequency response of `sys_new` is same as `sys`.

## Examples

This example shows how to change units of the frequency points in a frequency-response data model.

- 1 Create a frequency-response data model.

```
load AnalyzerData;
sys = frd(resp,freq);
```

The data file `AnalyzerData` has column vectors `freq` and `resp`. These vectors contain 256 test frequencies and corresponding complex-valued frequency response points, respectively. The default frequency units of `sys` is `rad/TimeUnit`, where `TimeUnit` is the system time units.

- 2 Change the frequency units.

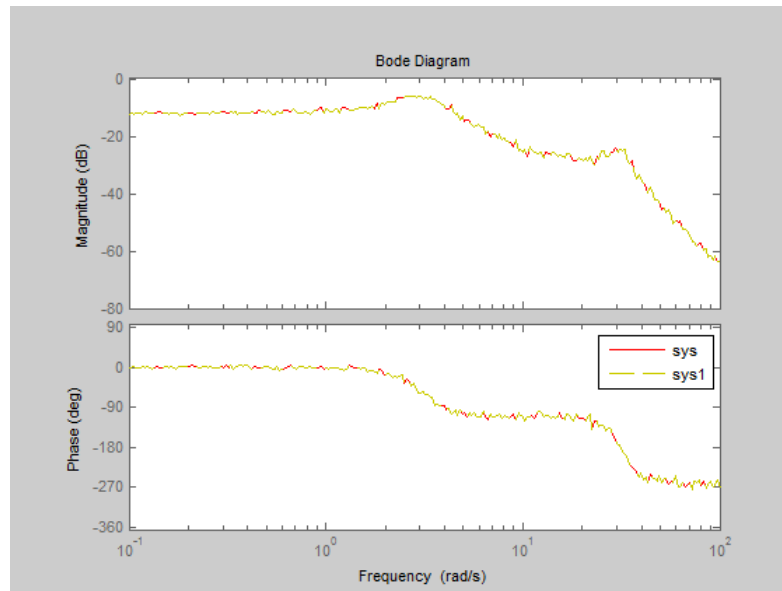
```
sys1 = chgFreqUnit(sys,'rpm');
```

The `FrequencyUnit` property of `sys1` is `rpm`.

- 3 Compare the Bode responses of `sys` and `sys1`.

```
bode(sys,'r',sys1,'y--');
legend('sys','sys1')
```

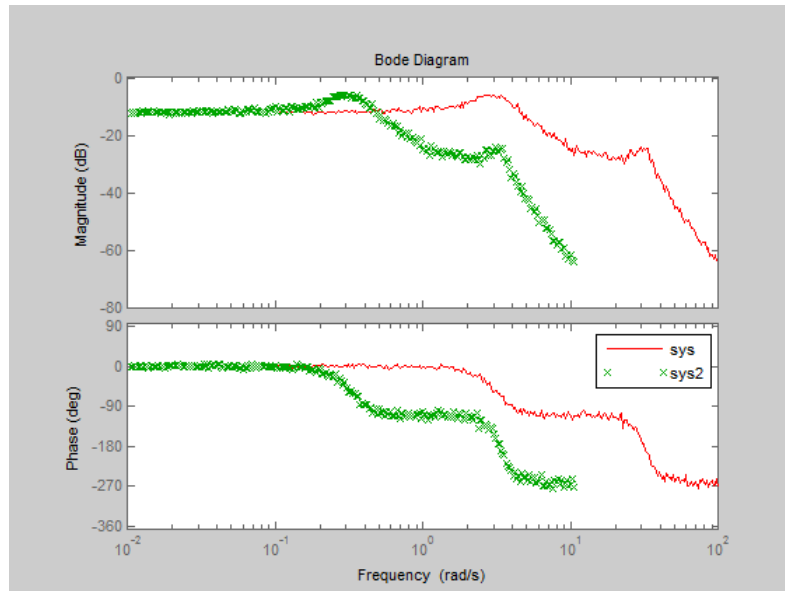
The magnitude and phase of `sys` and `sys1` match.



- 4** (Optional) Change the `FrequencyUnit` property of `sys` to compare the Bode response with the original system.

```
sys2=sys;  
sys2.FrequencyUnit = 'rpm';  
bode(sys, 'r', sys2, 'gx');  
legend('sys', 'sys2');
```

Changing the `FrequencyUnit` property changes the original system. Therefore, the Bode responses of `sys` and `sys2` do not match. For example, the original corner frequency at 2 rad/s changes to 2 rpm (or 0.2 rad/s).



**See Also** `chgTimeUnit` | `frd` | `idfrd`

## Tutorials

- “Specify Frequency Units of Frequency-Response Data Model”<sup>1</sup>

- 1.



<b>Purpose</b>	Change time units of dynamic system
<b>Syntax</b>	<code>sys_new = chgTimeUnit(sys,newtimeunits)</code>
<b>Description</b>	<code>sys_new = chgTimeUnit(sys,newtimeunits)</code> changes the time units of <code>sys</code> to <code>newtimeunits</code> . The time- and frequency-domain characteristics of <code>sys</code> and <code>sys_new</code> match.
<b>Tips</b>	<ul style="list-style-type: none"><li>• Use <code>chgTimeUnit</code> to change the time units without modifying system behavior.</li></ul>
<b>Input Arguments</b>	<p><b>sys</b> Dynamic system model</p> <p><b>newtimeunits</b> New time units, specified as one of the following strings:</p> <ul style="list-style-type: none"><li>• 'nanoseconds'</li><li>• 'microseconds'</li><li>• 'milliseconds'</li><li>• 'seconds'</li><li>• 'minutes'</li><li>• 'hours'</li><li>• 'days'</li><li>• 'weeks'</li><li>• 'months'</li><li>• 'years'</li></ul> <p><b>Default:</b> 'seconds'</p>

# chgTimeUnit

---

## Output Arguments

### **sys\_new**

Dynamic system model of the same type as **sys** with new time units. The time response of **sys\_new** is same as **sys**.

If **sys** is an identified linear model, both the model parameters as and their minimum and maximum bounds are scaled to the new time units.

## Examples

This example shows how to change the time units of a transfer function model.

- 1 Create a transfer function model.

```
num = [4 2];  
den = [1 3 10];  
sys = tf(num,den);
```

The default time units of **sys** is seconds.

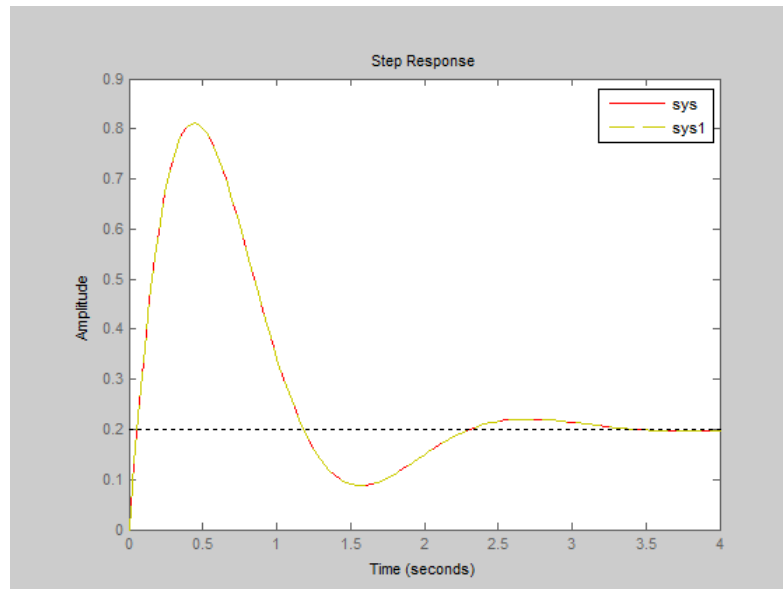
- 2 Change the time units.

```
sys1 = chgTimeUnit(sys,'minutes');
```

The `TimeUnit` property of **sys1** is milliseconds.

- 3 Compare the step responses of **sys** and **sys1**.

```
step(sys,'r',sys1,'y--');  
legend('sys','sys1');
```

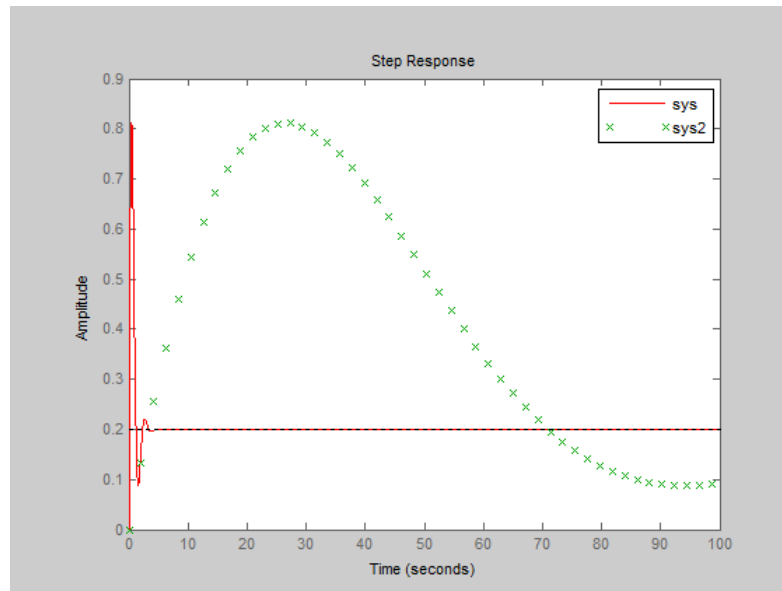


The step responses of `sys` and `sys1` match.

- 4 (Optional) Change the `TimeUnit` property of `sys`, and compare the step response with the original system.

```
sys2=sys;
sys2.TimeUnit = 'minutes';
step(sys, 'r', sys2, 'gx');
legend('sys', 'sys2');
```

Changing the `TimeUnit` property changes the original system. Therefore, the step responses of `sys` and `sys2` do not match. For example, the original rise time of 0.04 seconds changes to 0.04 minutes.



## See Also

`chgFreqUnit` | `tf` | `zpk` | `ss` | `frd` | `pid` | `idss` | `idpoly` | `idtf` | `idproc`

## Tutorials

- “Specify Model Time Units”

**Purpose**

Compare model output and measured output

**Syntax**

```
compare(data,sys)
compare(data,sys,prediction_horizon)
compare(data,sys,style,prediction_horizon)
compare(data,sys1,...,sysN,style,prediction_horizon)
compare(data,sys1,style1,...,sysN,styleN,prediction_horizon)
compare(data,sys1,...,___,opt)
[y,fit,x0] = compare(data,___)
```

**Description**

`compare(data,sys)` plots the simulated response of a dynamic system model, `sys`, superimposed over validation data, `data`, for comparison. The plot also displays the normalized root mean square (NRMSE) measure of the goodness of the fit.

`compare(data,sys,prediction_horizon)` compares the predicted response of `sys` to the measured response in `data`. Measured output values in `data` up to time `t-prediction_horizon` are used to predict the output of `sys` at time `t`.

The matching of the input/output channels in `data` and `sys` is based on the channel names. So, it is possible to evaluate models that do not use all the input channels that are available in `data`.

`compare(data,sys,style,prediction_horizon)` allows specification of the line style, color, or marker for the plot.

`compare(data,sys1,...,sysN,style,prediction_horizon)` compares multiple dynamic systems responses in a single figure.

`compare(data,sys1,style1,...,sysN,styleN,prediction_horizon)` compares systems responses using the color, linestyle, and markers specified by the `style` strings.

`compare(data,sys1,...,___,opt)` configures the comparison using an option set, `opt`.

`[y,fit,x0] = compare(data,___)` returns the model response, `y`, goodness of fit value, `fit`, and the initial states, `x0`. No plot is generated.

## Input Arguments

### **data**

Validation data.

Specify data as either an `iddata` or `idfrd` object.

If `sys` is an `iddata` object, then `data` must be an `iddata` object with matching domain, number of experiments and time or frequency vectors.

If `sys` is a frequency response model (`idfrd` or `frd`), then `data` must be a frequency response model too.

`data` can represent either time- or frequency domain data when comparing with linear models. `data` must be time-domain data when comparing with a nonlinear model.

For frequency domain data, the real and imaginary parts of the corresponding frequency functions are shown in separate axes.

When `data` is an FRD model, the frequency responses of `data` and `sys` are plotted.

### **sys**

Dynamic system model or data object.

Specify `sys` as either a dynamic system model or an `iddata` object.

When the time or frequency units of `data` do not match those of `sys`, `sys` is rescaled to match the units of `data`.

### **prediction\_horizon**

Prediction horizon.

Specify `prediction_horizon` as `Inf` to obtain a pure simulation of the system.

`prediction_horizon` is ignored when `sys` is an `iddata` object, an FRD model or a dynamic system with no noise component.

`prediction_horizon` is also ignored when using frequency response validation data.

For time-series models, use a finite value for `prediction_horizon`.

**Default:** Inf

**style**

Line style, marker, and color of both the linear and marker, specified as a one-, two-, or three-part string enclosed in single quotes (' '). The elements of the string can appear in any order. The string can specify only the line style, the marker, or the color.

For more information about configuring the **style** string, see “Colors, Line Styles, and Markers” in the MATLAB documentation.

**opt**

Comparison option set.

opt is an option set that specifies, among other options, the following:

- handling of initial conditions
- sample range for computing fit numbers
- data offsets
- output weighting

Use `compareOptions` to create the option set.

**Output Arguments**

**y**

Model response.

Measured output values in **data** up to time  $t = t\text{-prediction\_horizon}$  are used to predict the output of **sys** at time  $t$ .

For multi-model comparisons, **y** is a cell array, with one entry for each input model.

For multi-experiment data, **y** is a cell array, with one entry for each experiment.

For multi-model comparisons using multi-experiment data, **y** is an  $N_{sys}$ -by- $N_{exp}$  cell array. Here,  $N_{sys}$  is the number of models, and  $N_{exp}$  is the number of experiments.

If `sys` is a model array, then, `y` is an array, with an entry corresponding to each model in `sys` and experiment in `data`.

### **fit**

NRMSE fitness value.

The fit is calculated (in percentage) using :

$$\text{fit} = 100 \left( 1 - \frac{\|y - \hat{y}\|}{\|y - \text{mean}(y)\|} \right)$$

where  $y$  is the validation data output and  $\hat{y}$  is the output of `sys`.

For FRD models, `fit` is calculated by comparing the complex frequency response; the magnitude and phase curves shown in the plot are not compared separately.

If `data` is an `iddata` object, `fit` is an  $N_y$ -by-1 vector, where  $N_y$  is the number of outputs.

If `data` is an FRD model with  $N_y$  outputs and  $N_u$  inputs, `fit` is an  $N_y$ -by- $N_u$  matrix. Each entry of `fit` corresponds to an input/output pair in `sys`.

For multi-model comparisons, `fit` is a cell array, with one entry for each input model.

For multi-experiment data, `fit` is a cell array, with one entry for each experiment.

For multi-model comparisons using multi-experiment data, `fit` is an  $N_{sys}$ -by- $N_{exp}$  cell array. Here,  $N_{sys}$  is the number of models, and  $N_{exp}$  is the number of experiments.

### **x0**

Initial conditions used to compute system response.

When `sys` is an `frd` or `iddata` object, `x0` is `[]`.



For multi-model comparisons, `x0` is a cell array, with one entry for each input model.

For multi-experiment data, `x0` is a cell array, with one entry for each experiment.

For multi-model comparisons using multi-experiment data, `x0` is an  $N_{sys}$ -by- $N_{exp}$  cell array. Here,  $N_{sys}$  is the number of models, and  $N_{exp}$  is the number of experiments.

## Examples

### Compare Estimated Model to Data

Compare the output of an estimated state-space model to measured data.

Estimate a state-space model for measured data.

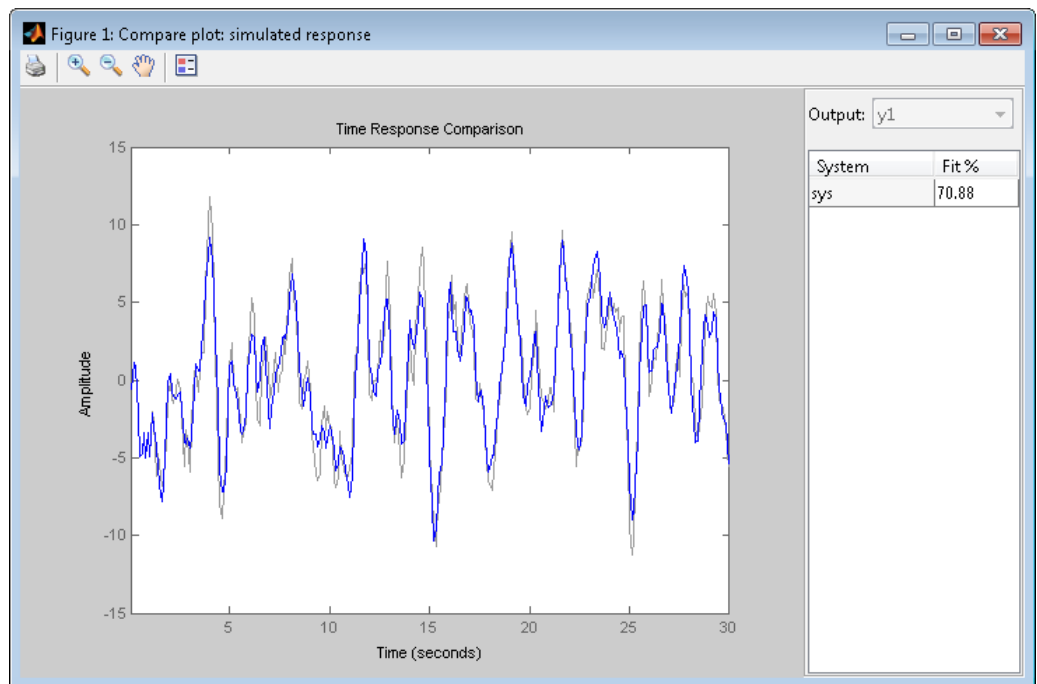
```
load iddata1 z1;  
sys = ssest(z1,3)
```

`sys`, an `idss` model, is a continuous-time state-space model.

Compare the 10 step ahead predicted output to the measured output.

```
prediction_horizon = 10;  
compare(z1,sys,prediction_horizon);
```

# compare



## Compare Multiple Estimated Models

Compare the outputs of an estimated process model, and an estimated Output-Error polynomial model to measured data.

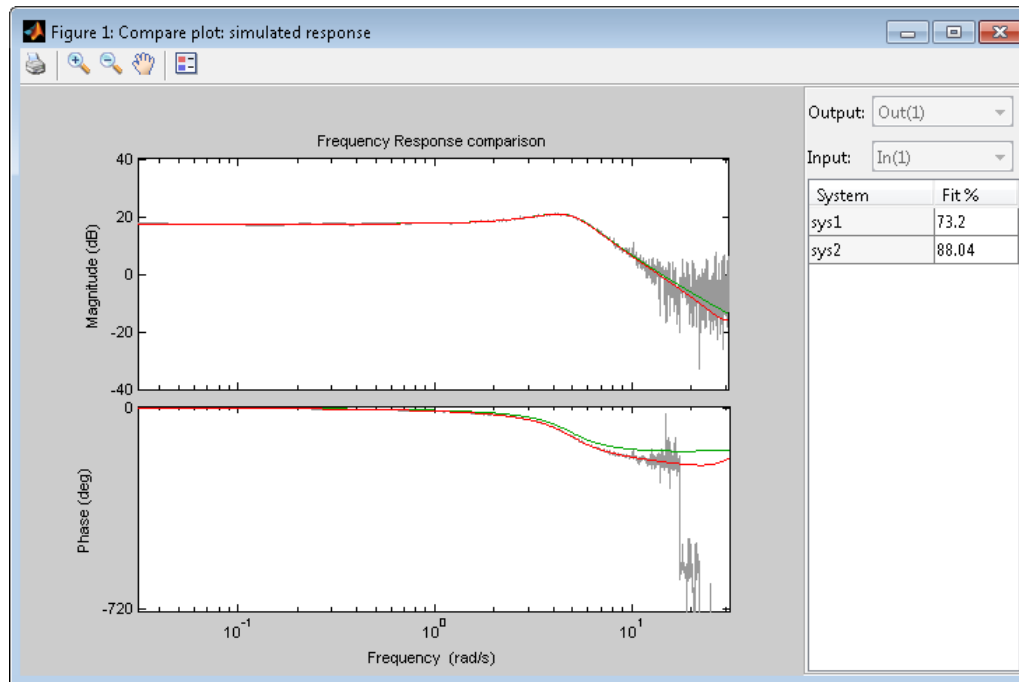
Estimate a process model and an Output-Error polynomial for frequency response data.

```
load demofr % frequency response data
zfr = AMP.*exp(1i*PHA*pi/180);
Ts = 0.1;
data = idfrd(zfr,W,Ts);
sys1 = procest(data,'P2UDZ');
sys2 = oe(data,[2 2 1]);
```

sys1, an idproc model, is a continuous-time process model. sys2, an idpoly model, is a discrete-time Output-Error model.

Compare the frequency response of the estimated models to data.

```
compare(data,sys1,'g',sys2,'r');
```



### Compare Estimated Model to Data and Specify Comparison Options

Compare an estimated model to measured data and specify that the initial conditions be estimated such that the prediction error of the observed output is minimized.

Estimate a transfer function for measured data.

# compare

```
load iddata1 z1;  
sys = tfest(z1,3)
```

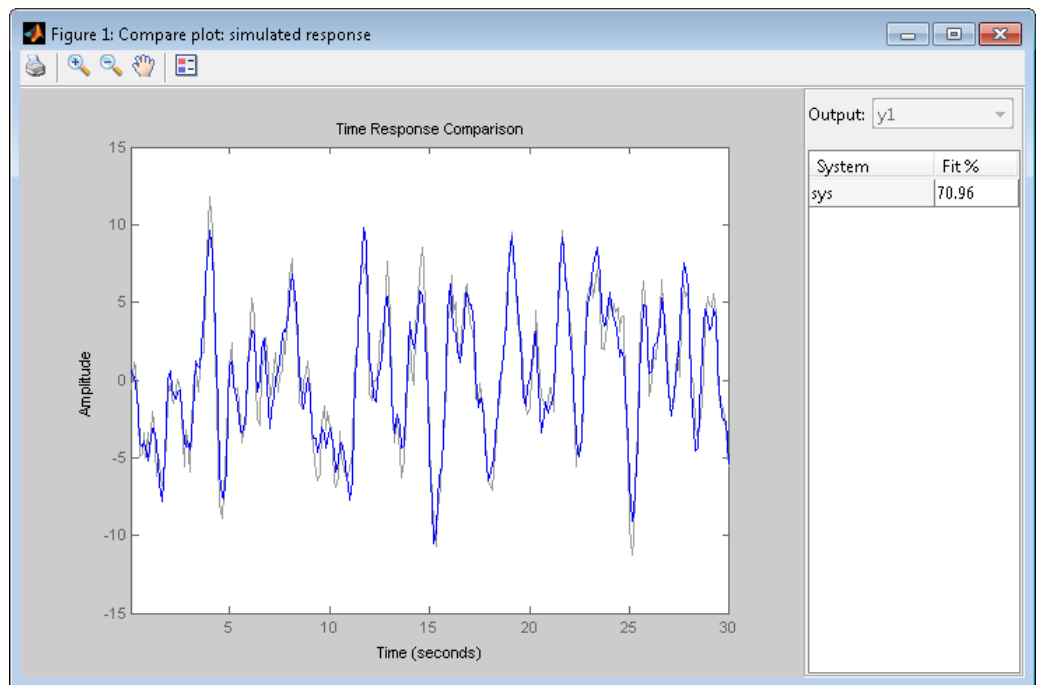
sys, an idtf model, is a continuous-time transfer function model.

Create an option set to specify the initial condition handling.

```
opt = compareOptions('InitialCondition','e');
```

Compare the estimated transfer function model's output to the measured data using the comparison option set.

```
compare(z1,sys,opt);
```



## See Also

`compareOptions` | `sim` | `predict` | `resid` | `forecast` | `interp`  
| `goodnessOfFit` | `chgTimeUnit` | `chgFreqUnit` | `bode`

# compareOptions

---

**Purpose** Option set for compare

**Syntax**  
opt = compareOptions  
opt = compareOptions(Name,Value)

**Description** opt = compareOptions creates the default options set for compare.  
opt = compareOptions(Name,Value) creates an option set with the options specified by one or more Name,Value pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments, where Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

### Samples

Data for which compare calculates fit values.

Specify Samples as a vector containing the data sample indices. For multiexperiment data, use a cell array of Ne vectors, where Ne is the number of experiments.

### InitialCondition

Specify how initial conditions are handled.

InitialCondition requires one of the following values:

- 'z' — Zero initial conditions.
- 'e' — Estimate initial conditions such that the prediction error for observed output is minimized.
- 'd' — Similar to 'e', but absorbs nonzero delays into the model coefficients. Use this option for discrete-time models only.

- `x0` — Numerical column vector denoting initial states. For multiexperiment data, use a matrix with  $N_e$  columns, where  $N_e$  is the number of experiments. Use this option for state-space models only.
- `io` — Structure with the following fields:
  - `Input`
  - `Output`

Use the `Input` and `Output` fields to specify the input/output history for a time interval that starts before the start time of the data used by `compare`. If the data used by `compare` is a time-series model, specify `Input` as `[]`. Use a row vector to denote a constant signal value. The number of columns in `Input` and `Output` must always equal the number of input and output channels, respectively. For multiexperiment data, specify `io` as a struct array of  $N_e$  elements, where  $N_e$  is the number of experiments.

- `x0obj` — Specification object created using `idpar`. Use this object for discrete-time state-space models only (`idss`, `idgrey`). Use `x0obj` to impose constraints on the initial states by fixing their value or specifying minimum/maximum bounds.

**Default:** `'e'`

## **InputOffset**

Removes offset from time domain input data for model response computation.

Specify as a column vector of length  $N_u$ , where  $N_u$  is the number of inputs.

Use `[]` to indicate no offset.

For multiexperiment data, specify `InputOffset` as a  $N_u$ -by- $N_e$  matrix. In this case,  $N_u$  is the number of inputs and  $N_e$  is the number of experiments.

Each entry specified by `InputOffset` is subtracted from the corresponding input data.

# compareOptions

---

**Default:** []

## **OutputOffset**

Removes offset from time domain output data for model response prediction.

Specify as a column vector of length  $N_y$ , where  $N_y$  is the number of outputs.

Use [] to indicate no offset.

For multiexperiment data, specify **OutputOffset** as a  $N_y$ -by- $N_e$  matrix. In this case,  $N_y$  is the number of outputs and  $N_e$  is the number of experiments.

Each entry specified by **OutputOffset** is subtracted from the corresponding output data.

**Default:** []

## **OutputWeight**

Weight of output for initial condition estimation.

**OutputWeight** requires one of the following values:

- [] — No weighting is used. This option is the same as using `eye(Ny)` for the output weight, where  $N_y$  is the number of outputs.
- 'noise' — Inverse of the noise variance stored with the model.
- matrix — A positive semi-definite matrix of dimension  $N_y$ -by- $N_y$ , where  $N_y$  is the number of outputs.

**Default:** []

## **Output Arguments**

**opt**

Option set containing the specified options for compare.



## Examples

### Create Default Options Set for Model Comparison

Create a default options set for compare.

```
opt = compareOptions;
```

### Specify Options for Model Comparison

Create an options set for compare using zero initial conditions. Set the input offset to 5.

```
opt = compareOptions('InitialCondition','z','InputOffset',5);
```

Alternatively, use dot notation to set the values of opt.

```
opt = compareOptions;  
opt.InitialCondition = 'z';  
opt.InputOffset = 5;
```

**See Also** [compare](#)

<b>Purpose</b>	Estimate impulse response using prewhitened-based correlation analysis
<b>Syntax</b>	<pre>ir=cra(data) [ir,R,cl] = cra(data,M,na,plot)</pre>
<b>Description</b>	<p><code>ir=cra(data)</code> estimates the impulse response for the time-domain data, <code>data</code>.</p> <p><code>[ir,R,cl] = cra(data,M,na,plot)</code> estimates correlation/covariance information, <code>R</code>, and the 99% confidence level for the impulse response, <code>cl</code>.</p> <p><code>cra</code> prewhitens the input sequence; that is, <code>cra</code> filters <code>u</code> through a filter chosen so that the result is as uncorrelated (white) as possible. The output <code>y</code> is subjected to the same filter, and then the covariance functions of the filtered <code>y</code> and <code>u</code> 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 <code>u</code> to later values of <code>y</code>. In other words, significant correlation for negative lags is an indication of feedback from <code>y</code> to <code>u</code> in the data.</p> <p>A properly scaled version of this correlation function is also an estimate of the system's impulse response <code>ir</code>. This is also graphed along with 99% confidence levels. The output argument <code>ir</code> is this impulse response estimate, so that its first entry corresponds to lag zero. (Negative lags are excluded in <code>ir</code>.) In the plot, the impulse response is scaled so that it corresponds to an impulse of height <math>1/T</math> and duration <math>T</math>, where <math>T</math> is the sampling interval of the data.</p>
<b>Input Arguments</b>	<p><b>data</b></p> <p>Input-output data.</p> <p>Specify <code>data</code> as an <code>iddata</code> object containing time-domain data only.</p>

---

data should contain data for a single-input, single-output experiment. For the multivariate case, apply `cra` to two signals at a time, or use `impulse`.

**M**

Number of lags for which the covariance/correlation functions are computed.

`M` specifies 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  $M$ .

**Default:** 20

**na**

Order of the AR model to which the input is fitted.

For the prewhitening, the input is fitted to an AR model of order `na`.

Use `na = 0` to obtain the covariance and correlation functions of the original data sequences.

**Default:** 10

**plot**

Plot display control.

Specify `plot` as one of the following integers:

- 0 — No plots are displayed.
- 1 — Plots the estimated impulse response with a 99% confidence region.
- 2 — Plots all the covariance functions.

**Default:** 1

**Output Arguments****ir**

Estimated impulse response.

The first entry of `ir` corresponds to lag zero. (Negative lags are excluded in `ir`.)

**R**

Covariance/correlation information.

- 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)`.

**cl**

99 % significance level for the impulse response.

**Examples**

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

```
load iddata1
z=z1;
ir = cra(z);
m = arx(z,[2 2 1]);
imp = [1;zeros(20,1)];
irth = sim(m,imp);
subplot(211)
plot([ir irth])
title('impulse responses')
subplot(212)
plot([cumsum(ir),cumsum(irth)])
title('step responses')
```

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

**See Also** `impulse` | `step` | `impulseest` | `covf` | `spa`

<b>Purpose</b>	Custom nonlinearity estimator for nonlinear ARX and Hammerstein-Wiener models
<b>Syntax</b>	<code>C=customnet(H)</code> <code>C=customnet(H,PropertyName,PropertyValue)</code>
<b>Description</b>	<code>customnet</code> is an object that stores a custom nonlinear estimator with a user-defined unit function. This custom unit function uses a weighted sum of inputs to compute a scalar output.
<b>Construction</b>	<code>C=customnet(H)</code> creates a nonlinearity estimator object with a user-defined unit function using the function handle <code>H</code> . <code>H</code> must point to a function of the form <code>[f,g,a] = gaussunit(x)</code> , where <code>f</code> is the value of the function, <code>g=df/dx</code> , and <code>a</code> indicates the unit function active range. Name the function <code>gaussunit.m</code> . <code>g</code> is significantly nonzero in the interval <code>[-a a]</code> . Hammerstein-Wiener models require that your custom nonlinearity have only one input and one output.  <code>C=customnet(H,PropertyName,PropertyValue)</code> creates a nonlinearity estimator using property-value pairs defined in “customnet Properties” on page 2-149.
<b>Tips</b>	Use <code>customnet</code> 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 $L$ : $F(x) = (x - r)PL + \alpha_1 f((x - r)Qb_1 + c_1) + \dots + \alpha_n f((x - r)Qb_n + c_n) + d$ 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.

$b$  represents  $q$ -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	<p>Integer specifies the number of nonlinearity units in the expansion. Default=10.</p> <p>For example:</p> <pre>customnet(H, 'NumberOfUnits', 5)</pre>
LinearTerm	<p>Can have the following values:</p> <ul style="list-style-type: none"> <li>• 'on'—Estimates the vector <math>L</math> in the expansion.</li> <li>• 'off'—Fixes the vector <math>L</math> to zero.</li> </ul> <p>For example:</p> <pre>customnet(H, 'LinearTerm', 'on')</pre>
Parameters	<p>A structure containing the parameters in the nonlinear expansion, as follows:</p> <ul style="list-style-type: none"> <li>• RegressorMean: 1-by-m vector containing the means of <math>x</math> in estimation data, <math>r</math>.</li> <li>• NonLinearSubspace: m-by-q matrix containing <math>Q</math>.</li> <li>• LinearSubspace: m-by-p matrix containing <math>P</math>.</li> <li>• LinearCoef: p-by-1 vector <math>L</math>.</li> <li>• Dilation: q-by-1 matrix containing the values <math>b_n</math>.</li> <li>• Translation: 1-by-n vector containing the values <math>c_n</math>.</li> <li>• OutputCoef: n-by-1 vector containing the values <math>a_n</math>.</li> <li>• OutputOffset: scalar <math>d</math>.</li> </ul> <p>Typically, the values of this structure are set by estimating a model with a customnet nonlinearity.</p>
UnitFcn	Stores the function handle that points to the unit function.



## Algorithms

customnet uses an iterative search technique for estimating parameters.

## Examples

Define custom unit function and save it in `gaussunit.m`:

```
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 nargin>1
    g = - 2*x.*f;
    a = 0.2;
end
```

---

Use custom networks in `nlarx` and `nlnhw` model estimation commands:

```
% 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](#) | [nlnhw](#)

## How To

- “Identifying Nonlinear ARX Models”

- “Identifying Hammerstein-Wiener Models”

**Purpose**

Custom regressor for nonlinear ARX models

**Syntax**

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

**Description**

customreg class represents arbitrary functions of past inputs and outputs, such as products, powers, and other MATLAB expressions of input and output variables.

You can specify custom regressors in addition to or instead of standard regressors for greater flexibility in modeling your data using nonlinear ARX models. For example, you can define regressors like  $\tan(u(t-1))$ ,  $u(t-1)^2$ , and  $u(t-1)*y(t-3)$ .

For simpler regressor expressions, specify custom regressors directly in the GUI or in the `nlarx` estimation command. For more complex expressions, create a `customreg` object for each custom regressor and specify these objects as inputs to the estimation. Regardless of how you specify custom regressors, the toolbox represents these regressors as `customreg` objects. Use `getreg` to list the expressions of all standard and custom regressors in your model.

A special case of custom regressors involves polynomial combinations of past inputs and outputs. For example, it is common to capture nonlinearities in the system using polynomial expressions like  $y(t-1)^2$ ,  $u(t-1)^2$ ,  $y(t-2)^2$ ,  $y(t-1)*y(t-2)$ ,  $y(t-1)*u(t-1)$ ,  $y(t-2)*u(t-1)$ . At the command line, use the `polyreg` command to generate polynomial-type regressors automatically by computing all combinations of input and output variables up to a specified degree. `polyreg` produces `customreg` objects that you specify as inputs to the estimation.

The nonlinear ARX model (`idnlarx` object) stores all custom regressors as the `CustomRegressors` property. You can list all custom regressors using `m.CustomRegressors`, where `m` is a nonlinear ARX model. For MIMO models, to retrieve the `r`th custom regressor for output `ky`, use `m.CustomRegressors{ky}(r)`.

Use the `Vectorized` property to specify whether to compute custom regressors using vectorized form during estimation. If you know

that your regressor formulas can be vectorized, set `Vectorized` to 1 to achieve better performance. To better understand vectorization, consider the custom regressor function handle `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  $(x_i, y_i)$  pair, and the results are concatenated to produce a `z` vector:

```
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:

```
% "." indicates element-wise operation
z=(x.^2).*y
```

### Construction

`C=customreg(Function,Variables)` specifies a custom regressor for a nonlinear ARX model. `C` is a `customreg` object that stores custom regressor. `Function` is a handle or string representing a function of input and output variables. `Variables` is a cell array of strings that represent the names of model inputs and outputs 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 `Variables` must match the number of `Function` inputs. For multiple-output models with `p` outputs, the custom regressor is a `p`-by-1 cell array or an array of `customreg` object, where the `ky`th entry defines the custom regressor for output `ky`. You must add these regressors to the `model` by assigning the `CustomRegressors` `model` property or by using `addreg`.

`C=customreg(Function,Variables,Delays,Vectorized)` create a custom regressor that includes the delays corresponding to inputs or outputs in `Arguments`. `Delays` is a vector of positive integers that represent the delays of `Variables` variables (default is 1 for each vector element). The size of `Delays` must match the size of `Variables`.

*Vectorized* value of 1 uses MATLAB vectorization rules to evaluate the regressor expression *Function*. By default, *Vectorized* value is 0 (false).

## 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(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', 1)
```

Property Name	Description
Function	Function handle or string representing a function of standards regressors.  For example:  <code>cr = @(x,y) x*y</code>
Variables	Cell array of strings that represent the names of model input and output variables in the function <code>Function</code> . Each input and output name must coincide with the strings in the <code>InputName</code> and <code>OutputName</code> properties of the <code>idnlarx</code> object—the model for which you define custom regressors. The size of <code>Variables</code> must match the number of <code>Function</code> inputs.  For example, <code>Variables</code> correspond to <code>{'y1', 'u1'}</code> in:  <code>C = customreg(cr, {'y1', 'u1'}, [2 3])</code>

# customreg

Property Name	Description
Delays	<p>Vector of positive integers representing the delays of Variables. The size of Delays must match the size of Arguments.</p> <p>Default: 1 for each vector element.</p> <p>For example, Delays are [2 3] in:</p> <pre>C = customreg(cr,{'y1','u1'},[2 3])</pre>
Vectorized	<p>Assignable values:</p> <ul style="list-style-type: none"><li>• 0 (default)—Function is not computed in vectorized form.</li><li>• 1—Function is computed in vectorized form when called with vector arguments.</li></ul>

## Examples

Define custom regressors as a cell array of strings:

```
load iddata1
m = nlarx(z1,[2 2 1]);
C={'u1(t-1)*sin(y1(t-3))','u1(t-2)^3'};
% u1 and y1 are system input and output

m.CustomRegressors = C;
m=pem(z1,m)
```

Define custom regressors directly in the estimation command nlarx:

```
m = nlarx(data,[na nb nk],'linear',...
          'CustomRegressors',...
          {'u1(t-1)*sin(y1(t-3))','u1(t-2)^3'});
```

Define custom regressors as an object array of customreg objects:

```

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

```

---

Use vectorization rules to evaluate regressor expression during estimation:

```

C = customreg(@(x,y) x*sin(y),{'u' 'y'},[1 3])
set(C,'Vectorized',1)
m = nlarx(data,[na nb nk],'sigmoidnet','CustomReg',C)

```

**See Also**

addreg | getreg | idnlarx | nlarx | polyreg

**How To**

- “Identifying Nonlinear ARX Models”

**Purpose** Convert model from discrete to continuous time

**Syntax**

```
sysc = d2c(sysd)
sysc = d2c(sysd,method)
sysc = d2c(sysd,opts)
[sysc,G] = d2c(sysd,method,opts)
```

**Description**

`sysc = d2c(sysd)` produces a continuous-time model `sysc` that is equivalent to the discrete-time dynamic system model `sysd` using zero-order hold on the inputs.

`sysc = d2c(sysd,method)` uses the specified conversion method `method`.

`sysc = d2c(sysd,opts)` converts `sysd` using the option set `opts`, specified using the `d2cOptions` command.

`[sysc,G] = d2c(sysd,method,opts)` returns a matrix `G` that maps the states `xd[k]` of the state-space model `sysd` to the states `xc(t)` of `sysc`.

**Tips**

- Use the syntax `sysc = d2c(sysd,'method')` to convert `sysd` using the default options for `'method'`. To specify `tustin` conversion with a frequency prewarp (formerly the `'prewarp'` method), use the syntax `sysc = d2c(sysd,opts)`. See the `d2cOptions` reference page for more information.

**Input Arguments**

**sysd**  
Discrete-time dynamic system model

You cannot directly use an `idgrey` model with `FcnType='d'` with `d2c`. Convert the model into `idss` form first.

**method**  
String specifying a discrete-to-continuous time conversion method:

- `'zoh'` — Zero-order hold on the inputs. Assumes the control inputs are piecewise constant over the sampling period.



- 'foh' — Linear interpolation of the inputs (modified first-order hold). Assumes the control inputs are piecewise linear over the sampling period.
- 'tustin' — Bilinear (Tustin) approximation to the derivative.
- 'matched' — Zero-pole matching method of [1] (for SISO systems only).

**Default:** 'zoh'

### **opts**

Discrete-to-continuous time conversion options, created using `d2cOptions`.

## **Output Arguments**

### **sysc**

Continuous-time model of the same type as the input system `sysd`.

When `sysd` is an identified (IDLTI) model, `sysc`:

- Includes both the measured and noise components of `sysd`. If the noise variance is  $\lambda$  in `sysd`, then the continuous-time model `sysc` has an indicated level of noise spectral density equal to  $T_s*\lambda$ .
- Does not include the estimated parameter covariance of `sysd`. If you want to translate the covariance while converting the model, use `translatecov`.

### **G**

Matrix mapping the states  $x_d[k]$  of the state-space model `sysd` to the states  $x_c(t)$  of `sysc`:

$$x_c(kT_s) = G \begin{bmatrix} x_d[k] \\ u[k] \end{bmatrix}.$$

Given an initial condition  $x_0$  for `sysd` and an initial input  $u_0 = u[0]$ , the corresponding initial condition for `sysc` (assuming  $u[k] = 0$  for  $k < 0$ ) is given by:

$$x_c(0) = G \begin{bmatrix} x_0 \\ u_0 \end{bmatrix}.$$

## Examples

### Example 1

Consider the discrete-time model with transfer function

$$H(z) = \frac{z-1}{z^2+z+0.3}$$

and sample time  $T_s = 0.1$  s. You can derive a continuous-time zero-order-hold equivalent model by typing

```
Hc = d2c(H)
```

Discretizing the resulting model `Hc` with the default zero-order hold method and sampling time  $T_s = 0.1$ s returns the original discrete model  $H(z)$ :

```
c2d(Hc,0.1)
```

To use the Tustin approximation instead of zero-order hold, type

```
Hc = d2c(H, 'tustin')
```

As with zero-order hold, the inverse discretization operation

```
c2d(Hc,0.1, 'tustin')
```

gives back the original  $H(z)$ .

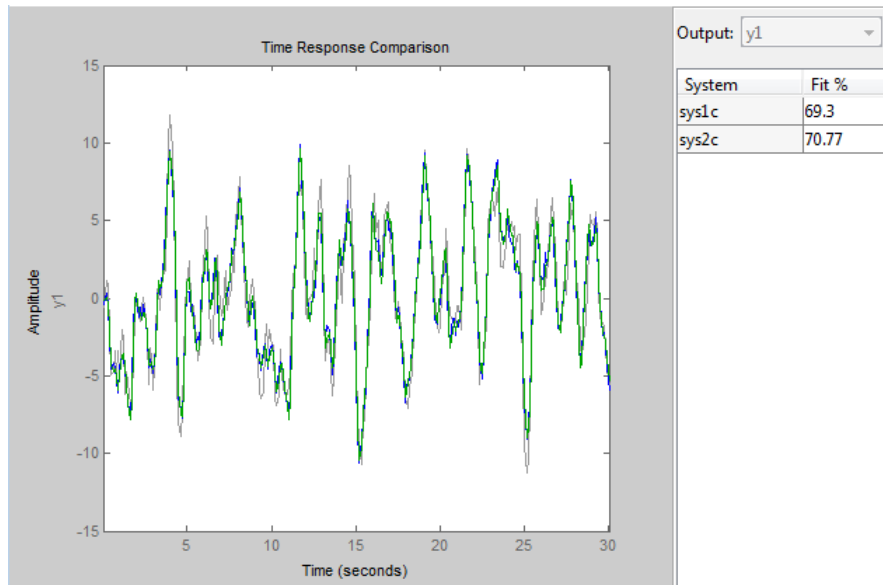
## Example 2

Convert an identified transfer function and compare its performance against a directly estimated continuous-time model.

```
load iddata1
sys1d = tfest(z1, 2, 'Ts', 0.1);
sys1c = d2c(sys1d, 'zoh');
sys2c = tfest(z1, 2);
```

```
compare(z1, sys1c, sys2c)
```

The two systems are virtually identical.



## Example 3

Analyze the effect of parameter uncertainty on frequency response across d2c operation on an identified model.

```
load iddata1
```

```
sysd = tfest(z1, 2, 'Ts', 0.1);  
sysc = d2c(sysd, 'zoh');
```

`sys1c` has no covariance information. Regenerate it using a zero iteration update with the same estimation command and estimation data:

```
opt = tfestOptions;  
opt.SearchOption.MaxIter = 0;  
sys1c = tfest(z1, sysc, opt);
```

```
h = bodeplot(sysd, sysc);  
showConfidence(h)
```

The uncertainties of `sysc` and `sysd` are comparable up to the Nyquist frequency. However, `sysc` exhibits large uncertainty in the frequency range for which the estimation data does not provide any information.

If you do not have access to the estimation data, use `translatecov` which is a Gauss-approximation formula based translation of covariance across model type conversion operations.

## Algorithms

`d2c` performs the 'zoh' conversion in state space, and relies on the matrix logarithm (see `logm` in the MATLAB documentation).

See “Continuous-Discrete Conversion Methods” for more details on the conversion methods.

## Limitations

The Tustin approximation is not defined for systems with poles at  $z = -1$  and is ill-conditioned for systems with poles near  $z = -1$ .

The zero-order hold method cannot handle systems with poles at  $z = 0$ . In addition, the 'zoh' conversion increases the model order for systems with negative real poles, [2]. The model order increases because the matrix logarithm maps real negative poles to complex poles. Single complex poles are not physically meaningful because of their complex time response.

Instead, to ensure that all complex poles of the continuous model come in conjugate pairs, `d2c` replaces negative real poles  $z = -a$  with a pair of complex conjugate poles near  $-a$ . The conversion then yields a continuous model with higher order. For example, to convert the discrete-time transfer function

$$H(z) = \frac{z + 0.2}{(z + 0.5)(z^2 + z + 0.4)}$$

type:

```
Ts = 0.1 % sample time 0.1 s
H = zpk(-0.2,-0.5,1,Ts) * tf(1,[1 1 0.4],Ts)
Hc = d2c(H)
```

These commands produce the following result.

Warning: System order was increased to handle real negative poles.

```
Zero/pole/gain:
-33.6556 (s-6.273) (s^2 + 28.29s + 1041)
-----
(s^2 + 9.163s + 637.3) (s^2 + 13.86s + 1035)
```

To convert `Hc` back to discrete time, type:

```
c2d(Hc,Ts)
```

yielding

```
Zero/pole/gain:
(z+0.5) (z+0.2)
-----
(z+0.5)^2 (z^2 + z + 0.4)
```

```
Sampling time: 0.1
```

This discrete model coincides with  $H(z)$  after canceling the pole/zero pair at  $z = -0.5$ .

**References**

[1] Franklin, G.F., Powell, D.J., and Workman, M.L., *Digital Control of Dynamic Systems* (3rd Edition), Prentice Hall, 1997..

[2] Kollár, I., G.F. Franklin, and R. Pintelon, "On the Equivalence of z-domain and s-domain Models in System Identification," *Proceedings of the IEEE® Instrumentation and Measurement Technology Conference*, Brussels, Belgium, June, 1996, Vol. 1, pp. 14-19.

**See Also**

[d2cOptions](#) | [c2d](#) | [d2d](#) | [translatecov](#) | [logm](#)

**Purpose** Create option set for discrete- to continuous-time conversions

**Syntax**  
`opts = d2cOptions`  
`opts = d2cOptions(Name,Value)`

**Description**  
`opts = d2cOptions` returns the default options for `d2c`.  
`opts = d2cOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### Name-Value Pair Arguments

#### method

Discretization method, specified as one of the following values:

'zoh'	Zero-order hold, where <code>d2c</code> assumes the control inputs are piecewise constant over the sampling period <code>Ts</code> .
'foh'	Linear interpolation of the inputs (modified first-order hold). Assumes the control inputs are piecewise linear over the sampling period.
'tustin'	Bilinear (Tustin) approximation. By default, <code>d2c</code> converts with no prewarp. To include prewarp, use the <code>PrewarpFrequency</code> option.
'matched'	Zero-pole matching method. (See [1], p. 224.)

**Default:** 'zoh'

#### PrewarpFrequency

Prewarp frequency for 'tustin' method, specified in `rad/TimeUnit`, where `TimeUnit` is the time units, specified in the `TimeUnit` property, of the discrete-time system. Specify the prewarp frequency as a positive scalar value. A value of 0 corresponds to the 'tustin' method without prewarp.

**Default:** 0

For additional information about conversion methods, see “Continuous-Discrete Conversion Methods”.

## Examples

Convert a discrete-time model to continuous-time using the 'tustin' method with frequency prewarping.

Create the discrete-time transfer function

$$\frac{z+1}{z^2+z+1}$$

```
hd = tf([1 1], [1 1 1], 0.1); % 0.1s sampling time
```

To convert to continuous-time, use `d2cOptions` to create the option set.

```
opts = d2cOptions('Method', 'tustin', 'PrewarpFrequency', 20);  
hc = d2c(hd, opts);
```

You can use `opts` to resample additional models using the same options.

## References

[1] Franklin, G.F., Powell, D.J., and Workman, M.L., *Digital Control of Dynamic Systems* (3rd Edition), Prentice Hall, 1997.

## See Also

`d2c`



**Purpose** Resample discrete-time model

**Syntax**

```
sys1 = d2d(sys, Ts)
sys1 = d2d(sys, Ts, 'method')
sys1 = d2d(sys, Ts, opts)
```

**Description** `sys1 = d2d(sys, Ts)` resamples the discrete-time dynamic system model `sys` to produce an equivalent discrete-time model `sys1` with the new sample time `Ts` (in seconds), using zero-order hold on the inputs.

`sys1 = d2d(sys, Ts, 'method')` uses the specified resampling method `'method'`:

- `'zoh'` — Zero-order hold on the inputs
- `'tustin'` — Bilinear (Tustin) approximation

`sys1 = d2d(sys, Ts, opts)` resamples `sys` using the option set with `d2dOptions`.

- Tips**
- Use the syntax `sys1 = d2d(sys, Ts, 'method')` to resample `sys` using the default options for `'method'`. To specify `tustin` resampling with a frequency prewarp (formerly the `'prewarp'` method), use the syntax `sys1 = d2d(sys, Ts, opts)`. See the `d2dOptions` reference page.
  - When `sys` is an identified (IDLTI) model, `sys1` does not include the estimated parameter covariance of `sys`. If you want to translate the covariance while converting the model, use `translatecov`.

## Examples

### Example 1

Consider the zero-pole-gain model

$$H(z) = \frac{z - 0.7}{z - 0.5}$$

with sample time 0.1 s. You can resample this model at 0.05 s by typing

```
H = zpkm(0.7,0.5,1,0.1)
```

```
H2 = d2d(H,0.05)
Zero/pole/gain:
(z-0.8243)
-----
(z-0.7071)
```

```
Sampling time: 0.05
```

The inverse resampling operation, performed by typing `d2d(H2,0.1)`, yields back the initial model  $H(z)$ .

```
Zero/pole/gain:
(z-0.7)
-----
(z-0.5)
```

```
Sampling time: 0.1
```

## Example 2

Suppose you estimate a discrete-time model of a sample time commensurate with the estimation data ( $T_s = 0.1$  seconds). However, your deployment application demands a faster sampling frequency ( $T_s = 0.01$  seconds).

```
load iddata1
sys = oe(z1, [2 2 1]);
sysFast = d2d(sys, 0.01, 'zoh')

bode(sys, sysFast)
```

## See Also

[d2dOptions](#) | [c2d](#) | [d2c](#) | [upsample](#) | [translatecov](#)

**Purpose** Create option set for discrete-time resampling

**Syntax**  
`opts = d2dOptions`  
`opts = d2dOptions('OptionName', OptionValue)`

**Description** `opts = d2dOptions` returns the default options for `d2d`.  
`opts = d2dOptions('OptionName', OptionValue)` accepts one or more comma-separated name/value pairs that specify options for the `d2d` command. Specify *OptionName* inside single quotes.

This table summarizes the options that the `d2d` command supports.

## Input Arguments

### Name-Value Pair Arguments

#### Method

Discretization method, specified as one of the following values:

- |          |  |
|----------|--|
| 'zoh'    | Zero-order hold, where <code>d2d</code> assumes the control inputs are piecewise constant over the sampling period $T_s$ .                                 |
| 'tustin' | Bilinear (Tustin) approximation. By default, <code>d2d</code> resamples with no prewarp. To include prewarp, use the <code>PrewarpFrequency</code> option. |

**Default:** 'zoh'

#### PrewarpFrequency

Prewarp frequency for 'tustin' method, specified in `rad/TimeUnit`, where `TimeUnit` is the time units, specified in the `TimeUnit` property, of the resampled system. Takes positive scalar values. The prewarp frequency must be smaller than the Nyquist frequency before and after resampling. A value of 0 corresponds to the standard 'tustin' method without prewarp.

**Default:** 0

# d2dOptions

---

## Examples

Resample a discrete-time model using the 'tustin' method with frequency prewarping.

Create the discrete-time transfer function

$$\frac{z+1}{z^2+z+1}$$

```
h1 = tf([1 1], [1 1 1], 0.1); % 0.1s sampling time
```

To resample to a different sampling time, use `d2dOptions` to create the option set.

```
opts = d2dOptions('Method', 'tustin', 'PrewarpFrequency', 20);  
h2 = d2d(h1, 0.05, opts);
```

You can use `opts` to resample additional models using the same options.

## See Also

`d2d`

---

<b>Purpose</b>	Natural frequency; damping ratio
<b>Syntax</b>	<pre>damp(sys) [Wn,zeta] = damp(sys) [Wn,zeta,P] = damp(sys)</pre>
<b>Description</b>	<p><code>damp(sys)</code> calculates the damping ratio (also called damping factor) and natural frequency of the poles of the linear model <code>sys</code>. When invoked without output arguments, <code>damp</code> displays a table of the eigenvalues of <code>sys</code>, along with the corresponding damping ratios and natural frequencies. For discrete-time <code>sys</code>, the table includes the magnitude of each pole and the damping ratio and frequencies of equivalent continuous-time poles (see “Algorithms” on page 2-173). Frequencies are expressed in units of the reciprocal of the <code>TimeUnit</code> property of <code>sys</code>.</p> <p><code>[Wn,zeta] = damp(sys)</code> returns vectors <code>Wn</code> and <code>zeta</code> containing the natural frequencies <math>\omega_n</math> and damping ratios <math>\zeta</math> of the poles of <code>sys</code>.</p> <p><code>[Wn,zeta,P] = damp(sys)</code> also returns a vector <code>P</code> containing the poles of <code>sys</code>.</p>
<b>Input Arguments</b>	<p><b>sys</b></p> <p>Any linear dynamic system model.</p>
<b>Output Arguments</b>	<p><b>Wn</b></p> <p>Vector containing the natural frequencies of each pole of <code>sys</code>, in order of increasing frequency. Frequencies are expressed in units of the reciprocal of the <code>TimeUnit</code> property of <code>sys</code>.</p> <p>If <code>sys</code> is a discrete-time model with specified sampling time, <code>Wn</code> contains the natural frequencies of the equivalent continuous-time poles (see “Algorithms” on page 2-173). If <code>sys</code> has unspecified sampling time (<code>Ts = -1</code>), <code>Wn</code> is empty.</p> <p><b>zeta</b></p>

Vector containing the damping ratios of each pole of `sys`, in the same order as `Wn`.

If `sys` is a discrete-time model with specified sampling time, `zeta` contains the damping ratios of the equivalent continuous-time poles (see “Algorithms” on page 2-173). If `sys` has unspecified sampling time (`Ts = -1`), `zeta` is empty.

## **P**

Vector containing the poles of `sys`, in order of increasing natural frequency. `P` is the same as the output of `pole(sys)`, up to ordering.

## **Examples**

### **Natural Frequency, Damping Ratio, and Poles of a Continuous-Time Transfer Function**

Compute the natural frequency, damping ratio and poles of a continuous-time transfer function.

Create the transfer function:

$$H(s) = \frac{2s^2 + 5s + 1}{s^2 + 2s + 3}$$

```
H = tf([2 5 1],[1 2 3]);
```

Display the natural frequencies, damping ratios, and poles of `H`.

```
damp(H)
```

Eigenvalue	Damping	Frequency
-1.00e+000 + 1.41e+000i	5.77e-001	1.73e+000
-1.00e+000 - 1.41e+000i	5.77e-001	1.73e+000

(Frequencies expressed in rad/seconds)

The system eigenvalues are the pole locations.

Obtain vectors containing the natural frequencies and damping ratios of the poles.

```
[Wn,zeta] = damp(H);
```

---

### Natural Frequency, Damping Ratio and Poles of a Discrete-Time Transfer Function

Compute the natural frequency, damping ratio and poles of a discrete-time transfer function.

```
H = tf([5 3 1],[1 6 4 4],0.01);
```

Display information about the poles of  $H$ .

```
damp(H)
```

Eigenvalue	Magnitude	Damping	Frequency
-3.02e-001 + 8.06e-001i	8.61e-001	7.74e-002	1.93e+002
-3.02e-001 - 8.06e-001i	8.61e-001	7.74e-002	1.93e+002
-5.40e+000	5.40e+000	-4.73e-001	3.57e+002

(Frequencies expressed in rad/seconds)

The system eigenvalues are the pole locations.

Obtain vectors containing the natural frequencies and damping ratios of the poles.

```
[Wn,zeta] = damp(H);
```

## Algorithms

For a continuous-time linear system  $G(s)$ , the natural frequency  $\omega_n$  of a pole at  $s = R$  is given by:

$$\omega_n = |R|.$$

# damp

For a discrete-time linear system  $G(z)$  with a pole at  $z = R$ , `damp` returns the natural frequencies and damping ratios of equivalent continuous time poles. The locations of the equivalent poles are given by

$$s = \frac{\ln(R)}{T_s}$$

$T_s$  is the sampling time.

The natural frequency, time constant, and damping ratio of the system poles are defined as follows.

	<b>Continuous Time</b>	<b>Discrete Time</b>
<b>Location of Pole</b>	Real or complex eigenvalue at $s = R$	Real or complex eigenvalue at $z = R$
<b>Natural Frequency</b>	$\text{Wn} = \text{abs}(R)$	$\text{Wn} = \text{abs}(\log(R)) / T_s$
<b>Damping Ratio</b>	$\text{zeta} = -\cos(\text{angle}(R))$	$\text{zeta} = -\cos(\text{angle}(\log(R)))$
<b>Time Constant</b>	<ul style="list-style-type: none"><li>• <math>\text{tau} = 1 / (\text{zeta} * \text{Wn})</math> for <math>\text{zeta} &gt; 0</math></li><li>• Inf otherwise</li></ul>	<ul style="list-style-type: none"><li>• <math>\text{tau} = 1 / (\text{zeta} * \text{Wn})</math> for <math>\text{zeta} &gt; 0</math></li><li>• Inf otherwise</li></ul>

## See Also

`eig` | `esort` | `dsort` | `pole` | `pzmap` | `zero`



**Purpose** Map past input/output data to current states of nonlinear ARX 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 2-354. 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 Arguments

- `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 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 steady-state (constant) output values.

## data2state(idnlarx)

---

- 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` command. For more information, see the `getDelayInfo` reference page.

---

### Output Arguments

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 an `idnlarx` model from the data. The model has 6 inputs and 2 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);
```

- 4 Represent the past input and output samples:

```
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 scalar values (10,20,30,40,50,60) for the input variables by setting `IOSTRUCT.Input = [10, 20, 30, 40, 50, 60]` instead of a matrix of values.

---

### See Also

[findop\(idnlarx\)](#) | [findstates\(idnlarx\)](#) | [getDelayInfo](#)

# db2mag

---

**Purpose** Convert decibels (dB) to magnitude

**Syntax** `y = db2mag(ydb)`

**Description** `y = db2mag(ydb)` returns the corresponding magnitude  $y$  for a given decibel (dB) value  $ydb$ . The relationship between magnitude and decibels is  $ydb = 20 * \log_{10}(y)$ .

**See Also** `mag2db`

**Purpose** Low-frequency (DC) gain of LTI system

**Syntax** `k = dcgain(sys)`

**Description** `k = dcgain(sys)` computes the DC gain `k` of the LTI model `sys`.

**Continuous Time**

The continuous-time DC gain is the transfer function value at the frequency  $s = 0$ . For state-space models with matrices  $(A, B, C, D)$ , this value is

$$K = D - CA^{-1}B$$

**Discrete Time**

The discrete-time DC gain is the transfer function value at  $z = 1$ . For state-space models with matrices  $(A, B, C, D)$ , this value is

$$K = D + C(I - A)^{-1}B$$

**Tips** The DC gain is infinite for systems with integrators.

**Examples** **Example 1**

To compute the DC gain of the MIMO transfer function

$$H(s) = \begin{bmatrix} 1 & \frac{s-1}{s^2+s+3} \\ \frac{1}{s+1} & \frac{s+2}{s-3} \end{bmatrix}$$

type

```
H = [1 tf([1 -1],[1 1 3]) ; tf(1,[1 1]) tf([1 2],[1 -3])];
dcgain(H)
```

to get the result:

# dcgain

---

```
ans =  
    1.0000   -0.3333  
    1.0000   -0.6667
```

## Example 2

To compute the DC gain of an identified process model, type;

```
load iddata1  
sys = idproc('p1d');  
syse = procest(z1, sys)
```

```
dcgain(syse)
```

The DC gain is stored same as `syse.Kp`.

## See Also

[evalfr](#) | [norm](#)

## 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.

## Tips

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

$$\begin{array}{ll} a \leq x < b & F(x) = 0 \\ x < a & F(x) = x - a \\ x \geq b & F(x) = x - b \end{array}$$

$y$  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:

# deadzone

---

```
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: <code>deadzone('ZeroInterval', [-1.5 1.5])</code>

## Examples

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

```
m=n1hw(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 `n1hw`.

## See Also

`n1hw`



**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:

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

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).

# detrend

---

**Purpose** Subtract offset or trend from data signals

**Syntax**

```
data_d = detrend(data)
data_d = detrend(data,Type)
[data_d,T] = detrend(data,Type)
data_d = detrend(data,1,brkp)
```

**Description**

`data_d = detrend(data)` subtracts the mean value from each time-domain or time-series signal `data`. `data_d` and `data` are `iddata` objects.

`data_d = detrend(data,Type)` subtracts a mean value from each signal when `Type = 0`, a linear trend (least-squares fit) when `Type = 1`, or a trend specified by a `TrendInfo` object when `Type = T`.

`[data_d,T] = detrend(data,Type)` stores the trend information as a `TrendInfo` object `T`.

`data_d = detrend(data,1,brkp)` subtracts a piecewise linear trend at one or more breakpoints `brkp`. `brkp` is a data index where discontinuities between successive linear trends occur. When `brkp` contains breakpoints that match the time vector, `detrend` subtracts a continuous piecewise linear trend. You cannot store piecewise linear trend information as an output argument.

**Examples** Subtract mean values from input and output signals and store the trend information:

```
% Load SISO data containing vectors u2 and y2.
load dryer2
% Create data object with sampling interval of 0.08 sec.
data=iddata(y2,u2,0.08)
% Plot data on a time plot. Data has a nonzero mean.
plot(data)
% Remove the mean from the data.
[data_d,T] = detrend(data,0)
% Plot detrended data on the same plot.
hold on
```

```
plot(data_d)
```

Remove specified offset from input and output signals:

```
% Load SISO data containing vectors u2 and y2.
load dryer2
% Create data object with sampling time of 0.08 sec.
data=iddata(y2,u2,0.08)
plot(data)
% Create a TrendInfo object for storing offsets and trends.
T = getTrend(data)
% Assign offset values to the TrendInfo object.
T.InputOffset=5;
T.OutputOffset=5;
% Subtract offset from the data.
data_d = detrend(data,T)
% Plot detrended data on the same plot.
hold on
plot(data_d)
```

Subtract several linear trends that connect at three breakpoints [30 60 90]:

```
data = detrend(data,1,[30 60 90]);
% [30 60 90] are data indexes where breakpoints occur.
```

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

## See Also

getTrend | TrendInfo

## How To

- “Handling Offsets and Trends in Data”

# diff

---

**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**

Estimate empirical transfer functions and periodograms

**Syntax**

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

**Description**

etfe estimates the transfer function `g` as an `idfrd` object of the general linear model:

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

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

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

$$w = [1:N]/N*\pi/T$$

The default value of `N` is 128.

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

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

When `etfe` is applied to time series, the corresponding spectral estimate is normalized in the way that is defined in "Spectrum Normalization". `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  $k*2*\pi/\text{period}$  for `k = 0` up to the Nyquist frequency.

## Examples

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

```
load iddata z1;
ge = etfe(z1);
gs = spa(z1);
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, 'r')
```

## Algorithms

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` | `freqresp` | `idfrd` | `nyquist` | `spa` | `spafdr` | `impulseest`  
| `spectrum`

**Purpose** Evaluate frequency response at given frequency

**Syntax** `frsp = evalfr(sys,f)`

**Description** `frsp = evalfr(sys,f)` evaluates the transfer function of the TF, SS, or ZPK model `sys` at the complex number `f`. For state-space models with data  $(A, B, C, D)$ , the result is

$$H(f) = D + C(fI - A)^{-1}B$$

`evalfr` is a simplified version of `freqresp` meant for quick evaluation of the response at a single point. Use `freqresp` to compute the frequency response over a set of frequencies.

## Examples **Example 1**

To evaluate the discrete-time transfer function

$$H(z) = \frac{z-1}{z^2+z+1}$$

at  $z = 1 + j$ , type

```
H = tf([1 -1],[1 1 1],-1);
z = 1+j;
evalfr(H,z)
```

to get the result:

```
ans =
    2.3077e-01 + 1.5385e-01i
```

## **Example 2**

To evaluate the frequency response of a continuous-time IDTF model at frequency  $w = 0.1$  rad/s, type:

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

# evalfr

---

```
w = 0.1;  
s = 1j*w;  
evalfr(sys, s)
```

The result is same as `freqresp(sys, w)`.

## Limitations

The response is not finite when `f` is a pole of `sys`.

## See Also

`bode` | `freqresp` | `sigma`



<b>Purpose</b>	Value of nonlinearity estimator at given input
<b>Syntax</b>	<code>value = evaluate(nl,x)</code>
<b>Arguments</b>	<p><code>nl</code> Nonlinearity estimator object.</p> <p><code>x</code> Value at which to evaluate the nonlinearity.</p> <p>If <code>nl</code> is a single nonlinearity estimator, then <code>x</code> is a 1-by-<code>nx</code> row vector or an <code>nv</code>-by-<code>nx</code> matrix, where <code>nx</code> is the dimension of the regression vector input to <code>nl</code> (<code>size(nl)</code>) and <code>nv</code> is the number of points where <code>nl</code> is evaluated.</p> <p>If <code>nl</code> is an array of <code>ny</code> nonlinearity estimators, then <code>x</code> is a 1-by-<code>ny</code> cell array of <code>nv</code>-by-<code>nx</code> matrices.</p>
<b>Description</b>	<code>value = evaluate(nl,x)</code> computes the value of a nonlinear estimator object of type <code>customnet</code> , <code>deadzone</code> , <code>linear</code> , <code>neuralnet</code> , <code>pwlinear</code> , <code>saturation</code> , <code>sigmoidnet</code> , <code>treepartition</code> , or <code>wavenet</code> .
<b>Examples</b>	<p>The following syntax evaluates the nonlinearity of an estimated nonlinear ARX model <code>m</code>:</p> <pre>value = evaluate(m.Nonlinearity,x)</pre> <p>where <code>m.Nonlinearity</code> accesses the nonlinearity estimator of the nonlinear ARX model.</p>
<b>See Also</b>	<code>idnlarx</code>   <code>idnlhw</code>

# fcats

---

**Purpose** Concatenate FRD models along frequency dimension

**Syntax** `sys = fcats(sys1,sys2,...)`

**Description** `sys = fcats(sys1,sys2,...)` takes two or more frd models and merges their frequency responses into a single frd model `sys`. The resulting frequency vector is sorted by increasing frequency. The frequency vectors of `sys1`, `sys2`, ... should not intersect. If the frequency vectors do intersect, use `fdel` to remove intersecting data from one or more of the models.

**See Also** `fdel` | `fselect` | `interp` | `frd` | `idfrd`

- Purpose** Delete specified data from frequency response data (FRD) models
- Syntax** `sysout = fdel(sys, freq)`
- Description** `sysout = fdel(sys, freq)` removes from the frd model `sys` the data nearest to the frequency values specified in the vector `freq`.
- Tips**
- Use `fdel` to remove unwanted data (for example, outlier points) at specified frequencies.
  - Use `fdel` to remove data at intersecting frequencies from `frd` models before merging them with `fcats`. `fcats` produces an error when you attempt to merge `frd` models that have intersecting frequency data.
  - To remove data from an `frd` model within a range of frequencies, use `fselect`.
- Input Arguments**
- sys**  
frd model.
- freq**  
Vector of frequency values.
- Output Arguments**
- sysout**  
frd model containing the data remaining in `sys` after removing the frequency points closest to the entries of `freq`.
- Examples**
- Remove selected data from a frd model. In this example, first obtain an frd model:

```
sys = frd(tf([1],[1 1]), logspace(0,1,10))
```

Frequency (rad/s)	Response
-----	-----
1.0000	0.5000 - 0.5000i
1.2915	0.3748 - 0.4841i

1.6681	0.2644 - 0.4410i
2.1544	0.1773 - 0.3819i
2.7826	0.1144 - 0.3183i
3.5938	0.0719 - 0.2583i
4.6416	0.0444 - 0.2059i
5.9948	0.0271 - 0.1623i
7.7426	0.0164 - 0.1270i
10.0000	0.0099 - 0.0990i

Continuous-time frequency response.

The following commands remove the data nearest 2, 3.5, and 6 rad/s from `sys`.

```
freq = [2, 3.5, 6];  
sysout = fdel(sys, freq)
```

Frequency (rad/s)	Response
-----	-----
1.0000	0.5000 - 0.5000i
1.2915	0.3748 - 0.4841i
1.6681	0.2644 - 0.4410i
2.7826	0.1144 - 0.3183i
4.6416	0.0444 - 0.2059i
7.7426	0.0164 - 0.1270i
10.0000	0.0099 - 0.0990i

Continuous-time frequency response.

You do not have to specify the exact frequency of the data to remove. `fdel` removes the data nearest to the specified frequencies.

## See Also

`fcats` | `fselect` | `frd` | `idfrd`

---

<b>Purpose</b>	Identify possible feedback data
<b>Syntax</b>	<code>[fbck,fbck0,nudir] = feedback(Data)</code>
<b>Description</b>	<p>Data is an <code>iddata</code> set with <math>N_y</math> outputs and <math>N_u</math> inputs.</p> <p><code>fbck</code> is an <math>N_y</math>-by-<math>N_u</math> matrix indicating the feedback. The <math>k_y, k_u</math> entry is a measure of feedback from output <math>k_y</math> to input <math>k_u</math>. The value is a probability <math>P</math> in percent. Its interpretation is that if the hypothesis that there is no feedback from output <math>k_y</math> to input <math>k_u</math> were tested at the level <math>P</math>, it would have been rejected. An intuitive but technically incorrect way of thinking about this is to see <math>P</math> as “the probability of feedback.” Often only values above 90% are taken as indications of feedback. When <code>fbck</code> is calculated, direct dependence at lag zero between <math>u(t)</math> and <math>y(t)</math> is not regarded as a feedback effect.</p> <p><code>fbck0</code>: Same as <code>fbck</code>, but direct dependence at lag 0 between <math>u(t)</math> and <math>y(t)</math> is viewed as feedback effect.</p> <p><code>nudir</code>: A vector containing those input numbers that appear to have a direct effect on some outputs, that is, no delay from input to output.</p>
<b>See Also</b>	<code>advice</code>   <code>iddata</code>

# ffplot

---

## Purpose

Compute and plot frequency response magnitude and phase for linear frequencies

---

**Note** `ffplot` will be removed in a future release. Use `bode` or `bodeplot` instead.

---

## Syntax

```
ffplot(m)
ffplot(m,w)
ffplot(m('noise'))
ffplot(m1,...,mN,'sd',sd,'mode','same','ap',ap,'fill')
[mag,phase,w] = ffplot(m)
[mag,phase,w,sdmag,sdphase] = ffplot(m)
```

## Description

`ffplot(m)` plots a frequency response plot for the model `m`, which can be an `idpoly`, `idss`, `idarx`, `idgrey`, or `idfrd` object. This frequency response is a function of linear frequencies in units of inverse time (stored as the `TimeUnit` model property). The default frequency values are determined from the model dynamics. For time series spectra, phase plots are omitted. For MIMO models, press **Enter** to view the next plot in the sequence of different I/O channel pairs, annotated using the `InputNames` and `OuputNames` model properties.

`ffplot(m,w)` plots a frequency response plot at specified frequencies `w` in inverse time units, which can be:

- A vector of values.
- `{wmin,wmax}`, which specifies 100 linearly spaced frequency values ranging from a minimum value `wmin` and a maximum value `wmax`.
- `{wmin,wmax,np}`, which specifies `np` linearly spaced frequency values.

---

**Note** For `idfrd` models, you cannot specify individual frequencies and can only limit the frequencies range for the internally stored frequencies using `{wmin,wmax}`.

---

`ffplot(m('noise'))` plots a frequency response plot of the output noise spectra when the model contains noise spectrum information.

`ffplot(m1,...,mN,'sd',sd,'mode','same','ap',ap,'fill')` plots a frequency response plot for several models. `sd` specifies the confidence region as a positive number that represents the number of standard deviations. The argument `'fill'` indicates that the confidence region is color filled. `mode = 'same'` displays all I/O channels in the same plot. Set `ap = 'A'` to show only amplitude plots, or `ap = 'P'` to show only phase plots.

`[mag,phase,w] = ffplot(m)` computes the magnitude `mag` and phase values of the frequency response, which are 3-D arrays with dimensions (number of outputs)-by-(number of inputs)-by-(length of `w`). `w` specifies the frequency values for computing the response even if you did not specify it as an input. For SISO systems, `mag(1,1,k)` and `phase(1,1,k)` are the magnitude and phase (in degrees) at the frequency `w(k)`. 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)`. When `m` is a time series, `mag` is its power spectrum and `phase` is zero.

`[mag,phase,w,sdmag,sdphase] = ffplot(m)` computes the standard deviations of the magnitude `sdmag` and the phase `sdphase`. `sdmag` is an array of the same size as `mag`, and `sdphase` is an array of the same size as `phase`.

## See Also

`bode` | `etfe` | `freqresp` | `idfrd` | `nyquist` | `spa` | `spafdr`

**Purpose** Transform `iddata` object to frequency domain data

**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` | `spa`



## 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,X0)
[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 2-354.

`[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,X0)` 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 Arguments

- `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.

# findop(idnlarx)

---

- **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`.

## Output Arguments

- **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  $\infty$ -norm of the gradient vector.
  - **SignalLevels**: Structure containing fields `Input` and `Output`, which are the input and output signal levels of the operating point.

## Algorithms

`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  $e(t) = y(t) - f(x(t), u(t))$ , 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

# findop(idnlarx)

---

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 2-354.

## 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)
```

## See Also

`data2state(idnlarx)` | `operspec(idnlarx)` | `sim(idnlarx)`

## 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,X0)
[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 “`idnlhw` States” on page 2-389.

`[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,X0)` 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 Arguments

- `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.

# findop(idnlhw)

---

- **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`.

## Output Arguments

- **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  $\infty$ -norm of the gradient vector.
  - **SignalLevels**: Structure containing fields `Input` and `Output`, which are the input and output signal levels of the operating point.

## Algorithms

`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  $X = \text{inv}(I-A)*B*f(U)$ , 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  $e_1 = y - g(L(x, f(u)))$ , 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 the 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:  $x = \text{inv}(I-A)*B*f(U)$ , where  $[A, B, C, D]$  are state-space parameters corresponding to the linear model  $L(x, u)$ . Thus, substituting  $x = \text{inv}(I-A)*B*f(U)$  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, X0)`, 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  $[A, B, C, D] = \text{ssdata}(\text{model.LinearModel})$ .



---

**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 = n1hw(z2,[4 3 2], 'wavenet', 'pw1');
```

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

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

## See Also

`findstates(idnlhw)` | `operspec(idnlhw)` | `sim(idnlhw)`

# findstates(idParametric)

---

**Purpose** Estimate initial states of identified linear state-space model from data

**Syntax**

```
x0 = findstates(sys,data)
x0 = findstates(sys,data,K)
x0 = findstates(sys,data,K,opt)
```

**Description**

`x0 = findstates(sys,data)` estimates the initial state values of a state-space model, `sys`, to maximize the least squares fit between the measured output data, `data`, and the model response.

`x0 = findstates(sys,data,K)` specifies the prediction horizon, `K`, for computing the response of `sys`.

`x0 = findstates(sys,data,K,opt)` estimates the initial state using the option set, `opt`, to specify initial condition constraints, signal offsets, etc.

## Input Arguments

**sys**  
Identified linear state-space model.  
Specify `sys` as an `idss` or `idgrey` model.

**data**  
Input-output data.  
Specify `data` as an `iddata` object with input/output dimensions that match `sys`.

`data` can be a frequency-domain `iddata` object. Ideally, the frequency vector of `data` should be symmetric about the origin.

If you are converting time-domain data into frequency-domain data, use `fft`. Use the `'comp1'` input argument with `fft` and ensure that there is sufficient zero padding. Note that for an  $N$ -point `fft`, the input/output signals are scaled by  $1/\sqrt{N}$ . Therefore, the estimated `x0` vector is also scaled by this factor. So, scale your data appropriately when you compare `x0` between the time-domain and frequency-domain.

## **K**

Prediction horizon for computing the response of `sys`.

Specify `K` as a positive integer between 1 and `Inf`. The most common values used are:

- `K=1` — Minimizes the 1-step prediction error. That is, the 1-step ahead prediction response of `sys` is compared to the output signals in `data` to determine `x0`.
- `K=Inf` — Minimizes the simulation error. That is, the difference between the measured output, `data.y`, and the simulated response of `sys` to the measured input data, `data.u`.

For continuous-time models, specify `K` as either 1 or `Inf`.

For continuous-time frequency-domain data, specify `K` as `Inf`.

**Default:** 1 (for all data except continuous-time frequency-domain data)

## **opt**

Option set.

`opt` is an options set that allows you to constrain the initial state, remove signal offsets, etc.

Use `findstatesOptions` to create the options set.

## **Output Arguments**

### **x0**

Estimated initial state.

For multi-experiment data, `x0` is a matrix with one column for each experiment.

## **Examples**

### **Estimate Initial States of State-Space Model**

Estimate an `idss` model and simulate it such that the response of the estimated model matches the estimation data's output signal as closely as possible.

## findstates(idParametric)

---

Load sample data.

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

Estimate a linear model from the data.

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

Estimate the value of the initial states to best fit the measured output z1.y.

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

Simulate the model.

```
opt = simOptions('InitialCondition',x0est)  
sim(model,z1.u,opt)
```

### See Also

```
findstatesOptions | idpar | pe | compare | sim | predict  
| forecast | findstates(idnlarx) | findstates(idnlhw) |  
findstates(idnlgrey) | ssest
```

## Purpose

Estimate initial states of nonlinear ARX model from data

## Syntax

```
X0 = findstates(MODEL,DATA)
X0 = findstates(MODEL,DATA,X0INIT)
X0 = findstates(MODEL,DATA,X0INIT,PRED_OR_SIM)
X0 = findstates(MODEL,DATA,X0INIT,PRED_OR_SIM,PVPairs)
[X0, REPORT] = findstates(...)
```

## Description

`X0 = 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 2-354.

`X0 = findstates(MODEL,DATA,X0INIT)` specifies an initial guess for estimating the initial states.

`X0 = findstates(MODEL,DATA,X0INIT,PRED_OR_SIM)` allows switching between prediction-error (default) and simulation-error minimization.

`X0 = findstates(MODEL,DATA,X0INIT,PRED_OR_SIM,PVPairs)` lets you specify the algorithm properties that control the numerical optimization process as property-value pairs.

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

## Input Arguments

- `MODEL`: `idnlarx` model.
- `DATA`: `iddata` object from which to estimate the initial states of `MODEL`.
- `X0INIT`: Initial guess for value of `X0`. 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:

## findstates(idnlarx)

---

- '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 `Display`.

### Output Arguments

- `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  $\infty$ -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')
```

### 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);
```

## findstates(idnlarx)

---

- 2 Estimate the initial states for each experiment in the data set, such that the one-step-ahead prediction error is minimized for each data set.

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

### See Also

```
data2state(idnlarx) | getDelayInfo | findop(idnlarx) |  
findstates(idParametric) | findstates(idnlhw)
```



## 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 2-375.

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

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

## Input Arguments

- 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  $U = [U_1 \ U_2 \ \dots \ U_m]$ , where the  $k^{th}$  column vector is input  $U_k$
  - 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  $T_s$  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.

## findstates(idnlgrey)

---

- '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, X0INIT 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.

### Output Arguments

- X0: Matrix containing the initial states. In the single experiment case it is a column vector of length Nx. For multi-experiment data, X0 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 describing the model structure, the model orders, and model parameters.

```
% Specify the file describing the model structure:
FileName = 'dcmotor_m';
% Specify the model orders [ny nu nx]
Order = [2 1 2];
% Specify the model parameters
% (see idnlgreydemo1 for more information)
Parameters = [0.24365; 0.24964];
```

- 2 Estimate the model parameters 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'});
```

## findstates(idnlgrey)

---

- 5 Estimate the free states of the model.

```
[X0,EstInfo] = findstates(nlgr,z)
```

### Estimating Initial States Starting from States Stored in Model

This example shows how you can estimate all of the initial states, starting from the initial state 0, then from the initial states stored in the model `nlgr`, and finally using a numerical initial states vector as the initial guess.

- 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`.

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

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

```
nlgr.Algorithm.Display = '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, 'Display', 'off');
```

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

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

---

<b>Purpose</b>	Estimate initial states of nonlinear Hammerstein-Wiener model from data
<b>Syntax</b>	<pre>X0 = findstates(MODEL,DATA) X0 = findstates(MODEL,DATA,X0INIT) X0 = findstates(MODEL,DATA,X0INIT,PVPairs) [X0, REPORT] = findstates(...)</pre>
<b>Description</b>	<p><code>X0 = findstates(MODEL,DATA)</code> estimates the initial states of an <code>idnlhw</code> model from given data. The states of an <code>idnlhw</code> model are defined as the states of its embedded linear model (<code>Model.LinearModel</code>). For more information about the states of <code>idnlhw</code> models, see “<code>idnlhw States</code>” on page 2-389.</p> <p><code>X0 = findstates(MODEL,DATA,X0INIT)</code> specifies an initial guess for value of <code>X0</code> using <code>X0INIT</code>.</p> <p><code>X0 = findstates(MODEL,DATA,X0INIT,PVPairs)</code> specifies property-value pairs representing the algorithm properties that control the numerical optimization process.</p> <p><code>[X0, REPORT] = findstates(...)</code> creates a report to summarize results of numerical optimization that is performed to search for the model states.</p>
<b>Input Arguments</b>	<ul style="list-style-type: none"><li>• <code>MODEL</code>: <code>idnlhw</code> model.</li><li>• <code>DATA</code>: <code>iddata</code> object from which to estimate the initial states of <code>MODEL</code>.</li><li>• <code>X0INIT</code>: Initial guess for value of <code>X0</code>. Must be a vector of length equal to the number of the states of <code>MODEL</code>.</li><li>• <code>PVPairs</code>: Property-value pairs that specify the algorithm properties that control numerical optimization process. By default, algorithm properties are read from the <code>Algorithm</code> property of <code>MODEL</code>. You can override <code>MODEL.Algorithm</code> properties using property-value pairs. For example you might set <code>SearchMethod</code>, <code>MaxSize</code>, <code>Tolerance</code>, and <code>Display</code>.</li></ul>

## Output Arguments

- **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  $\infty$ -norm of the gradient vector.

## Examples

In this example, you create an `idnlrx` 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`.

## findstates(idnlhw)

---

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



## Purpose

Option set for findstates

## Syntax

```
opt = findstatesOptions  
opt = findstatesOptions(Name,Value)
```

## Description

`opt = findstatesOptions` creates the default options set for `findstates(idParametric)`.

`opt = findstatesOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### InitialState

Specify how initial states are handled.

`InitialState` takes one of the following:

- 'e' — Estimate initial state such that the prediction error for observed output is minimized.
- 'd' — Similar to 'e', but absorbs nonzero delays into the model coefficients. Use this option for discrete-time models only.
- `x0obj` — Specification object created using `idpar`. Use this object for discrete-time state-space models only and when `K` is 1 or `Inf`. Use `x0obj` to impose constraints on the initial states by fixing their value or specifying minimum/maximum bounds.

**Default:** 'e'

### InputOffset

# findstatesOptions

---

Removes offset from time domain input data during estimation.

Specify as a column vector of length  $Nu$ , where  $Nu$  is the number of inputs.

Use `[]` to indicate no offset.

For multiexperiment data, specify `InputOffset` as a  $Nu$ -by- $Ne$  matrix.  $Nu$  is the number of inputs, and  $Ne$  is the number of experiments.

Each entry specified by `InputOffset` is subtracted from the corresponding input data.

**Default:** `[]`

## **OutputOffset**

Removes offset from time domain output data during estimation.

Specify as a column vector of length  $Ny$ , where  $Ny$  is the number of outputs.

Use `[]` to indicate no offset.

For multiexperiment data, specify `OutputOffset` as a  $Ny$ -by- $Ne$  matrix.  $Ny$  is the number of outputs, and  $Ne$  is the number of experiments. Each entry specified by `OutputOffset` is subtracted from the corresponding output data.

**Default:** `[]`

## **OutputWeight**

Weight of output for initial condition estimation.

`OutputWeight` takes one of the following:

- `'noise'` — Inverse of the noise variance stored with the model.
- `matrix` — A positive semi-definite matrix of dimension  $Ny$ -by- $Ny$ , where  $Ny$  is the number of outputs.

**Default:** `'noise'`

## Output Arguments

**opt**

Option set containing the specified options for `findstates(idParametric)`.

## Examples

### Create Default Options Set for State Identification

Create a default options set for `findstates(idParametric)`.

```
opt = findstatesOptions;
```

### Identify Initial States using Options Set

Create an options set for `findstates(idParametric)` by using an initial state specification object.

Identify a state-space model from data.

```
load iddata8 z8;  
ssest_opt = ssestOptions('Focus','simulation');  
sys = ssest(z8,4,ssest_opt);
```

`z8` is an `iddata` object containing time-domain system response data.

`ssest_opt` specifies the 'Focus' option for state-space estimation as 'simulation'.

`sys` is a fourth-order `idss` model that is identified from the data.

Configure a specification object for the initial states of `sys`.

```
x0obj = x0par([1;nan(3,1)]);  
x0obj.Free(1) = false;  
x0obj.Minimum(2) = 0;  
x0obj.Maximum(2) = 1;
```

`x0obj` specifies estimation constraints on the initial conditions.

The value of the first state is specified as 1 when `x0obj` is created.

`x0obj.Free(1) = false` specifies the first initial state as a fixed estimation parameter.

## findstatesOptions

---

The second state is unknown. But, `x0obj.Minimum(2) = 0` and `x0obj.Maximum(2) = 1` specify the lower and upper bounds of the second state as 0 and 1, respectively.

Create an option set for `findstates` to identify the initial states of `sys`.

```
opt = findstatesOptions('InitialState',x0obj);
```

Alternatively, use dot notation to set the values of `opt`.

```
opt = findstatesOptions;  
opt.InitialState = x0obj;
```

Identify the initial states of `sys`.

```
x0_estimated = findstates(sys,z8,Inf,opt);
```

### See Also

```
findstates(idParametric) | idpar
```

**Purpose** Pointwise peak gain of FRD model

**Syntax** `fnm = fnorm(sys)`  
`fnm = fnorm(sys,ntype)`

**Description** `fnm = fnorm(sys)` computes the pointwise 2-norm of the frequency response contained in the FRD model `sys`, that is, the peak gain at each frequency point. The output `fnm` is an FRD object containing the peak gain across frequencies.

`fnm = fnorm(sys,ntype)` computes the frequency response gains using the matrix norm specified by `ntype`. See `norm` for valid matrix norms and corresponding `NTYPE` values.

**See Also** `norm` | `abs`

# forecast

---

**Purpose** Forecast linear system response into future

**Syntax**

```
yf = forecast(model,past_data,K)
yf = forecast(model,past_data,K,future_inputs)
yf = forecast(model,past_data,K, ___,opt)
[yf,x0efmod] = forecast(model,past_data,K, ___)
```

**Description**

`yf = forecast(model,past_data,K)` forecasts the output of a linear identified time series model, `model`, `K` steps into the future using the historical output data record, `past_data`. Here, “future” denotes the time samples beyond the last sample time in `past_data`.

`yf = forecast(model,past_data,K,future_inputs)` uses the future values of the inputs to `model`, `future_inputs` to forecast the response of an input-output model.

`yf = forecast(model,past_data,K, ___,opt)` forecasts the future output of `model` using the option set `opt` to specify the forecasting algorithm options.

`[yf,x0efmod] = forecast(model,past_data,K, ___)` returns the estimated values for initial states, `x0e`, and a forecasting model, `fmod`.

## Input Arguments

**model**  
Identified linear model.

**past\_data**  
Historical input/output values.

If `model` is a time-series model, which do not have input signals, `past_data` may be specified as an `iddata` object with no inputs. Or, `past_data` may be a matrix of historical time-series data.

**K**  
Time horizon of the forecast.

$K$  must be a positive integer that is a multiple of the sampling time of the data, `past_data.Ts`.

### **future\_inputs**

Future values of inputs to `model`.

Specify `future_inputs` for the time interval `past_data.Tstart + (N+1:N+K)*past_data.Ts`, where  $N$  is the number of samples in `past_data`.

`future_inputs` must be a matrix of size  $[K, Nu]$ , where  $K$  is forecast horizon and  $Nu$  is the number of inputs.

`future_inputs` is only relevant if `model` is not a time-series model.

Alternatively, use an `iddata` model to specify `future_inputs`.

Use `[]` if `model` is a time-series model.

If `past_data` is specified for multiple experiments, then specify `future_inputs` as:

- a multi-experiment `iddata` object.
- cell array of matrices, with an array entry for each corresponding `past_data` experiment data set.

**Default:** 0

### **opt**

Options set for forecast.

Use `forecastOptions` to define options.

## **Output Arguments**

### **yf**

Forecasted response.

`yf` is the forecasted response at times after the last sample time in `past_data`. Let  $N$  be the number of samples in `past_data`. In true

time, `yf` contains data for the time interval `past_data.Tstart + (N+1:N+K)*past_data.Ts`.

## **x0e**

Estimated initial states.

`x0e` is returned only for state-space systems.

## **fmod**

Forecasting model.

`fmod` is a dynamic system whose simulation, using `future_inputs` and `x0e`, yields `yf` as the output.

`fmod` is always a discrete-time system.

If `past_data` is specified for multiple experiments, then `fmod` is an array of dynamic models, with each entry corresponding to an experiment in `past_data`.

## **Examples**

### **Forecast Response of Time Series Model**

Forecast the response of a time series model for a given number of time steps in the future.

Obtain a time series model and past data.

```
load iddata9 z9
model = ar(z9,4);
past_data = z9.OutputData(1:51); % double data
```

`z9` is an `iddata` object that contains the measured output only.

`model` is an `idpoly` time-series model.

`past_data` contains the first 51 data points of `z9`.

Forecast the system response into the future for a given time horizon.

```
K = 100;
```



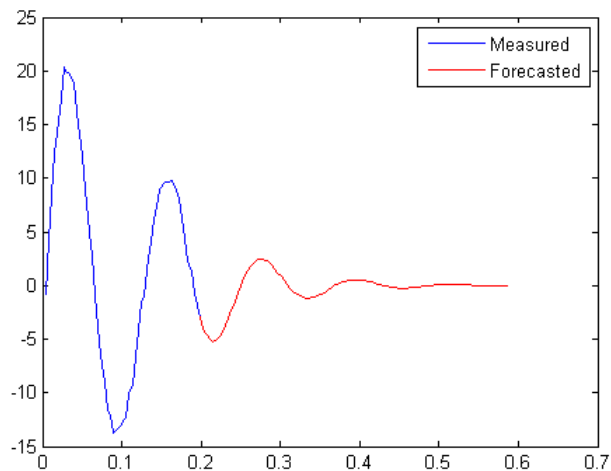
```
yf = forecast(model,past_data(1:50),K);
```

K specifies the forecasting time horizon as 100 samples, with the same sampling time as past\_data.

yf is the forecasted model response.

Analyze the forecasted data.

```
t = z9.SamplingInstants;  
t1 = t(1:51);  
t2 = t(51:150)';  
plot(t1,past_data,t2,yf,'r')  
legend('Measured','Forecasted')
```



### Forecast Model Response Using Future Inputs

Forecast the response of a model using the knowledge of its future inputs.

Obtain past data, future inputs and identified linear model.

# forecast

---

```
load iddata1 z1
z1 = iddata(cumsum(z1.y),cumsum(z1.u),z1.Ts,'InterSample','foh'); % integrate
past_data = z1(1:100);
future_inputs = z1.u(101:end);
model = polyest(z1, [2 2 2 0 0 1],'IntegrateNoise',true);
```

`z1` is an `iddata` object that contains integrated data.

`model` is an `idpoly` model.

`past_data` contains the first 100 data points of `z1`.

`future_inputs` contains the last 200 data points of `z1`.

Forecast the system response into the future for a given time horizon and future inputs.

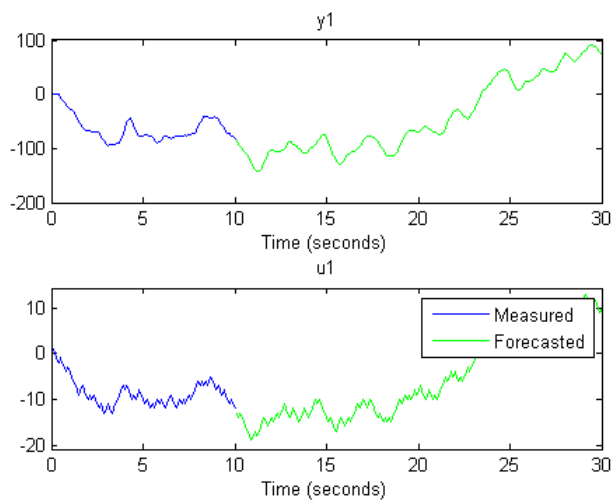
```
K = 200;
yf = forecast(model,past_data,K,future_inputs);
```

`K` specifies the forecasting time horizon as 200 samples, with the same sampling time as `past_data`.

`yf` is the forecasted model response.

Analyze the forecasted data.

```
plot(past_data,yf);
legend('Measured','Forecasted')
```

**See Also**

`forecastOptions` | `predict` | `compare` | `sim` | `lsim` | `ar` | `arx`  
| `n4sid` | `iddata`

# forecastOptions

---

**Purpose** Option set for forecast

**Syntax**  
`opt = forecastOptions`  
`opt = forecastOptions(Name,Value)`

**Description** `opt = forecastOptions` returns the default option set for forecast.  
`opt = forecastOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### InitialCondition

Specify initial conditions.

`InitialCondition` takes one of the following:

- 'z' — Zero initial conditions.
- 'e' — Estimate initial conditions such that the 1-step prediction error is minimized for the observed output.
- 'd' — Similar to 'e', but absorbs nonzero delays into the model coefficients.
- `x0` — Numerical column vector denoting initial states. For multi-experiment data, use a matrix with  $N_e$  columns, where  $N_e$  is the number of experiments. Use this option for state-space models only.
- `x0obj` — Specification object created using `idpar`. Use this object for discrete-time state-space models only. Use `x0obj` to impose constraints on the initial states by fixing their value or specifying minimum/maximum bounds.

The effects of initial conditions on the forecasted response is negligible if the observed data is for a sufficiently long time interval, or if the model has finite memory. For such systems, using zero initial conditions is sufficient. Otherwise, the initial conditions will influence the forecasted values; the influence will usually diminish over the forecasted time interval.

**Default:** 'e'

### **InputOffset**

Input signal offset.

Specify as a column vector of length  $Nu$ , where  $Nu$  is the number of inputs.

Use [] to indicate no offset.

For multi-experiment data, specify **InputOffset** as a  $Nu$ -by- $Ne$  matrix. Here  $Nu$  is the number of inputs and  $Ne$  is the number of experiments.

Each entry specified by **InputOffset** will be subtracted from the corresponding input data before it is used to simulate the model.

**Default:** []

### **OutputOffset**

Output signal offset.

Specify as a column vector of length  $Ny$ , where  $Ny$  is the number of outputs.

Use [] to indicate no offset.

For multiexperiment data, specify **OutputOffset** as a  $Ny$ -by- $Ne$  matrix. Here  $Ny$  is the number of outputs and  $Ne$  is the number of experiments.

Each entry specified by **OutputOffset** will be subtracted from the corresponding output data.

# forecastOptions

---

**Default:** []

## **OutputWeight**

Weight of output for initial condition estimation.

OutputWeight takes one of the following:

- '[]' — No weighting. This is the same as using  $\text{eye}(N_y)$ , where  $N_y$  is the number of outputs.
- 'noise' — Inverse of the noise variance stored with the model.
- matrix — A positive semi-definite matrix of dimension  $N_y$ -by- $N_y$ , where  $N_y$  is the number of outputs.

**Default:** '[]'

## **Output Arguments**

### **opt**

Option set containing the specified options for forecast.

## **Examples**

### **Create Default Options Set for Model Forecasting**

Create a default options set for forecast.

```
opt = forecastOptions;
```

### **Specify Options for Model Forecasting**

Create an options set for forecast using zero initial conditions and set the input offset to 5.

```
opt = forecastOptions('InitialCondition','z','InputOffset',5);
```

Alternatively, use dot notation to set the values of opt.

```
opt = forecastOptions;  
opt.InitialCondition = 'z';  
opt.InputOffset = 5;
```

## **See Also**

forecast | idpar

<b>Purpose</b>	Akaike Final Prediction Error for estimated model
<b>Syntax</b>	<code>fp = fpe(Model1,Model2,Model3,...)</code>
<b>Description</b>	<p><code>Model</code> is the name of an <code>idtf</code>, <code>idgrey</code>, <code>idpoly</code>, <code>idproc</code>, <code>idss</code>, <code>idnlarx</code>, <code>idnlhw</code>, or <code>idnlgrey</code> model object.</p> <p><code>fp</code> is returned as a row vector containing the values of the Akaike Final Prediction Error (FPE) for the different models.</p>
<b>Definitions</b>	<p>Akaike's Final Prediction Error (FPE) criterion provides a measure of model quality by simulating the situation where the model is tested on a different data set. After computing several different models, you can compare them using this criterion. According to Akaike's theory, the most accurate model has the smallest FPE.</p>

---

**Note** If you use the same data set for both model estimation and validation, the fit always improves as you increase the model order and, therefore, the flexibility of the model structure.

---

Akaike's Final Prediction Error (FPE) is defined by the following equation:

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

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

The toolbox assumes that the final prediction error is asymptotic for  $d \ll N$  and uses the following approximation to compute FPE:

$$FPE = V \left( 1 + 2d/N \right)$$

The loss function  $V$  is defined by the following equation:

$$V = \det \left( \frac{1}{N} \sum_1^N \varepsilon(t, \theta_N) (\varepsilon(t, \theta_N))^T \right)$$

where  $\theta_N$  represents the estimated parameters.

**References**

Sections 7.4 and 16.4 in Ljung (1999).

**See Also**

aic | goodnessOfFit



**Purpose** Access data for frequency response data (FRD) object

**Syntax**

```
[response,freq] = frdata(sys)
[response,freq,covresp] = frdata(sys)
[response,freq,Ts,covresp] = frdata(sys,'v')
[response,freq,Ts] = frdata(sys)
```

**Description** `[response,freq] = frdata(sys)` returns the response data and frequency samples of the FRD model `sys`. For an FRD model with  $N_y$  outputs and  $N_u$  inputs at  $N_f$  frequencies:

- `response` is an  $N_y$ -by- $N_u$ -by- $N_f$  multidimensional array where the  $(i,j)$  entry specifies the response from input  $j$  to output  $i$ .
- `freq` is a column vector of length  $N_f$  that contains the frequency samples of the FRD model.

See the `frd` reference page for more information on the data format for FRD response data.

`[response,freq,covresp] = frdata(sys)` also returns the covariance `covresp` of the response data `resp` for `idfrd` model `sys`. The covariance `covresp` is a 5D-array where `covH(i,j,k,:,:) contains the 2-by-2 covariance matrix of the response resp(i,j,k). The  $(1,1)$  element is the variance of the real part, the  $(2,2)$  element the variance of the imaginary part and the  $(1,2)$  and  $(2,1)$  elements the covariance between the real and imaginary parts.`

For SISO FRD models, the syntax

```
[response,freq] = frdata(sys,'v')
```

forces `frdata` to return the response data as a column vector rather than a 3-dimensional array (see example below). Similarly

```
[response,freq,Ts,covresp] = frdata(sys,'v')
```

for an IDFRD model `sys` returns `covresp` as a 3-dimensional rather than a 5-dimensional array.

```
[response,freq,Ts] = frdata(sys)
```

also returns the sample time `Ts`.

Other properties of `sys` can be accessed with `get` or by direct structure-like referencing (e.g., `sys.Frequency`).

## Arguments

The input argument `sys` to `frdata` must be an FRD model.

## Examples

### Extract Data from Frequency Response Data Model

Create a frequency response data model and extract the frequency response data.

Create a frequency response data by computing the response of a transfer function on a grid of frequencies.

```
H = tf([-1.2, -2.4, -1.5], [1, 20, 9.1]);  
w = logspace(-2, 3, 101);  
sys = frd(H, w);
```

`sys` is a SISO frequency response data (`frd`) model containing the frequency response at 101 frequencies.

Extract the frequency response data from `sys`.

```
[response, freq] = frdata(sys);
```

`response` is a 1-by-1-by-101 array. `response(1, 1, k)` is the complex frequency response at the frequency `freq(k)`.

## See Also

`frd` | `get` | `set` | `idfrd` | `freqresp` | `spectrum`

---

<b>Purpose</b>	Frequency response over grid
<b>Syntax</b>	<pre>[H,wout] = freqresp(sys) H = freqresp(sys,w) H = freqresp(sys,w,units) [H,wout,covH] = freqresp(idsys,...)</pre>
<b>Description</b>	<p>[H,wout] = freqresp(sys) returns the frequency response of the dynamic system model <code>sys</code> at frequencies <code>wout</code>. The <code>freqresp</code> command automatically determines the frequencies based on the dynamics of <code>sys</code>.</p> <p>H = freqresp(sys,w) returns the frequency response on the real frequency grid specified by the vector <code>w</code>.</p> <p>H = freqresp(sys,w,units) explicitly specifies the frequency units of <code>w</code> with the string <code>units</code>.</p> <p>[H,wout,covH] = freqresp(idsys,...) also returns the covariance <code>covH</code> of the frequency response of the identified model <code>idsys</code>.</p>
<b>Input Arguments</b>	<p><b>sys</b></p> <p>Any dynamic system model or model array.</p> <p><b>w</b></p> <p>Vector of real frequencies at which to evaluate the frequency response. Specify frequencies in units of <code>rad/TimeUnit</code>, where <code>TimeUnit</code> is the time units specified in the <code>TimeUnit</code> property of <code>sys</code>.</p> <p><b>units</b></p> <p>String specifying the units of the frequencies in the input frequency vector <code>w</code>. Units can take the following values:</p> <ul style="list-style-type: none"><li>• <code>'rad/TimeUnit'</code> — radians per the time unit specified in the <code>TimeUnit</code> property of <code>sys</code></li></ul>

- 'cycles/TimeUnit' — cycles per the time unit specified in the TimeUnit property of `sys`
- 'rad/s'
- 'Hz'
- 'kHz'
- 'MHz'
- 'GHz'
- 'rpm'

**Default:** 'rad/TimeUnit'

## **idsys**

Any identified model.

## **Output Arguments**

### **H**

Array containing the frequency response values.

If `sys` is an individual dynamic system model having `Ny` outputs and `Nu` inputs, `H` is a 3D array with dimensions `Ny`-by-`Nu`-by-`Nw`, where `Nw` is the number of frequency points. Thus, `H(:, :, k)` is the response at the frequency `w(k)` or `wout(k)`.

If `sys` is a model array of size `[Ny Nu S1 ... Sn]`, `H` is an array with dimensions `Ny`-by-`Nu`-by-`Nw`-by-`S1`-by-...-by-`Sn` array.

If `sys` is a frequency response data model (such as `frd`, `genfrd`, or `idfrd`), `freqresp(sys,w)` evaluates to `NaN` for values of `w` falling outside the frequency interval defined by `sys.frequency`. The `freqresp` command can interpolate between frequencies in `sys.frequency`. However, `freqresp` cannot extrapolate beyond the frequency interval defined by `sys.frequency`.

### **wout**

Vector of frequencies corresponding to the frequency response values in `H`. If you omit `w` from the inputs to `freqresp`, the command automatically determines the frequencies of `wout` based on the system dynamics. If you specify `w`, then `wout = w`

### **covH**

Covariance of the response `H`. The covariance is a 5D array where `covH(i, j, k, :, :)` contains the 2-by-2 covariance matrix of the response from the  $i$ th input to the  $j$ th output at frequency  $w(k)$ . The (1,1) element of this 2-by-2 matrix is the variance of the real part of the response. The (2,2) element is the variance of the imaginary part. The (1,2) and (2,1) elements are the covariance between the real and imaginary parts of the response.

## **Definitions**

### **Frequency Response**

In continuous time, the *frequency response* at a frequency  $\omega$  is the transfer function value at  $s = j\omega$ . For state-space models, this value is given by

$$H(j\omega) = D + C(j\omega I - A)^{-1}B$$

In discrete time, the frequency response is the transfer function evaluated at points on the unit circle that correspond to the real frequencies. `freqresp` maps the real frequencies  $w(1), \dots, w(N)$  to points on the unit circle using the transformation  $z = e^{j\omega T_s}$ .  $T_s$  is the sample time. The function returns the values of the transfer function at the resulting  $z$  values. For models with unspecified sample time, `freqresp` uses  $T_s = 1$ .

## **Examples**

### **Frequency Response**

Compute the frequency response of the 2-input, 2-output system

$$\text{sys} = \begin{bmatrix} 0 & \frac{1}{s+1} \\ \frac{s-1}{s+2} & 1 \end{bmatrix}$$

```
sys11 = 0;  
sys22 = 1;  
sys12 = tf(1,[1 1]);  
sys21 = tf([1 -1],[1 2]);  
sys = [sys11,sys12;sys21,sys22];
```

```
[H,wout] = freqresp(sys);
```

H is a 2-by-2-by-45 array. Each entry  $H(:, :, k)$  in H is a 2-by-2 matrix giving the complex frequency response of all input-output pairs of `sys` at the corresponding frequency `wout(k)`. The 45 frequencies in `wout` are automatically selected based on the dynamics of `sys`.

## Response on Specified Frequency Grid

Compute the frequency response of the 2-input, 2-output system

$$\text{sys} = \begin{bmatrix} 0 & \frac{1}{s+1} \\ \frac{s-1}{s+2} & 1 \end{bmatrix}$$

on a logarithmically-spaced grid of 200 frequency points between 10 and 100 radians per second.

```
sys11 = 0;  
sys22 = 1;  
sys12 = tf(1,[1 1]);  
sys21 = tf([1 -1],[1 2]);  
sys = [sys11,sys12;sys21,sys22];
```

```
w = logspace(1,2,200);
```

```
H = freqresp(sys,w);
```

H is a 2-by-2-by-200 array. Each entry  $H(:, :, k)$  in H is a 2-by-2 matrix giving the complex frequency response of all input-output pairs of sys at the corresponding frequency  $w(k)$ .

### Frequency Response and Associated Covariance

Compute the frequency response and associated covariance for an identified model at its peak response frequency.

```
load iddata1 z1
model = procest(z1, 'P2UZ');
w = 4.26;
[H,~,covH] = freqresp(model, w)
```

## Algorithms

For transfer functions or zero-pole-gain models, `freqresp` evaluates the numerator(s) and denominator(s) at the specified frequency points. For continuous-time state-space models ( $A, B, C, D$ ), the frequency response is

$$D + C(j\omega - A)^{-1}B, \quad \omega = \omega_1, \dots, \omega_N$$

For efficiency,  $A$  is reduced to upper Hessenberg form and the linear equation  $(j\omega - A)X = B$  is solved at each frequency point, taking advantage of the Hessenberg structure. The reduction to Hessenberg form provides a good compromise between efficiency and reliability. See [1] for more details on this technique.

## References

[1] Laub, A.J., "Efficient Multivariable Frequency Response Computations," *IEEE Transactions on Automatic Control*, AC-26 (1981), pp. 407-408.

## Alternatives

Use `evalfr` to evaluate the frequency response at individual frequencies or small numbers of frequencies. `freqresp` is optimized for medium-to-large vectors of frequencies.

# freqresp

---

## See Also

`evalfr` | `bode` | `nyquist` | `nichols` | `sigma` | `ltiview` | `interp` | `spectrum`



**Purpose** Select frequency points or range in FRD model

**Syntax** `subsys = fselect(sys, fmin, fmax)`  
`subsys = fselect(sys, index)`

**Description** `subsys = fselect(sys, fmin, fmax)` takes an FRD model `sys` and selects the portion of the frequency response between the frequencies `fmin` and `fmax`. The selected range `[fmin, fmax]` should be expressed in the FRD model units. For an IDFRD model, the `SpectrumData`, `CovarianceData` and `NoiseCovariance` values, if non-empty, are also selected in the chosen range.

`subsys = fselect(sys, index)` selects the frequency points specified by the vector of indices `index`. The resulting frequency grid is

`sys.Frequency(index)`

**See Also** `interp` | `fcats` | `fdel` | `frd` | `idfrd`

**Purpose** Access model property values

**Syntax** Value = get(sys,'PropertyName')  
Struct = get(sys)

**Description** Value = get(sys,'PropertyName') returns the current value of the property `PropertyName` of the model object `sys`. The string `'PropertyName'` can be the full property name (for example, `'UserData'`) or any unambiguous case-insensitive abbreviation (for example, `'user'`). See reference pages for the individual model object types for a list of properties available for that model.

`Struct = get(sys)` converts the TF, SS, or ZPK object `sys` into a standard MATLAB structure with the property names as field names and the property values as field values.

Without left-side argument,

```
get(sys)
```

displays all properties of `sys` and their values.

**Examples** Consider the discrete-time SISO transfer function defined by

```
h = tf(1,[1 2],0.1,'inputname','voltage','user','hello')
```

You can display all properties of `h` with

```
get(h)
    num: {[0 1]}
    den: {[1 2]}
  ioDelay: 0
  Variable: 'z'
    Ts: 0.1
  InputDelay: 0
  OutputDelay: 0
  InputName: {'voltage'}
  OutputName: {''}
```

```
InputGroup: [1x1 struct]
OutputGroup: [1x1 struct]
Name: ''
Notes: {}
UserData: 'hello'
```

or query only about the numerator and sample time values by

```
get(h, 'num')
```

```
ans =
    [1x2 double]
```

```
and
```

```
get(h, 'ts')
```

```
ans =
    0.1000
```

Because the numerator data (num property) is always stored as a cell array, the first command evaluates to a cell array containing the row vector [0 1].

## Tips

An alternative to the syntax

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

is the structure-like referencing

```
Value = sys.PropertyName
```

For example,

```
sys.Ts
sys.a
sys.user
```

## get

---

return the values of the sample time,  $A$  matrix, and UserData property of the (state-space) model `sys`.

### See Also

`frdata` | `set` | `ssdata` | `tfdata` | `idssdata` | `polydata` | `getpvec` | `getcov`

---

<b>Purpose</b>	Parameter covariance information in either raw or factored form
<b>Syntax</b>	<code>cov = getcov(sys,format)</code>
<b>Description</b>	<code>cov = getcov(sys,format)</code> returns covariance data for the parameters of the identified model <code>sys</code> .
<b>Input Arguments</b>	<p><b>sys</b> Identified model. <code>sys</code> may be an array of identified models.</p> <p><b>format</b> Specifies whether <code>cov</code> should be in the raw or factored form. <code>format</code> is a string that takes the following values:</p> <ul style="list-style-type: none"><li>• 'value' — specifies the output as raw parameter covariance.</li><li>• 'factors' — specifies the output as the factored form of the parameter covariance as output. The factored form of the covariance data is useful when the covariance matrix is ill-conditioned.</li></ul> <p style="text-align: center;"><b>Default:</b> 'value'</p>
<b>Output Arguments</b>	<p><b>cov</b> Parameter covariance information for identified model. If <code>format</code> is</p> <ul style="list-style-type: none"><li>• 'value' — <code>COV</code> is an <math>np</math>-by-<math>np</math> matrix of doubles, where <math>np</math> is the number of parameters of <code>sys</code>.</li></ul> <p>If a parameter is fixed, then its corresponding row and column entries contain zeros.</p> <p>For an estimated model, <code>COV</code> is equal to <code>sys.Report.Parameters.ParameterCovariance</code>.</p>

If `sys` is a model array, then `COV` is a cell array of corresponding size. Each cell of `COV` contains the parameter covariance information for the corresponding model.

- 'factors' — `COV` is a structure that stores the covariance data in a “factored” form using the following fields:
  - `R` — usually the Cholesky factor of inverse of covariance.
  - `T` — transformation matrix.
  - `Free` — logical vector of length  $np$  indicating if a parameter is free. Here  $np$  is equal to the number of parameters of `sys`.

To calculate the covariance:

```
cov = getcov(sys, 'factors');
Free = cov.Free;
T = cov.T;
R = cov.R;
np = nparams(sys);
covariance_matrix = zeros(np);
covariance_matrix(Free, Free) = T*inv(R'*R)*T';
```

Use this option when the covariance matrix is ill-conditioned; for example, when  $\text{rank}(R) < \text{size}(R)$ . You can often compute the response uncertainties, such as confidence bounds on the step or frequency response, by using the factors directly, without explicitly forming the covariance matrix.

There is no numerical advantage to fetching the covariance information in the factored form for the following cases:

- `sys` is estimated using certain instrument variable methods
- the parameter covariance for `sys` is explicitly set using the deprecated `CovarianceMatrix` model property

If `sys` is a model array, then `COV` is a structure array of corresponding size. Each entry of `COV` contains the factored parameter covariance information for the corresponding model.

**Examples****Raw Covariance**

Obtain raw covariance for an identified model.

```
load iddata1 z1;  
sys = tfest(z1,2);  
cov_raw = getcov(sys);
```

`getcov(sys)` uses the default value for the `format` argument, `value`, which returns the raw covariance data.

**Factored Covariance**

Obtain covariance data for an identified model in factored form.

```
load iddata z1;  
sys = tfest(z1,2);  
cov_factored = getcov(sys, 'factors');
```

**See Also**

[setcov](#) | [nparams](#) | [rsample](#) | [sim](#) | [simstd](#) | [getpar](#)

# getDelayInfo

---

**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:

$p(t-1), p(t-2), \dots, p(t-D)$

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 2-354

**Input Arguments** 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  $[n_y, n_u] = \text{size}(\text{MODEL})$ ).
- 'channelwise': DELAYS contains delay values separated for each output ( $n_y$ -by- $(n_y+n_u)$  matrix).

## Output Arguments

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

## Examples

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

**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.

## getDelayInfo

---

- 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 ( $u_5(t-1), \dots, u_5(t-5)$ ). Applying this definition to all I/O channels, the complete state vector for model `M` becomes:

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

---

### See Also

`data2state(idnlarx)` | `getreg` | `idnlarx`

**Purpose**

Specific experiments from multiple-experiment data set

**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

---

<b>Purpose</b>	Values of idnlgrey model initial states
<b>Syntax</b>	<code>getinit(model)</code> <code>getinit(model,prop)</code>
<b>Arguments</b>	<code>model</code> Name of the idnlgrey model object.  <code>Property</code> Name of the InitialStates model property field, such as 'Name', 'Unit', 'Value', 'Minimum', 'Maximum', and 'Fixed'.  Default: 'Value'.
<b>Description</b>	<code>getinit(model)</code> gets the initial-state values in the 'Value' field of the InitialStates model property.  <code>getinit(model,prop)</code> 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.
<b>See Also</b>	<code>getpar</code>   <code>idnlgrey</code>   <code>setinit</code>   <code>setpar</code>

**Purpose** Return @PlotOptions handle or plot options property

**Syntax** `p = getoptions(h)`  
`p = getoptions(h,propertyname)`

**Description** `p = getoptions(h)` returns the plot options handle associated with plot handle `h`. `p` contains all the settable options for a given response plot.

`p = getoptions(h,propertyname)` returns the specified options property, `propertyname`, for the plot with handle `h`. You can use this to interrogate a plot handle. For example,

```
p = getoptions(h, 'Grid')
```

returns 'on' if a grid is visible, and 'off' when it is not.

For a list of the properties and values available for each plot type, see “Properties and Values Reference”.

**See Also** `setoptions`

# getpar

---

**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', 'Maximum', or 'Fixed'.  
  
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  $N_p$ -by-1 cell array of values, where  $N_p$  is the number of parameters.

**See Also** `getinit` | `idnlgrey` | `setinit` | `setpar` | `getpvec`

<b>Purpose</b>	Model parameters and associated uncertainty data
<b>Syntax</b>	<pre>pvec = getpvec(sys) [pvec,pvec_sd] = getpvec(sys) [pvec,pvec_sd] = getpvec(sys,'free')</pre>
<b>Description</b>	<p><code>pvec = getpvec(sys)</code> returns a vector, <code>pvec</code>, containing the values of all the parameters of the identified model <code>sys</code>.</p> <p><code>[pvec,pvec_sd] = getpvec(sys)</code> also returns the 1 standard deviation value of the uncertainty associated with the parameters of <code>sys</code>. If the model covariance information for <code>sys</code> is not available, <code>pvec_sd</code> is <code>[]</code>.</p> <p><code>[pvec,pvec_sd] = getpvec(sys,'free')</code> returns the values and standard deviation data for only the free parameters of <code>sys</code>.</p>
<b>Input Arguments</b>	<p><b>sys</b> Identified model.</p>
<b>Output Arguments</b>	<p><b>pvec</b> Values of the parameters of <code>sys</code>.</p> <p>If <code>sys</code> is an array of models, then <code>pvec</code> is a cell array with parameter value vectors corresponding to each model in <code>sys</code>.</p> <p><b>pvec_sd</b> 1 standard deviation value of the parameters of <code>sys</code>.</p> <p>If the model covariance information for <code>sys</code> is not available, <code>pvec_sd</code> is <code>[]</code>.</p> <p>If <code>sys</code> is an array of models, then <code>pvec_sd</code> is a cell array with standard deviation vectors corresponding to each model in <code>sys</code>.</p>

## Examples

Obtain the parameter values for an estimated transfer function.

```
load iddata1 z1;  
sys = tfest(z1,3);  
pvec = getpvec(sys);
```

---

Obtain the parameter values and associated 1 standard deviation values for an estimated state-space model.

```
load iddata2 z2;  
sys = ssest(z2,3);  
[pvec, pvec_sd]=getpvec(sys)
```

---

Obtain the free parameter values and associated 1 standard deviation values for an estimated state-space model.

```
load iddata2 z2;  
sys = ssest(z2,3);  
[pvec, pvec_sd]=getpvec(sys,'free')
```

## See Also

[setpvec](#) | [getcov](#) | [idssdata](#) | [tfdata](#) | [zpkdata](#)



**Purpose**

Regressor expressions and numerical values in nonlinear ARX model

**Syntax**

```
Rs = getreg(model)
Rs = getreg(model,subset)
Rm = getreg(model,subset,data)
Rm = getreg(model,subset,data,init)
```

**Description**

`Rs = getreg(model)` returns expressions for computing regressors in the nonlinear ARX model. `Rs` is a cell array of strings. `model` is an `idnlarx` object.

`Rs = getreg(model,subset)` returns regressor expressions for a specified subset of regressors. `subset` is a string.

`Rm = getreg(model,subset,data)` returns regressor values as a matrix for a specified subset of regressors.

`Rm = getreg(model,subset,data,init)` returns regressor values as matrices for a specified subset of regressors. The first `N` rows of each regressor matrix depend on the initial states `init`, where `N` is the maximum delay in the regressors (see `getDelayInfo`). For multiple-output models, `Rm` is a cell array of cell arrays.

**Input Arguments**

`data`

`iddata` object containing measured data.

`init`

Initial conditions of your data:

- 'z' (default) specifies zero initial state.
- Real column vector containing the initial state values. `input` and output data values at a time instant before the first sample in `data`. To create the initial state vector from the input-output data, use the `data2state` method of the `idnlarx` class. For multiple-experiment data, this is a matrix where each column specifies the initial state of the model corresponding to that experiment.

- `iddata` object containing input and output samples at time instants before to the first sample in `data`. When the `iddata` object contains more samples than the maximum delay in the model, only the most recent samples are used. The minimum number of samples required is equal to `max(getDelayInfo(model))`.

`model`

`iddata` object representing nonlinear ARX model.

`subset`

String that represents a subset of all regressors:

- (Default) `'all'` — All regressors.
- `'custom'` — Only custom regressors.
- `'input'` — Only standard regressors computed from input data.
- `'linear'` — Only regressors not used in the nonlinear block.
- `'nonlinear'` — Only regressors used in the nonlinear block.

---

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

---

- `'output'` — Only regressors computed from output data.
- `'standard'` — Only standard regressors (excluding any custom regressors).

## Output Arguments

`Rm`

Matrix of regressor values for all or a specified subset of regressors. Each matrix in `Rm` contains as many rows as there are data samples. For a model with `ny` outputs, `Rm` is an `ny`-by-1 cell array of matrices. When `data` contains multiple experiments, `Rm` is a cell array where each element corresponds to a matrix of regressor values for an experiment.

Rs

Regressor expressions represented as a cell array of strings. For a model with  $n_y$  outputs, Rs is an  $n_y$ -by-1 cell array of cell arrays of strings. For example, the expression 'u1(t-2)' computes the regressor by delaying the input signal u1 by two time samples. Similarly, the expression 'y2(t-1)' computes the regressor by delaying the output signal y2 by one time sample.

The order of regressors in Rs corresponds to regressor indices in the `idnlarx` object property `model.NonlinearRegressors`.

## Examples

Get regressor expressions and values, and evaluate the predicted model output:

```
% Load sample data u and y:
load twotankdata;
Ts = 0.2; % Sampling interval is 0.2 min
% Create data object:
z = iddata(y,u,Ts);
% Use first 1000 samples for estimation:
ze = z(1:1000);
% Estimate nonlinear ARX model
model = nlarx(ze,[3 2 1]);
% Get regressor expressions:
Rs = getreg(model)
% Get regressor values:
Rm = getreg(model,'all',ze)
% Evaluate model output for one-step-prediction:
Y = evaluate(model.Nonlinearity,Rm)
% The previous result is equivalent to:
Y_p = predict(model,ze,1,'z')
```

## See Also

[addreg](#) | [customreg](#) | [evaluate](#) | [polyreg](#)

## How To

- “Identifying Nonlinear ARX Models”

# getTrend

---

**Purpose** Data offset and trend information

**Syntax**

```
T = getTrend(data)
T = getTrend(data,0)
T = getTrend(data,1)
```

**Description** `T = getTrend(data)` constructs a `TrendInfo` object to store offset, mean, or linear trend information for detrending or retrending data. You can assign specific offset and slope values to `T`.

`T = getTrend(data,0)` computes the means of input and output signals and stores them as `InputOffset` and `OutputOffset` properties of `T`, respectively.

`T = getTrend(data,1)` computes a best-fit straight line for both input and output signals and stores them as properties of `T`.

**Examples** Compute input-output signal means, store them, and detrend the data:

```
% Load SISO data containing vectors u2 and y2
load dryer2
% Create data object with sampling time of 0.08 sec
data=iddata(y2,u2,0.08)
% Plot data on a time plot - it has a nonzero mean
plot(data)
% Compute the mean of the data
T = getTrend(data,0)
% Remove the mean from the data
data_d = detrend(data,T)
% Plot detrended data on the same plot
hold on
plot(data_d)
```

Remove a specific offset from input and output data signals:

```
% Load SISO data containing vectors u2 and y2
load dryer2
% Create data object with sampling time of 0.08 sec
```

```
data=iddata(y2,u2,0.08)
plot(data)
% Create a TrendInfo object for storing offsets and trends
T = getTrend(data)
% Assign offset values to the TrendInfo object
T.InputOffset=5;
T.OutputOffset=5;
% Subtract specific offset from the data
data_d = detrend(data,T)
% Plot detrended data on the same plot
hold on
plot(data_d)
```

## See Also

detrend | retrend | TrendInfo

## How To

- “Handling Offsets and Trends in Data”

# goodnessOfFit

---

**Purpose** Goodness of fit between test and reference data

**Syntax** `fit = goodnessOfFit(x,xref,cost_func)`

**Description** `fit = goodnessOfFit(x,xref,cost_func)` returns the goodness of fit between the data matrix, `x`, and the reference, `xref` using a cost function specified by `cost_func`.

**Input Arguments**

**x**

Test data.

`x` is an `Ns`-by-`N` matrix, where `Ns` is the number of samples and `N` is the number of channels.

`x` can also be a cell array of multiple test data sets.

`x` must not contain any NaN or Inf values.

**xref**

Reference data.

`xref` must be of the same size as `x`.

`xref` can also be a cell array of multiple reference sets. In this case, each individual reference set should be of the same size as the corresponding test data set.

`xref` must not contain any NaN or Inf values.

**cost\_func**

Cost function to determine goodness of fit.

`cost_func` takes the following strings:

- 'MSE' — Mean square error.

$$fit = \frac{|x - xref|^2}{Ns - 1}$$

`fit` is a scalar value.

- 'NRMSE' — Normalized root mean square error.

$$fit(i) = 1 - \frac{|x(:,i) - xref(:,i)|}{|x(:,i) - mean(xref(:,i))|}$$

`fit` is a row vector of length `N` and `i = 1, ..., N`, where `N` is the number of channels.

NRMSE costs vary between `-Inf` (bad fit) to `1` (perfect fit). If the cost function is equal to zero, then `x` is no better than a straight line at matching `xref`.

- 'NMSE' — Normalized mean square error.

$$fit(i) = 1 - \frac{|x(:,i) - xref(:,i)|}{|x(:,i) - mean(xref(:,i))|^2}$$

`fit` is a row vector of length `N` and `i = 1, ..., N`, where `N` is the number of channels.

NMSE costs vary between `-Inf` (bad fit) to `1` (perfect fit). If the cost function is equal to zero, then `x` is no better than a straight line at matching `xref`.

## Output Arguments

### `fit`

Goodness of fit between test and reference data.

For a single test data set and reference pair, `fit` is a :

- scalar if `cost_func` is MSE.
- row vector of length `N`, where `N` is the number of channels, if `cost_func` is NRMSE or NMSE.

If `x` and/or `xref` are cell arrays, then `fit` is an array containing the goodness of fit values for each test data and reference pair.

# goodnessOfFit

---

## Examples

Calculate the goodness of fit between estimated output and measured output using NRMSE measure.

```
load iddata1 z1
sys = tfest(z1, 2);
y = sim(sys, z1.u);
yref = z1.y;
Fit = goodnessOfFit(y, yref, 'nrmse')
```

% Note: compare function shows the same number (in percentage); try compa

## See Also

[compare](#) | [pe](#) | [resid](#) | [fpe](#) | [aic](#)



<b>Purpose</b>	Linear grey box model estimation
<b>Syntax</b>	<pre>sys = greyest(data,init_sys) sys = greyest(data,init_sys,opt)</pre>
<b>Description</b>	<p><code>sys = greyest(data,init_sys)</code> estimates a linear grey box model, <code>sys</code>, using time or frequency domain data, <code>data</code>. The dimensions of the inputs and outputs of <code>data</code> and <code>init_sys</code>, an <code>idgrey</code> model, must match. <code>sys</code> is an identified <code>idgrey</code> model that has the same structure as <code>init_sys</code>.</p> <p><code>sys = greyest(data,init_sys,opt)</code> estimates a linear grey box model using the option set, <code>opt</code>, to configure the estimation options.</p>
<b>Input Arguments</b>	<p><b>data</b></p> <p>Estimation data.</p> <p>The dimensions of the inputs and outputs of <code>data</code> and <code>init_sys</code> must match.</p> <p>For time domain estimation, <code>data</code> is an <code>iddata</code> object containing the input and output signal values.</p> <p>For frequency domain estimation, <code>data</code> can be one of the following:</p> <ul style="list-style-type: none"><li>• recorded frequency response data (<code>frd</code> or <code>idfrd</code>)</li><li>• <code>iddata</code> object with its <code>Domain</code> property set to 'Frequency'</li></ul> <p><b>init_sys</b></p> <p>Identified linear grey box model that configures the initial parameterization of <code>sys</code>.</p> <p><code>init_sys</code>, an <code>idgrey</code> model, must have the same input and output dimensions as <code>data</code>.</p> <p><b>opt</b></p> <p>Estimation options.</p>

opt is an option set that specifies the following:

- estimation objective
- initialization choice
- disturbance model handling
- numerical search method to be used in estimation

Use `greyestOptions` to create the option set.

## Output Arguments

### **sys**

Estimated linear grey box model.

`sys` is an `idgrey` model which encapsulates the estimated linear grey box model.

## Examples

### **Estimate Grey-Box Model**

Estimate the parameters of a DC motor using the linear grey-box framework.

Load the measured data.

```
load(fullfile(matlabroot, 'toolbox', 'ident', ...
    'iddemos', 'data', 'dcmotordata'));
data = iddata(y, u, 0.1, 'Name', 'DC-motor');
set(data, 'InputName', 'Voltage', 'InputUnit', 'V');
set(data, 'OutputName', {'Angular position', 'Angular velocity'});
set(data, 'OutputUnit', {'rad', 'rad/s'});
set(data, 'Tstart', 0, 'TimeUnit', 's');
```

`data` is an `iddata` object containing the measured data for the outputs, the angular position and the angular velocity, and the input, the driving voltage.

Create a grey-box model representing the dynamics of the DC motor.

For this example, choose the angular position (rad) and the angular velocity (rad/s) of the motor as its outputs and the driving voltage (V) as the input. Set up a linear state-space structure of the following form:

$$\dot{x}(t) = \begin{bmatrix} 0 & 1 \\ 0 & -\frac{1}{\tau} \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ \frac{G}{\tau} \end{bmatrix} u(t)$$

$$y(t) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x(t)$$

$\tau$  is the time-constant of the motor in seconds and  $G$  is the static gain from the input to the angular velocity in rad/(V\*s) .

```
G = 0.25;
tau = 1;
```

```
init_sys = idgrey('motor',tau,'cd',G,0);
```

The governing equations in state-space form are represented in the MATLAB file `motor.m`. To view the contents of this file, enter `edit motor.m` at the MATLAB command prompt.

In this example, we treat  $G$  as a known quantity that is provided to `motor.m` as an optional argument.  $\tau$  is treated as a free estimation parameter.

`init_sys` is an `idgrey` model associated with `motor.m`.

Estimate  $\tau$ .

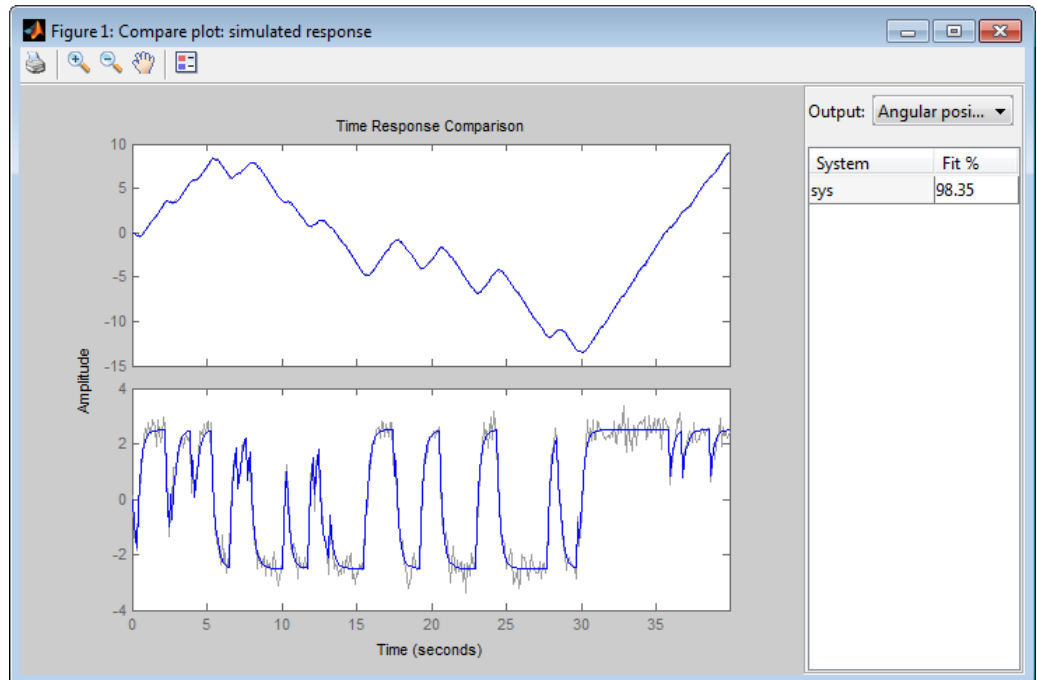
```
sys = greyest(data,init_sys);
```

`sys` is an `idgrey` model containing the estimated value of  $\tau$ .

To obtain the estimated parameter values associated with `sys`, use `getpvec(sys)`.

Analyze the result.

```
opt = compareOptions('InitialCondition','zero');  
compare(data,sys,Inf,opt)
```



sys provides a 98.35% fit for the angular position and an 84.42% fit for the angular velocity.

## See Also

[idgrey](#) | [greyestOptions](#) | [iddata](#) | [idfrd](#) | [srest](#) | [idnlgrey](#)  
| [pem](#)

## Related Examples

- “Estimate Model Using Zero/Pole/Gain Parameters”

**Purpose** Option set for greyst

**Syntax**  
opt = greystOptions  
opt = greystOptions(Name,Value)

**Description** opt = greystOptions creates the default options set for greyst.  
opt = greystOptions(Name,Value) creates an option set with the options specified by one or more Name,Value pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments, where Name is the argument name and Value is the corresponding value. Name must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

### InitialState

Specify how initial states are handled during estimation.

InitialState takes one of the following strings:

- 'model' — The initial state is parameterized by the ODE file used by the idgrey model. The ODE file must return 6 or more output arguments.
- 'zero' — The initial state is set to zero. Any values returned by the ODE file are ignored.
- 'estimate' — The initial state is treated as an independent estimation parameter.
- 'backcast' — The initial state is estimated using the best least squares fit.
- 'auto' — The software chooses the method to handle initial states based on the estimation data.

- vector of doubles — Specify a column vector of length  $Nx$ , where  $Nx$  is the number of states. For multi-experiment data, specify a matrix with  $Ne$  columns, where  $Ne$  is the number of experiments. The specified values are treated as fixed values during the estimation process.

**Default:** 'auto'

## **DisturbanceModel**

Specify how the disturbance component ( $K$ ) is handled during estimation.

`DisturbanceModel` takes one of the following strings:

- 'model' —  $K$  values are parameterized by the ODE file used by the `idgrey` model. The ODE file must return 5 or more output arguments.
- 'fixed' — The value of the `k` property of the `idgrey` model is considered to be fixed to its original value.
- 'none' —  $K$  is fixed to zero. Any values returned by the ODE file are ignored.
- 'estimate' —  $K$  is treated as an independent estimation parameter.
- 'auto' — The software chooses the method to handle how the disturbance component is handled during estimation. The software uses the 'model' method if the ODE file returns 5 or more output arguments with a finite value for  $K$ . Else, the software uses the 'fixed' method.

---

**Note** Noise model cannot be estimated using frequency domain data.

---

**Default:** 'auto'

## **Focus**

Defines how the errors  $e$  between the measured and the modeled outputs are weighed at specific frequencies during the minimization of the prediction error.

Higher weighting at specific frequencies emphasizes the requirement for a good fit at these frequencies.

Focus can take the following values:

- '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.

This method provides a stable model.

- 'prediction' — Same as 'simulation', except that it does not enforce the stability of the resulting model.
- 'stability' — Same as 'prediction' but with model stability enforced.
- Passbands — Row vector or matrix containing frequency values that define desired passbands. For example:

```
[w1,wh]
[w11,w1h;w21,w2h;w31,w3h;...]
```

where  $w1$  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.

- SISO filter — Enter any SISO linear filter in any of the following ways:
  - A single-input-single-output (SISO) linear system.
  - The {A,B,C,D} format, which specifies the state-space matrices of the filter.
  - The {numerator, denominator} format, 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. To obtain a good model fit for a specific frequency range, you must choose the filter with a passband in this range. The estimation result is the same if you first prefilter the data using `idfilt`.

- **Weighting vector** — 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.

**Default:** 'prediction'

## **EstCovar**

Controls whether parameter covariance data is generated or not.

If `EstCovar` is true, then use `getcov` to fetch the covariance matrix from the estimated model.

**Default:** true

## **Display**

Specifies whether estimation progress should be displayed.

`Display` requires one of the following strings:

- 'on' — Information on model structure and estimation results are displayed in a progress viewer window
- 'off' — No progress or results information is displayed

**Default:** 'off'

## **InputOffset**

Removes offset from time domain input data during estimation.



Specify as a column vector of length  $Nu$ , where  $Nu$  is the number of inputs.

Use [] to indicate no offset.

For multiexperiment data, specify `InputOffset` as a  $Nu$ -by- $Ne$  matrix.  $Nu$  is the number of inputs, and  $Ne$  is the number of experiments.

Each entry specified by `InputOffset` is subtracted from the corresponding input data.

**Default:** []

### **OutputOffset**

Removes offset from time domain output data during estimation.

Specify as a column vector of length  $Ny$ , where  $Ny$  is the number of outputs.

Use [] to indicate no offset.

For multiexperiment data, specify `OutputOffset` as a  $Ny$ -by- $Ne$  matrix.  $Ny$  is the number of outputs, and  $Ne$  is the number of experiments. Each entry specified by `OutputOffset` is subtracted from the corresponding output data.

**Default:** []

### **OutputWeight**

Specifies criterion used during minimization.

`OutputWeight` can have the following values:

- 'noise' — 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.

- positive semi-definite symmetric matrix ( $W$ ) — Minimize the trace of the weighted prediction error matrix  $\text{trace}(E' * E * W)$ , where  $E$  is the matrix of prediction errors, with one column for each output, and  $W$  is the positive semi-definite symmetric matrix of size equal to the number of outputs. Use  $W$  to specify the relative importance of outputs in multiple-input multiple-output models, or the reliability of corresponding data.

This option is relevant only for multi-input, multi-output models.

- `[]` — The software chooses between the 'noise' or using the identity matrix for  $W$ .

**Default:** `[]`

## SearchMethod

Search method used for iterative parameter estimation.

SearchMethod is a string that can take the following values:

- `gn` — The subspace Gauss-Newton direction.
- `gna` — An adaptive version of subspace Gauss-Newton approach, suggested by Wills and Ninness [1].
- `lm` — Uses the Levenberg-Marquardt method.
- `lsqnonlin` — Uses the trust region reflective algorithm. Requires Optimization Toolbox software.
- `grad` — The steepest descent gradient search method.
- `auto` — A choice among the above is made in the algorithm. The descent direction is calculated using `gn`, `gna`, `lm` and `grad` successively, in that order, at each iteration until a sufficient reduction in error is achieved.

**Default:** `'auto'`

## SearchOption

SearchOption is an options set for the search algorithm with the fields:

SearchMethod	SearchOption
gn,gna,lm,grad and auto	<ul style="list-style-type: none"> <li>• <b>Tolerance</b> — 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 <b>Tolerance</b>, the iterations are stopped. 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.  <b>Default:</b> 0.01</li> <li>• <b>MaxIter</b> — Maximum number of iterations during loss-function minimization. The iterations stop when <b>MaxIter</b> is reached or another stopping criterion is satisfied, such as <b>Tolerance</b>.  Setting <b>MaxIter</b> = 0 returns the result of the startup procedure.  Use <code>sys.Report.Termination.Iterations</code> to get the actual number of iterations during an estimation, where <code>sys</code> is an <code>idtf</code> model.  <b>Default:</b> 20</li> <li>• <b>Advanced</b> — Search settings:             <ul style="list-style-type: none"> <li>▪ <b>GnPinvConst</b> — Singular values of the Jacobian that are smaller than <math>\text{GnPinvConst} \cdot \max(\text{size}(J) \cdot \text{norm}(J)) \cdot \text{eps}</math> are discarded when computing the search direction and <b>SearchMethod</b> is 'gn'.</li> </ul> <p><b>GnPinvConst</b> must be a positive real value.  <b>Default:</b> 10000</p> </li> </ul>

SearchMethod	SearchOption
	<ul style="list-style-type: none"><li data-bbox="583 361 1267 465">▪ <b>InitGnaTol</b> — Initial value of <i>gamma</i> when SearchMethod is 'gna'. <b>Default:</b> .0001</li><li data-bbox="583 491 1267 630">▪ <b>LMStartValue</b> — Starting value of search-direction length <i>d</i> in the Levenberg-Marquardt method. Applicable when SearchMethod is 'lm'. <b>Default:</b> .001</li><li data-bbox="583 656 1267 855">▪ <b>LMStep</b> — Size of the Levenberg-Marquardt step. The next value of the search-direction length <i>d</i> in the Levenberg-Marquardt method is LMStep times the previous one. Applicable when SearchMethod is 'lm'. <b>Default:</b> 2</li><li data-bbox="583 881 1267 1064">▪ <b>MaxBisections</b> — Maximum number of bisections used by the line search along the search direction. <b>Default:</b> 25</li><li data-bbox="583 1090 1267 1150">▪ <b>MaxFunEvals</b> — Iterations are stopped if the number of calls to the model file exceeds this value. MaxFunEvals must be a positive integer value. <b>Default:</b> Inf</li><li data-bbox="583 1177 1267 1289">▪ <b>MinParChange</b> — Smallest parameter update allowed per iteration MinParChange must be a positive, real value. <b>Default:</b> 0</li><li data-bbox="583 1315 1267 1446">▪ <b>RelImprovement</b> — Iterations are stopped if the relative improvement of the criterion function is less than RelImprovement.</li></ul>

SearchMethod	SearchOption
	<p>RelImprovement must be a positive integer value.</p> <p><b>Default:</b> 0</p> <ul style="list-style-type: none"> <li>▪ StepReduction — 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.</li> </ul> <p>StepReduction must be a positive, real value that is greater than 1.</p> <p><b>Default:</b> 2</p>
lsqnonlin	<ul style="list-style-type: none"> <li>• TolFun — Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values.</li> </ul> <p>The value of TolFun is the same as that of <code>sys.SearchOption.Advanced.TolFun</code>.</p> <p><b>Default:</b> 1e-5</p> <ul style="list-style-type: none"> <li>• TolX — Termination tolerance on the estimated parameter values.</li> </ul> <p>The value of TolX is the same as that of <code>sys.SearchOption.Advanced.TolX</code>.</p> <p><b>Default:</b> 1e-6</p> <ul style="list-style-type: none"> <li>• MaxIter — Maximum number of iterations during loss-function minimization. The iterations stop when MaxIter is reached or another stopping criterion is satisfied, such as TolFun etc..</li> </ul> <p>The value of MaxIter is the same as that of <code>sys.SearchOption.Advanced.MaxIter</code>.</p> <p><b>Default:</b> 20</p> <ul style="list-style-type: none"> <li>• Advanced — Options set for lsqnonlin.</li> </ul>

SearchMethod	SearchOption
	<p>For more information, see “Optimization Options”.</p> <p>Use <code>optimset('lsqnonlin')</code> to create an options set for <code>lsqnonlin</code> and then modify it to specify its various options.</p>

## Advanced

Advanced is a structure with the following fields:

- **ErrorThreshold** — Specifies when to adjust the weight of large errors from quadratic to linear.

Errors larger than `ErrorThreshold` times the estimated standard deviation have a linear weight in the 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 the chapter “Computing the Estimate” of [2].) `ErrorThreshold = 0` disables robustification and leads to a purely quadratic criterion. When estimating with frequency-domain data, `ErrorThreshold` is set to zero.

**Default:** 0

- **MaxSize** — Specifies the maximum number of elements in a segment when input-output data is split into segments.

`MaxSize` must be a positive integer.

**Default:** 250000

- **StabilityThreshold** — Specifies thresholds for stability tests.

`StabilityThreshold` is a structure with the following fields:

- **s** — Specifies the location of the right-most pole to test the stability of continuous-time models. A model is considered stable when its right-most pole is to the left of `s`.

**Default:** 0

- `z` — 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 `z` from the origin.

**Default:** `1+sqrt(eps)`

- `AutoInitThreshold` — Specifies when to automatically estimate the initial conditions.

When `InitialCondition = 'Auto'`, the initial condition is estimated when the ratio of the prediction-error norm with a zero initial condition to the norm with an estimated initial condition exceeds `AutoInitialState`.

**Default:** `1.05`

## Output Arguments

### `opt`

Option set containing the specified options for `greyest`.

## Examples

### Create Default Options Set for Linear Grey Box Estimation

Create a default options set for `greyest`.

```
opt = greyestOptions;
```

### Specify Options for Linear Grey Box Estimation

Create an options set for `greyest` using the `'backcast'` algorithm to initialize the state and set the `Display` to `'on'`.

```
opt = greyestOptions('InitialState','backcast','Display','on');
```

Alternatively, use dot notation to set the values of `opt`.

```
opt = greyestOptions;
opt.InitialState = 'backcast';
opt.Display = 'on';
```

## References

[1] Wills, Adrian , B. Ninness, and S. Gibson. "On Gradient-Based Search for Multivariable System Estimates" IFAC World Congress, Prague, 2005.

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

## See Also

greyest | idgrey | idnlgrey | pem | ssest



**Purpose** True for linear model with time delays

**Syntax**  
B = hasdelay(sys)  
B = hasdelay(sys, 'elem')

**Description** B = hasdelay(sys) returns 1 (true) if the model sys has input delays, output delays, or I/O delays, and 0 (false) otherwise. If sys is a model array, then B is true if least one model in sys has delays.

B = hasdelay(sys, 'elem') returns a logical array of the same size as the model array sys. The logical array indicates which models in sys have delays.

**See Also** absorbDelay | totaldelay

## Purpose

Multiple-output ARX polynomials, impulse response, or step response model

---

**Note** `idarx` will be removed in a future release. Use `idpoly` instead.

---

## 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.

$$y(t) + A_1 y(t-1) + A_2 y(t-2) + \dots + A_{na} y(t-na) = B_0 u(t) + B_1 u(t-1) + \dots + B_{nb} u(t-nb) + e(t)$$

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:

$$A(:, :, k+1) = A_k \\ A(:, :, 1) = \text{eye}(ny)$$

Similarly **B** is an  $ny$ -by- $nu$ -by- $(nb+1)$  array with:

$$B(:, :, k+1) = B_k$$

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).

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]]);
```

`arx` | `arxdata` | | `idpoly`

## How To

- “Using Linear Model for Nonlinear ARX Estimation”

**Purpose** Time- or 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)
```

**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 signal `y` 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 “Properties” on page 2-295.

`data = iddata(idfrd_object)` transforms an `idfrd` object to a frequency-domain `iddata` object.

## Arguments

`y`  
Name of MATLAB variable that represents the output signal from a 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 to a 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.

## Constructor

## Requirements for Constructing an `iddata` Object

To construct an `iddata` object, you must have already imported data into the MATLAB workspace, as described in “Time-Domain Data Representation”.

## Constructing an iddata Object for Time-Domain Data

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

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

You can also specify additional properties, as follows:

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

For more information about accessing object properties, see “Properties” on page 2-295.

Here, *Ts* is the sampling time, or the time interval, between successive data samples:

- For uniformly sampled data, *Ts* is a scalar value equal to the sampling interval of your experiment.
- For nonuniformly sampled data, *Ts* is [], and the value of the `SamplingInstants` property is a column vector containing individual time values. For example:

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

where `TimeVector` represents a vector of time values.

---

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

---

The default time unit is seconds, but you can specify any unit string using the `TimeUnit` property. For more information about iddata time properties, see “Modifying Time and Frequency Vectors”.

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

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

where  $y$  is the output data,  $[\ ]$  indicates empty input data, and  $T_s$  is the sampling interval.

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

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

MATLAB returns the following output:

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

```
Outputs      Unit (if specified)  
  y1
```

```
Inputs      Unit (if specified)  
  u1
```

The default channel name 'y1' is assigned to the first and only output channel. When  $y_2$  contains several channels, the channels are assigned default names 'y1', 'y2', 'y2', ..., 'yn'. Similarly, the default channel name 'u1' is assigned to the first and only input channel. For more information about naming channels, see “Naming, Adding, and Removing Data Channels”.



## Constructing an iddata Object for Frequency-Domain Data

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

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

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

---

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

---

For more information about `iddata` time and frequency properties, see “Modifying Time and Frequency Vectors”.

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

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

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

For more information about accessing object properties, see “Properties” on page 2-295.

## Properties

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

Use `set` or dot notation to set a property of an existing object.

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,  $n_y$  is the number of output channels,  $n_u$  is the number of input channels, and  $N_e$  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 be enough to uniquely identify the property.

---

Property Name	Description	Value
Domain	Specifies whether the data is in the time domain or frequency domain.	<ul style="list-style-type: none"><li>'Frequency' — Frequency-domain data.</li><li>'Time' (Default) — Time-domain data.</li></ul>
ExperimentName	Name of each data set contained in the <code>iddata</code> object.	For $N_e$ experiments, a 1-by- $N_e$ 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 $N_e$ experiments, a 1-by- $N_e$ cell array and each cell contains the frequencies of the corresponding experiment.
InputData	Name of MATLAB variable that stores the input signal to a system.	For $n_u$ input channels and N data samples, this is an N-by- $n_u$ matrix.

Property Name	Description	Value
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.
InterSample	Specifies the behavior of the input signals between samples for transformations between discrete-time and continuous-time.	<p>For a single experiment:</p> <ul style="list-style-type: none"> <li>• zoh— (Default) Zero-order hold maintains a piecewise-constant input signal between samples.</li> <li>• foh— First-order hold maintains a piecewise-linear input signal between samples.</li> <li>• b1— Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.</li> </ul> <p>For Ne experiments, InterSample is an nu-by-Ne cell array. Each cell contains one of these values corresponding to each experiment.</p>

# iddata

Property Name	Description	Value
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 a system.	For $n_y$ output channels and $N$ samples, this is an $N$ -by- $n_y$ matrix.
OutputName	For a multiple-output system, specifies the names of individual output channels.	Cell array of length $n_y$ -by-1 contains the name string of each output channel. Default names are <code>{'y1'; 'y2'; ...}</code> .
OutputUnit	Specifies the units of each output channel.	For $n_y$ output channels, a cell array of length $n_y$ -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 <code>inf</code> . For a multiple-input signal, this is an $n_u$ -by-1 vector and the $k$ th entry contains the period of the $k$ th input. For $N_e$ experiments, this is a 1-by- $N_e$ cell array and each cell contains a scalar or vector of periods for the corresponding experiment.

Property Name	Description	Value
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. Specify TimeUnit as one of the following: 'nanoseconds', 'microseconds', 'milliseconds' (default), 'minutes', 'hours', 'days', 'weeks', 'months' or 'years'.
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.  <hr/> <b>Note</b> Your data must be uniformly sampled. <hr/>	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

Property Name	Description	Value
		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, Tstart is a 1-by-Ne cell array and each cell contains the starting time of the corresponding experiment.
FrequencyUnit	(Frequency-domain data only) Frequency unit.	Specifies the units of the frequency vector (see Frequency). Specify as one of the following: 'rad/TimeUnit', 'cycles/TimeUnit', 'rad/s', 'Hz', 'kHz', 'MHz', 'GHz', or 'rpm'. The units 'rad/TimeUnit' and 'cycles/TimeUnit' are relative to the time units specified in the TimeUnit property. Setting FrequencyUnit does not change the frequency vector. To convert the units and automatically scale frequency points, use chgFreqUnit.
UserData	Additional comments.	Text string.

To view the properties, use the get command. For example:

```
load dryer2 % Load input u2 and output y2
```

---

```
data = iddata(y2,u2,0.08); % Create iddata object
get(data)                 % Get property values of data
```

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

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

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

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

or equivalently:

```
data.ts = 0.05
```

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

---

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

---

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

```
% Load input u2 and output y2
load dryer2;
% Create iddata object
data = iddata(y2,u2,0.08);
% Take the Fourier transform of the data
```

# iddata

---

```
% transforming it to frequency domain
data = fft(data)
% Get property values of data
get(data)
```

## See Also

```
advice | detrend | fcat | getexp | idfilt | idfrd | plot | resample
| size
```



---

<b>Purpose</b>	Open System Identification Tool GUI
<b>Syntax</b>	<code>ident</code> <code>ident(session,path)</code>
<b>Description</b>	<p><code>ident</code> opens the System Identification Tool GUI.</p> <p><code>ident(session,path)</code> opens the saved session <code>session</code> in the System Identification Tool GUI. <code>path</code> specifies the location of this file. Omit <code>path</code> when the session file is on <code>MATLABPATH</code>.</p> <p>You can also open the System Identification Tool interactively. On the <b>Apps</b> tab of the MATLAB desktop, in the <b>Apps</b> section, click <b>System Identification</b>.</p>
<b>Input Arguments</b>	<p><b>session</b></p> <p>Session file to be opened using the System Identification Tool GUI.</p> <p>You create a <code>session</code> file by saving a running session of the GUI. <code>session</code> contains the set of data objects, models and layout settings in use at the time of saving. If the GUI is already open, <code>ident(session)</code> merges the contents of the new session file with those already present in the GUI.</p> <p><b>path</b></p> <p>Location of <code>session</code> file.</p> <p>You do not need to specify <code>path</code> if <code>session</code> is on the MATLAB path.</p>
<b>Examples</b>	<p>Open a saved session <code>iddata1</code>:</p> <pre>ident('iddata1.sid')</pre> <p>Open a saved session <code>mydata</code> in a specified folder:</p> <pre>ident('mydata.sid','\matlab\data\cdplayer\')</pre>
<b>See Also</b>	<code>midprefs</code>

## **How To**

- “Working with the System Identification Tool GUI”

**Purpose** Set System Identification Toolbox preferences

**Syntax** `identpref`

**Description** `identpref` opens a Graphical User Interface (GUI) which allows you to change the System Identification Toolbox preferences. Preferences set in this GUI affect future plots only (existing plots are not altered).

Your preferences are stored to disk (in a system-dependent location) and will be automatically reloaded in future MATLAB sessions using the System Identification Toolbox software.

**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]; ... ;[wpl,wph]
```

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  $W_f$  is a vector of possibly complex values that define the filter's frequency response, so that the inputs and outputs at frequency  $Z$ .Frequency( $k_f$ ) are multiplied by  $W_f(k_f)$ .  $W_f$  is a column vector of length = number of frequencies in  $Z$ . If the data object has several experiments,  $W_f$  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  $Z_f$ .

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`' than just to prefilter the data. The proper values for '`Focus`' are the same as the argument `filter` in `idfilt`.

## Algorithms

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

`iddata` | `resample`

**Purpose** Frequency-response data or model

**Syntax**

```
h = idfrd(Response,Freq,Ts)
h = idfrd(Response,Freq,Ts,...
    'CovarianceData',Covariance,'SpectrumData',Spec,...
    'NoiseCovariance',Speccov)
h = idfrd(Response,Freq,Ts,...
    'P1',V1,'PN',VN)
h = idfrd(mod)
h = idfrd(mod,Freqs)
```

**Description** `h = idfrd(Response,Freq,Ts)` constructs an `idfrd` object that stores the frequency response `Response` of a linear system at frequency values `Freq`. `Ts` is the sampling time interval. For a continuous-time system, set `Ts=0`.

`h = idfrd(Response,Freq,Ts,...`  
`'CovarianceData',Covariance,'SpectrumData',Spec,...`  
`'NoiseCovariance',Speccov)` also stores the uncertainty of the response `Covariance`, the spectrum of the additive disturbance (noise) `Spec`, and the covariance of the noise `Speccov`.

`h = idfrd(Response,Freq,Ts,...`  
`'P1',V1,'PN',VN)` constructs an `idfrd` object that stores a frequency-response model with properties specified by the `idfrd` model property-value pairs.

`h = idfrd(mod)` converts a System Identification Toolbox or Control System Toolbox™ linear model to frequency-response data at default frequencies, including the output noise spectra and their covariance.

`h = idfrd(mod,Freqs)` converts a System Identification Toolbox or Control System Toolbox linear model to frequency-response data at frequencies `Freqs`.

For a model

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

stores the transfer function estimate  $G(e^{i\omega})$ , as well as the spectrum of the additive noise ( $\Phi_v$ ) at the output

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

where  $\lambda$  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  $\omega = \text{Freqs}(kf)$ . 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. `Ts = 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 format for spectrum information 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  $n_y$ -by- $n_y$ -by- $N_f$ , such that `speccov(ky1,ky1,kf)` is the variance of the corresponding power spectrum.

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 linear identified model, `mod`.

If the frequencies `Freqs` are not specified, a default choice is made based on the dynamics of the model `mod`.

The estimated covariances are computed using the Gauss approximation formula from the uncertainty information in `mod`. For grey-box models (`idgrey`), numerical differentiation is applied. The step sizes for the numerical derivatives are determined by `nuderst`.

Intersample behavior: For discrete-time frequency response data ( $T_s > 0$ ), you can also specify the intersample behavior of the input signal that was in effect when the samples were collected originally from an experiment. To specify the intersample behavior, use:

```
mf = idfrd(Response,Freq,Ts,'InterSample','zoh');
```

For multi-input systems, specify the intersample behavior using an  $N_u$ -by-1 cell array, where  $N_u$  is the number of inputs. The `InterSample` property is irrelevant for continuous-time data.

Frequency responses for submodels can be obtained by the standard subreferencing, `h = idfrd(m(2,3))`. `h = idfrd(m(:,[]))` gives an `h` that just contains `SpectrumData`.

The `idfrd` models can be graphed with `bode`, `spectrum`, and `nyquist`, which all accept mixtures of parametric models, such as `idtf` and `idfrd` models as arguments. Note that `spa`, `spafdr`, and `etfe` return their estimation results as `idfrd` objects.



## Constructor

The `idfrd` represents complex frequency-response data. Before you can create an `idfrd` object, you must import your data as described in “Frequency-Response Data Representation”.

---

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

---

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

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

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

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

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

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

## Properties

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

Use `set` or dot notation to set a property of an existing object.

Property Name	Description
ResponseData	3-D array of the complex-valued frequency response as described above. For SISO systems use <code>Response(1,1,:)</code> 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 above.
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.
FrequencyUnit	Unit of the frequency vector. Specify as one of the following: 'rad/TimeUnit', 'cycles/TimeUnit', 'rad/s', 'Hz', 'kHz', 'MHz', 'GHz', or 'rpm'. The units 'rad/TimeUnit' and 'cycles/TimeUnit' are relative to the time units specified in TimeUnit. Changing this property changes the overall system behavior. Use <code>chgFreqUnit</code> to convert between frequency units without modifying system behavior.

Property Name	Description
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, and nyquist.

Property Name	Description
InterSample	<p data-bbox="866 319 1262 635">Intersample behavior of inputs. Specifies the behavior of the input signals between samples for transformations between discrete-time and continuous-time. This property is meaningful for discrete-time <code>idfrd</code> models only. Specify <code>InterSample</code> as one of the following:</p> <ul data-bbox="866 673 1285 1402" style="list-style-type: none"><li data-bbox="866 673 1285 826">• <code>'zoh'</code> — The input signal used for construction/estimation of the frequency response data was subject to a zero-order-hold filter.</li><li data-bbox="866 847 1285 933">• <code>'foh'</code> — The input signal was subject to a first-order-hold filter.</li><li data-bbox="866 954 1285 1402">• <code>'b1'</code> — The input signal has no power above the Nyquist frequency (<math>\pi/\text{sys.Ts}</math> rad/s). This is typically the case when the input signal is measured experimentally using an anti-aliasing filter and a sampler. Ideally, treat the data as continuous-time. That is, if the signals used for the estimation of the frequency response were subject to anti-aliasing filters, set <code>sys.Ts</code> to zero.</li></ul>

Property Name	Description
	For multi-input data, specify <code>InterSample</code> as an $Nu$ -by-1 cell array, where $Nu$ is the number of inputs.
Notes	An arbitrary field to store extra information and notes about the object.
UserData	An arbitrary field for any possible use.
Report	<p>Information about the estimation process that is behind the frequency data. It contains the following fields:</p> <ul style="list-style-type: none"> <li>• <b>Status:</b> Specifies whether the model was created directly, transformed or estimated.</li> <li>• <b>Method:</b> The identification routine that created the model.</li> <li>• <b>WindowSize:</b> If the model was estimated by <code>spa</code>, <code>spafdr</code>, or <code>etfe</code>, the size of window (input argument <code>M</code>, the resolution parameter) that was used. This is scalar or a vector.</li> <li>• <b>DataUsed:</b> Information on the estimation data such as its name, type, sample time and intersample behavior.</li> </ul>

To view the properties of the `idfrd` object, you can use the `get` command. The following example shows how to create an `idfrd` object

that contains 100 frequency-response values with a sampling time interval of 0.08 s and get its properties:

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

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

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

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

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

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

**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'})`.

## 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 of h1, h2, ..., hN 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.

```
load iddata1 z1;  
m = armax(z1,[2 2 2 1]);  
g = spa(z1)  
g = spafdr(z1,[],{1e-3,10})  
bode(g,m)
```

## See Also

bode | etfe | freqresp | nyquist | spa | spafdr | tfest

## Purpose

Linear ODE (grey-box model) with identifiable parameters

## Syntax

```
sys = idgrey(odefun,parameters,fcn_type)
sys = idgrey(odefun,parameters,fcn_type,optional_args)
sys = idgrey(odefun,parameters,fcn_type,optional_args,Ts)
sys =
idgrey(odefun,parameters,fcn_type,optional_args,Ts,Name,
      Value)
```

## Description

`sys = idgrey(odefun,parameters,fcn_type)` creates a linear grey-box model with identifiable parameters, `sys`. `odefun` specifies the user-defined function that relates the model parameters, `parameters`, to its state-space representation.

`sys = idgrey(odefun,parameters,fcn_type,optional_args)` creates a linear grey-box model with identifiable parameters using the optional arguments required by `odefun`.

`sys = idgrey(odefun,parameters,fcn_type,optional_args,Ts)` creates a linear grey-box model with identifiable parameters with the specified sample time, `Ts`.

`sys = idgrey(odefun,parameters,fcn_type,optional_args,Ts,Name,Value)` creates a linear grey-box model with identifiable parameters with additional options specified by one or more `Name,Value` pair arguments.

## Object Description

An `idgrey` model represents a system as a continuous-time or discrete-time state-space model with identifiable (estimable) coefficients.

A state-space model of a system with input vector,  $u$ , output vector,  $y$ , and disturbance,  $e$ , takes the following form in continuous time:

$$\begin{aligned}\dot{x}(t) &= Ax(t) + Bu(t) + Ke(t) \\ y(t) &= Cx(t) + Du(t) + e(t)\end{aligned}$$



In discrete time, the state-space model takes the form:

$$\begin{aligned}x[k+1] &= Ax[k] + Bu[k] + Ke[k] \\y[k] &= Cx[k] + Du[k] + e[k]\end{aligned}$$

For `idgrey` models, the state-space matrices  $A$ ,  $B$ ,  $C$ , and  $D$  are expressed as a function of user-defined parameters using a MATLAB function. You access estimated parameters using `sys.Structures.Parameters`, where `sys` is an `idgrey` model.

Use an `idgrey` model when you know the system of equations governing the system dynamics explicitly, in the form of ordinary differential or difference equations. You can also use `idgrey` models to prescribe complex relationships and constraints among the parameters that are not achievable by using structured state-space models (`idss`).

You can create an `idgrey` model using the `idgrey` command. You must write a MATLAB function that returns the  $A$ ,  $B$ ,  $C$ , and  $D$  matrices for given values of the estimable parameters and sampling time. The MATLAB function can also return the  $K$  matrix and accept optional input arguments. The matrices returned may represent a continuous-time or discrete-time model, as indicated by the sampling time.

Use the estimating functions `pem` or `greyest` to obtain estimated values for the unknown parameters of an `idgrey` model.

You can convert an `idgrey` model into other dynamic systems, such as `idpoly`, `idss`, `tf`, `ss` etc. You cannot convert a dynamic system into an `idgrey` model.

## Examples

### Create Grey-Box Model with Identifiable Parameters

Create an `idgrey` model to represent a DC motor. Specify the motor time-constant as an estimable parameter, and that the ODE function can return continuous- or discrete-time state-space matrices.

Create the `idgrey` model.

```
odefun = 'motor';
```

```
parameters = 1;  
fcn_type = 'cd';  
optional_args = 0.25;  
Ts = 0;  
sys = idgrey(odefun,parameters,fcn_type,optional_args,Ts);
```

`sys` is an `idgrey` model that is configured to use the shipped file `motor.m` to return the  $A$ ,  $B$ ,  $C$ ,  $D$ , and  $K$  matrices. `motor.m` also returns the initial conditions,  $X0$ . The motor constant,  $\tau$ , is defined in `motor.m` as an estimable parameter, and `parameters = 1` specifies its initial value as 1.

You can use `pem` or `greyest` to refine the estimate for  $\tau$ .

## Configure Identifiable Parameters of Grey-Box Model

Specify minimum constraints for the estimable parameters of an `idgrey` model.

Create an `idgrey` model.

```
odefun = 'ModalFormODE';  
  
sigma1=0.1;  
sigma2=0.1;  
w1=1;  
w2=1;  
B = [1 0 0 0]';  
C = [1 1 1 1];  
  
parameters = {'sigma1',sigma1;...  
              'w1',w1;...  
              'sigma2',sigma2;...  
              'w2',w2;...  
              'B',B;...  
              'C',C};  
fcn_type = 'c';  
  
sys = idgrey(odefun,parameters,fcn_type);
```

sys is an idgrey model that is configured to use the function ModalFormODE to return the  $A$ ,  $B$ ,  $C$ , and  $D$  matrices. The code for the function ModalFormODE is:

```
function [A,B,C,D] = ModalFormODE(sigma1,w1,sigma2,w2,Bpar,Cpar,varargin)
%MODALFORMODE Function that parameterizes a 4th order state-space model
%modal form.
%
% Parameters:
%   sigma1: The absolute value of the real part of the first
%           complex-conjugate pair of poles. Positive scalar.
%           w1: The absolute value of the imaginary part of the first
%              complex-conjugate pair of poles. Positive scalar.
%   sigma2: Similar to sigma1. Positive scalar.
%           w2: Similar to w1. Positive scalar.
%   BPar: A 4-by-1 real vector. No constraints on coefficients.
%   CPar: A 1-by-4 real vector. No constraints on coefficients.
%
% This file parameterizes the state-space model in continuous-time on.
A1 = [-sigma1, w1; -w1, -sigma1];
A2 = [-sigma2, w2; -w2, -sigma2];
A = blkdiag(A1,A2);
B = Bpar;
C = Cpar;
D = 0;
end
```

The function defines  $\sigma_1$ ,  $w_1$ ,  $\sigma_2$ ,  $w_2$ ,  $B_{par}$ , and  $C_{par}$  as estimable parameters.

Specify minimum constraints for some of the estimable parameters.

```
sys.Structure.Parameters(1).Minimum = 0;
sys.Structure.Parameters(2).Minimum = 0;
sys.Structure.Parameters(3).Minimum = 0;
sys.Structure.Parameters(4).Minimum = 0;
```

The first four parameters in `sys.Structure.Parameters` are `sigma1`, `w1`, `sigma2`, and `w2`. This code specifies 0 as the minimum value for these parameters.

You can use `pem` or `greyest` to estimate the estimable parameters of `sys`. When you do so, the software enforces the constraints specified in `sys.Structure` in estimating the parameters.

## Specify Additional Attributes of Grey-Box Model

Create a grey-box model with identifiable parameters. Name the input and output channels of the model, and specify seconds for the model time units.

You can use `Name,Value` pair arguments to specify additional model properties on model creation.

```
odefun = 'motor';  
parameters = 1;  
fcn_type = 'cd';  
optional_args = 0.25;  
Ts = 0;  
sys = idgrey(odefun,parameters,fcn_type,optional_args,Ts,'InputName','Vol'  
            'OutputName',{'Angular Position','Angular Velocity'});
```

To change or specify more attributes of an existing model, you can use dot notation. For example:

```
sys.TimeUnit = 'seconds';
```

## Array of Grey-Box Models

Create an array of grey-box models.

Use the `stack` command to create an array of linear grey-box models.

```
odefun1 = @motor;  
parameters1 = [1 2];  
fcn_type = 'cd';  
optional_args1 = 1;  
sys1 = idgrey(odefun1,parameters1,fcn_type,optional_args1);
```

```

odefun2 = 'motor';
parameters2 = {[1 2]};
optional_args2 = 0.5;
sys2 = idgrey(odefun2,parameters2,fcn_type,optional_args2);

sysarr = stack(1,sys1,sys2);

```

stack creates a 2-by-1 array of idgrey models, sysarr.

## Input Arguments

### odefun

MATLAB function that relates the model parameters to its state-space representation.

odefun specifies, as a string, the name of a MATLAB function (.m, .p, a function handle or .mex\* file). This function establishes the relationship between the model parameters, parameters, and its state-space representation. The function may optionally relate the model parameters to the disturbance matrix and initial states.

If the function is not on the MATLAB path, then specify the full file name, including the path.

The syntax for odefun must be as follows:

```
[A,B,C,D] = odefun(par1,par2,...,parN,Ts,optional_arg1,optional_arg2,
```

Here, the function outputs describe the model in the following linear state-space innovations form:

$$\begin{aligned}
 xn(t) &= Ax(t) + Bu(t) + Ke(t); x(0) = x_0 \\
 y(t) &= Cx(t) + Du(t) + e(t)
 \end{aligned}$$

In discrete time  $xn(t)=x(t+Ts)$  and in continuous time,  $xn(t) = \dot{x}(t)$ .

par1, par2, . . . , parN are model parameters. Each entry may be a scalar, vector or matrix.

Ts is the sample time.

`optional_arg1, optional_arg2, ...` are the optional inputs that `odefun` may require. The values of the optional input arguments are unchanged through the estimation process. However, the values of `par1, par2, ..., parN` are updated during estimation to fit the data. Use optional input arguments to vary the constants and coefficients used by your model without editing `odefun`.

The disturbance matrix,  $K$ , and the initial state values,  $x0$ , are not parametrized and are determined separately, using the `DisturbanceModel` and `InitialState` estimation options, respectively. For more information regarding the estimation options, see `greyestOptions`.

A good choice for achieving the best simulation results is to set the `DisturbanceModel` option to 'none', which fixes  $K$  to zero.

(Optional) Parameterizing Disturbance: `odefun` can also return the disturbance component,  $K$ , using the syntax:

```
[A,B,C,D,K] = odefun(par1,par2,...,parN,Ts,optional_arg1,optional_arg2,...
```

If `odefun` returns a value for  $K$  that contains NaN values, then the estimating function assumes that  $K$  is not parameterized. In this case, the value of the `DisturbanceModel` estimation option determines how  $K$  is handled.

(Optional) Parameterizing Initial State Values: To make the model initial states,  $X0$ , dependent on the model parameters, use the following syntax for `odefun`:

```
[A,B,C,D,K,X0] = odefun(par1,par2,...,parN,Ts,optional_arg1,optional_arg2,...
```

If `odefun` returns a value for  $X0$  that contains NaN values, then the estimating function assumes that  $X0$  is not parameterized. In this case,  $X0$  may be fixed to zero or estimated separately, using the `InitialStates` estimation option.

### **parameters**

Initial values of the parameters required by `odefun`.

Specify `parameters` as a cell array containing the parameter initial values. If your model requires only one parameter, which may itself be a vector or a matrix, you may specify `parameters` as a matrix.

You may also specify parameter names using an  $N$ -by-2 cell array, where  $N$  is the number of parameters. The first column specifies the names, and the second column specifies the values of the parameters.

For example:

```
parameters = {'mass',par1;'stiffness',par2;'damping',par3}
```

### **fcn\_type**

Indicates whether the model is parameterized in continuous-time, discrete-time, or both.

`fcn_type` takes one of the following strings:

- 'c' — `odefun` returns matrices corresponding to a continuous-time system, regardless of the value of `Ts`.
- 'd' — `odefun` returns matrices corresponding to a discrete-time system, whose values may or may not depend on the value of `Ts`.
- 'cd' — `odefun` returns matrices corresponding to a continuous-time system, if `Ts=0`.

Else, if `Ts>0`, `odefun` returns matrices corresponding to a discrete-time system. Select this option to sample your model using the values returned by `odefun`, rather than using the software's internal sample time conversion routines.

### **optional\_args**

Optional input arguments required by `odefun`.

Specify `optional_args` as a cell array.

If `odefun` does not require optional input arguments, specify `optional_args` as `{}`.

### **Ts**

Model sampling time.

If  $T_s$  is unspecified, it is assumed to be:

- -1 — If `fcn_type` is 'd' or 'cd'.  
 $T_s = -1$  indicates a discrete-time model with unknown sampling time.
- 0 — If `fcn_type` is 'c'.  
 $T_s = 0$  indicates a continuous-time model.

## **Name,Value**

Specify optional comma-separated pairs of **Name,Value** arguments, where **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Use **Name,Value** arguments to specify additional properties of `idgrey` models during model creation. For example, `idgrey(odefun,parameters,fcn_type,'InputName','Voltage')` creates an `idgrey` model with the `InputName` property set to `Voltage`.

## **Properties**

`idgrey` object properties include:

### **a,b,c,d**

Values of state-space matrices.

- **a** — State matrix  $A$ , an  $N_x$ -by- $N_x$  matrix, as returned by the ODE function associated with the `idgrey` model.  $N_x$  is the number of states.
- **b** — Input-to-state matrix  $B$ , an  $N_x$ -by- $N_u$  matrix, as returned by the ODE function associated with the `idgrey` model.  $N_u$  is the number of inputs and  $N_x$  is the number of states.



- **c** — State-to-output matrix  $C$ , an  $N_y$ -by- $N_x$  matrix, as returned by the ODE function associated with the `idgrey` model.  $N_x$  is the number of states and  $N_y$  is the number of outputs.
- **d** — Feedthrough matrix  $D$ , an  $N_y$ -by- $N_u$  matrix, as returned by the ODE function associated with the `idgrey` model.  $N_y$  is the number of outputs and  $N_u$  is the number of inputs.

As **a**, **b**, **c**, **d** are returned by the ODE function associated with the `idgrey` model, you can only read these matrices; you cannot set their values.

### **k**

Value of state disturbance matrix,  $K$

**k** is  $N_x$ -by- $N_y$  matrix, where  $N_x$  is the number of states and  $N_y$  is the number of outputs.

- If `odefun` parameterizes the  $K$  matrix, then **k** has the value returned by `odefun`. `odefun` parameterizes the  $K$  matrix if it returns at least 5 outputs and the value of the fifth output does not contain NaN values.
- If `odefun` does not parameterize the  $K$  matrix, then **k** is a zero matrix of size  $N_x$ -by- $N_y$ .  $N_x$  is the number of states and  $N_y$  is the number of outputs. The value is treated as a fixed value of the  $K$  matrix during estimation, unless it is designated to be estimable using the `DisturbanceModel` estimation option.
- Regardless of whether the  $K$  matrix is parameterized by `odefun` or not, you can set the value of the **k** property explicitly as an  $N_x$ -by- $N_y$  matrix.  $N_x$  is the number of states and  $N_y$  is the number of outputs. The specified value is treated as a fixed value of the  $K$  matrix during estimation, unless it is designated to be estimable using the `DisturbanceModel` estimation option.

To create an estimation option set for `idgrey` models, use `greyestOptions`.

### **StateName**

State names. Set `StateName` to a string for first-order models, or to a cell array of strings for models with two or more states. Use an empty string `''` for unnamed states.

**Default:** Empty string `''` for all states

## StateUnit

State units. Use `StateUnit` to keep track of the units each state is expressed in. Set `StateUnit` to a string for first-order models, or to a cell array of strings for models with two or more states. `StateUnit` has no effect on system behavior.

**Default:** Empty string `''` for all states

## Structure

Information about the estimable parameters of the `idgrey` model.

Structure stores information regarding the MATLAB function that parameterizes the `idgrey` model.

- `Structure.Function` — Name or function handle of the MATLAB function used to create the `idgrey` model.
- `Structure.FcnType` — Indicates whether the model is parameterized in continuous-time, discrete-time, or both.
- `Structure.Parameters` — Information about the estimated parameters. `Structure.Parameters` contains the following fields:
  - `Value` — Parameter values. For example, `sys.Structure.Parameters(2).Value` contains the initial or estimated values of the second parameter.  
NaN represents unknown parameter values.
  - `Minimum` — Minimum value that the parameter can assume during estimation. For example, `sys.Structure.Parameters(1).Minimum = 0` constrains the first parameter to be greater than or equal to zero.

- **Maximum** — Maximum value that the parameter can assume during estimation.
  - **Free** — Boolean specifying whether the parameter is a free estimation variable. If you want to fix the value of a parameter during estimation, set the `Free = false` for the corresponding entry.
  - **Scale** — Scale of the parameter's value. `Scale` is not used in estimation.
  - **Info** — Structure array for storing parameter units and labels. The structure has `Label` and `Unit` fields.  
Use these fields for your convenience, to store strings that describe parameter units and labels.
- `Structure.ExtraArgs` — Optional input arguments required by the ODE function.
  - `Structure.StateName` — Names of the model states.
  - `Structure.StateUnit` — Units of the model states.

### **NoiseVariance**

The variance (covariance matrix) of the model innovations  $e$ .

An identified model includes a white, Gaussian noise component  $e(t)$ . `NoiseVariance` is the variance of this noise component. Typically, the model estimation function (such as `greyest` or `pem`) determines this variance.

For SISO models, `NoiseVariance` is a scalar. For MIMO models, `NoiseVariance` is a  $N_y$ -by- $N_y$  matrix, where  $N_y$  is the number of outputs in the system.

### **Report**

Information about the estimation process.

`Report` contains the following fields:

- **Status** — Whether model was obtained by construction, estimated, or modified after estimation.
- **Method** — Name of estimation method used.
- **InitialState** — Initial state handling during model estimation.
- **DisturbanceModel** — Disturbance component (the  $K$  matrix) handling of the model during estimation.
- **Fit** — Quantitative quality assessment of estimation, including percent fit to data and final prediction error.
- **Parameters** — Estimated values of model parameters and their covariance
- **OptionsUsed** — Options used during estimation (see `greyestOptions`).
- **RandState** — Random number stream state at the start of estimation.
- **DataUsed** — Attributes of the data used for estimation, such as name and sampling time.
- **Termination** — Termination conditions for the iterative search scheme used for prediction error minimization, such as final cost value and stopping criterion.

### **InputDelay**

Input delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. For continuous-time systems, specify input delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify input delays in integer multiples of the sampling period  $T_s$ . For example, `InputDelay = 3` means a delay of three sampling periods.

For a system with  $N_u$  inputs, set `InputDelay` to an  $N_u$ -by-1 vector, where each entry is a numerical value representing the input delay for the corresponding input channel. You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0 for all input channels

### **OutputDelay**

Output delays.

For identified systems, like `idgrey`, `OutputDelay` is fixed to zero.

### **Ts**

Sampling time.

For continuous-time models,  $T_s = 0$ . For discrete-time models,  $T_s$  is a positive scalar representing the sampling period expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sampling time, set  $T_s = -1$ .

Changing this property does not discretize or resample the model.

For `idgrey` models, there is no unique default value for  $T_s$ .  $T_s$  depends on the value of `fcn_type`.

### **TimeUnit**

String representing the unit of the time variable, any time delays in the model (for continuous-time models), and the sampling time  $T_s$  (for discrete-time models). `TimeUnit` can take the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'

- 'years'

Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

## InputName

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

Alternatively, use automatic vector expansion to assign input names for multi-input models. For example, if `sys` is a two-input model, enter:

```
sys.InputName = 'controls';
```

The software automatically expands the input names to `{'controls(1)'; 'controls(2)'}`.

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `sys.u` is equivalent to `sys.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** Empty string '' for all input channels

## InputUnit

Input channel units. Use `InputUnit` to keep track of input signal units. Set `InputUnit` to a string for single-input model, or to a cell array of strings for a multi-input model. `InputUnit` has no effect on system behavior.

**Default:** Empty string '' for all input channels

## InputGroup

Input channel groups. The `InputGroup` property lets you assign the input channels of MIMO systems into groups and refer to each group by name. Specify input groups as a structure whose field names are the group names and whose field values are the input channels belong to each group. For example:

```
sys.InputGroup.controls = [1 2];  
sys.InputGroup.noise = [3 5];
```

creates input groups named `controls` and `noise` that include input channels 1, 2 and 3, 5, respectively. You can then extract the subsystem from the `controls` inputs to all outputs using:

```
sys(:, 'controls')
```

**Default:** Struct with no fields

## OutputName

Output channel names. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

Alternatively, use automatic vector expansion to assign output names for multi-output models. For example, if `sys` is a two-output model, enter:

```
sys.OutputName = 'measurements';
```

The software automatically expands the output names to `{'measurements(1)'; 'measurements(2)'}`.

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `sys.y` is equivalent to `sys.OutputName`.

Output channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems

- Specifying connection points when interconnecting models

**Default:** Empty string '' for all input channels

## **OutputUnit**

Output channel units. Use `OutputUnit` to keep track of output signal units. Set `OutputUnit` to a string for single-input model, or to a cell array of strings for a multi-input model. `OutputUnit` has no effect on system behavior.

**Default:** Empty string '' for all input channels

## **OutputGroup**

Output channel groups. The `OutputGroup` property lets you assign the output channels of MIMO systems into groups and refer to each group by name. Specify output groups as a structure whose field names are the group names and whose field values are the output channels belong to each group. For example:

```
sys.OutputGroup.temperature = [1];  
sys.InputGroup.measurement = [3 5];
```

creates output groups named `temperature` and `measurement` that include output channels 1, and 3, 5, respectively. You can then extract the subsystem from all inputs to the `measurement` outputs using:

```
sys('measurement', :)
```

**Default:** Struct with no fields

## **Name**

System name. Set `Name` to a string to label the system.

**Default:** ''

## **Notes**



Any text that you wish to associate with the system. Set `Notes` to a string or a cell array of strings.

**Default:** `{}`

#### **UserData**

Any type of data you wish to associate with system. Set `UserData` to any MATLAB data type.

**Default:** `[]`

#### **See Also**

`greyest` | `greyestOptions` | `pem` | `idnlgrey` | `idss` | `ssest` | `getpvec` | `setpvec`

# idinput

---

**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 so 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  $\pi * [\text{band}(1) \text{ band}(2)]$ . 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.

## Algorithms

Very simple algorithms are used. The frequency contents are achieved for 'rgs' by an eighth-order Butterworth, noncausal filter, using `idfilt`. 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);
spectrum(etfe(u), 'r*', etfe(u2), '+')
```

# idmodel

---

## **Purpose**

Superclass for linear models

---

**Note** `idmodel` has been removed. See `idgrey`, `idpoly`, `idproc`, `idss` or `idtf` instead.

---

**Purpose**

Nonlinear ARX model

**Syntax**

```
m = idnlarx([na nb nk])  
m = idnlarx([na nb nk],Nonlinearity)  
m = idnlarx([na nb nk],Nonlinearity, 'PropertyName',  
    PropertyValue)  
m = idnlarx(LinModel)  
m = idnlarx(LinModel,Nonlinearity)  
m = idnlarx(LinModel,Nonlinearity, 'PropertyName',  
    PropertyValue)
```

**Description**

Represents nonlinear ARX model. The nonlinear ARX structure is an extension of the linear ARX structure and contains linear and nonlinear functions. For more information, see “Nonlinear ARX Model Extends the Linear ARX Structure”.

Typically, you use the `nlarx` command to both construct the `idnlarx` object and estimate the model parameters. You can configure the model properties directly in the `nlarx` syntax.

You can also use the `idnlarx` constructor to create the nonlinear ARX model structure and then estimate the parameters of this model using `nlarx` or `pem`.

For `idnlarx` object properties, see:

- “`idnlarx` Model Properties” on page 2-343
- “`idnlarx` Algorithm Properties” on page 2-346
- “`idnlarx` Advanced Algorithm Properties” on page 2-350
- “`idnlarx` EstimationInfo Properties” on page 2-351

**Construction**

`m = idnlarx([na nb nk])` creates an `idnlarx` object using a default wavelet network as its nonlinearity estimator. *na*, *nb*, and *nk* are positive integers that specify model orders and delays.

`m = idnlarx([na nb nk],Nonlinearity)` specifies a nonlinearity estimator *Nonlinearity*, as a nonlinearity estimator object or string representing the nonlinearity estimator type.

`m = idnlarx([na nb nk],Nonlinearity,'PropertyName',PropertyValue)` creates the object using options specified as `idnlarx` property name and value pairs. Specify *PropertyName* inside single quotes.

`m = idnlarx(LinModel)` creates an `idnlarx` object using a linear model (in place of `[na nb nk]`), and a wavelet network as its nonlinearity estimator. *LinModel* is a discrete time input-output polynomial model of ARX structure (`idpoly`) for single-output systems and `idarx` object for multi-output systems. *LinModel* sets the model orders, input delay, input-output channel names and units, sample time, and time unit of *m*, and the polynomials initialize the linear function of the nonlinearity estimator.

`m = idnlarx(LinModel,Nonlinearity)` specifies a nonlinearity estimator *Nonlinearity*.

`m = idnlarx(LinModel,Nonlinearity,'PropertyName',PropertyValue)` creates the object using options specified as `idnlarx` property name and value pairs.

## Input Arguments

### **na nb nk**

Positive integers that specify the model orders and delays.

For *ny* output channels and *nu* input channels, *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, where each row defines the orders for the corresponding output.

### **Nonlinearity**

Nonlinearity estimator, specified as a nonlinearity estimator object or string representing the nonlinearity estimator type.



'wavenet' or wavenet object (default)	Wavelet network
'sigmoidnet' or sigmoidnet object	Sigmoid network
'treepartition' or treepartition object	Binary-tree
'linear' or [ ] or linear object	Linear function
neuralnet object	Neural network
customnet object	Custom network

Specifying a string creates a nonlinearity estimator object with default settings. Use object representation to configure the properties of a nonlinearity estimator.

For *ny* output channels, you can specify nonlinear estimators individually for each output channel by setting *Nonlinearity* to an *ny-by-1* cell array or object array of nonlinearity estimators. To specify the same nonlinearity for all outputs, specify *Nonlinearity* as a single nonlinearity estimator.

### LinModel

Discrete time input-output polynomial model of ARX structure, typically estimated using the `arx` command:

- `idpoly` object for single-output systems
- `idarx` object for multi-output systems

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

```
% Get the model time unit
get(m, 'TimeUnit')
% Get value of Nonlinearity property
m.Nonlinearity
```

You can specify property name-value pairs in the model estimator or constructor to configure the model structure and estimation algorithm.

## idnlarx Model Properties

Use set or dot notation to set a property of an existing object.

The following table summarizes `idnlarx` model properties. The general `idnmodel` properties also apply to this nonlinear model object (see the corresponding reference page).

Property Name	Description
Algorithm	A structure that specifies the estimation algorithm options, as described in “ <code>idnlarx</code> Algorithm Properties” on page 2-346.
CustomRegressors	<p>Custom expression in terms of standard regressors. Assignable values:</p> <ul style="list-style-type: none"> <li>• Cell array of strings. For example: <code>{'y1(t-3)^3','y2(t-1)*u1(t-3)','sin(u3(t-2))'}</code>.</li> <li>• Object array of <code>customreg</code> objects. Create these objects using commands such as <code>customreg</code> and <code>polyreg</code>. For more information, see the corresponding reference pages.</li> </ul>
EstimationInfo	A read-only structure that stores estimation settings and results, as described in “ <code>idnlarx</code> EstimationInfo Properties” on page 2-351.
Focus	<p>Specifies 'Prediction' or 'Simulation'. Assignable values:</p> <ul style="list-style-type: none"> <li>• 'Prediction' (default) — The estimation algorithm minimizes <math>\ y - \hat{y}\ </math>, where <math>\hat{y}</math> is the 1-step ahead predicted output. This algorithm does not necessarily minimize the simulation error.</li> <li>• 'Simulation' — The estimation algorithm minimizes the simulation error and optimizes the results of <code>compare(data,model,Inf)</code>. That is, when computing <math>\hat{y}</math>, <math>y</math> in the regressors in <math>F</math> are replaced by values simulated from the input only. 'Simulation' requires that the model include only differentiable nonlinearities.</li> </ul>

Property Name	Description
	<hr/> <p><b>Note</b> If your model includes the <code>treepartition</code> or <code>neuralnet</code> nonlinearity, the algorithm always uses <code>'prediction'</code>, regardless of the <code>Focus</code> value. If your model includes the <code>wavenet</code> nonlinearity, the first estimation of this model uses <code>'prediction'</code>.</p> <hr/>
NonlinearRegressors	<p>Specifies which standard or custom regressors enter the nonlinear block. For multiple-output models, use cell array of <math>n_y</math> elements (<math>n_y</math> = number of model outputs). For each output, assignable values are:</p> <ul style="list-style-type: none"> <li>• <code>'all'</code> — All regressors enter the nonlinear block.</li> <li>• <code>'search'</code> — Specifies that the estimation algorithm searches for the best regressor combination. This is useful when you want to reduce a large number of regressors entering the nonlinear function block or the nonlinearity estimator.</li> <li>• <code>'input'</code> — Input regressors only.</li> <li>• <code>'output'</code> — Output regressors only.</li> <li>• <code>'standard'</code> — Standard regressors only.</li> <li>• <code>'custom'</code> — Custom regressors only.</li> <li>• <code>'[]'</code> — No regressors enter the nonlinear block.</li> <li>• 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 <code>getreg</code>.</li> </ul>

Property Name	Description
Nonlinearity	<p>Nonlinearity estimator object. Assignable values include <code>wavenet</code> (default), <code>sigmoidnet</code>, <code>treepartition</code>, <code>customnet</code>, <code>neuralnet</code>, and <code>linear</code>. If the model contains only one regressor, you can also use <code>saturation</code>, <code>deadzone</code>, <code>pwlinear</code>, or <code>poly1d</code>.</p> <p>For <math>n_y</math> outputs, <code>Nonlinearity</code> is an <math>n_y</math>-by-1 array. For example, <code>[sigmoidnet;wavenet]</code> for a two-output model. When you specify a scalar object, this nonlinearity applies to all outputs.</p>
na nb nk	<p>Nonlinear ARX model orders and input delays, where <code>na</code> is the number of output terms, <code>nb</code> is the number of input terms, and <code>nk</code> is the delay from input to output in terms of the number of samples.</p> <p>For <math>n_y</math> outputs and <math>n_u</math> inputs, <code>na</code> is an <math>n_y</math>-by-<math>n_y</math> matrix whose <math>i</math>-<math>j</math>th entry gives the number of delayed <math>j</math>th outputs used to compute the <math>i</math>th output. <code>nb</code> and <code>nk</code> are <math>n_y</math>-by-<math>n_u</math> matrices.</p>

## 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 “ <code>idnlarx</code> Advanced Algorithm Properties” on page 2-350.
Criterion	<p>The search method of <code>lsqnonlin</code> supports the Trace criterion only.</p> <p>Use for multiple-output models only. <code>Criterion</code> can have the following values:</p> <ul style="list-style-type: none"> <li>• <code>'Det'</code>: Minimize <math>\det(E' * E)</math>, where <math>E</math> represents the prediction error. This is the optimal choice in a statistical</li> </ul>

Property Name	Description
	<p>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.</p> <ul style="list-style-type: none"> <li>• 'Trace': Minimize the trace of the weighted prediction error matrix <math>\text{trace}(E' * E * W)</math>, where <math>E</math> is the matrix of prediction errors, with one column for each output, and <math>W</math> is a positive semi-definite symmetric matrix of size equal to the number of outputs. By default, <math>W</math> is an identity matrix of size equal to the number of model outputs (so the minimization criterion becomes <math>\text{trace}(E' * E)</math>, or the traditional least-squares criterion). You can specify the relative weighting of prediction errors for each output using the <code>Weighting</code> field of the <code>Algorithm</code> property. If the model contains <code>neuralnet</code> or <code>treepartition</code> as one of its nonlinearity estimators, weighting is not applied because estimations are independent for each output.</li> </ul> <p>Both the <code>Det</code> and <code>Trace</code> criteria are derived from a general requirement of minimizing a weighted sum of least squares of prediction errors. <code>Det</code> 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 <code>Trace</code> criterion.</p> <p>If you want to achieve better accuracy for a particular channel in MIMO models, use <code>Trace</code> with weighting that favors that channel. Otherwise, use <code>Det</code>. If you use <code>Det</code>, check <code>cond(model.NoiseVariance)</code> after estimation. If the matrix is ill-conditioned, try using the <code>Trace</code> criterion. You can also use <code>compare</code> on validation data to check whether the relative error for different channels corresponds to your needs or expectations. Use the <code>Trace</code> criterion if you need to modify the relative errors, and check <code>model.NoiseVariance</code> to determine what weighting modifications to specify.</p>

Property Name	Description
IterWavenet	<p>(For wavenet nonlinear estimator only)            Toggles performing iterative or noniterative estimation.            Default: 'auto'.            Assignable values:</p> <ul style="list-style-type: none"> <li>• 'auto' — First estimation is noniterative and subsequent estimation are iterative.</li> <li>• 'On' — Perform iterative estimation only.</li> <li>• 'Off' — Perform noniterative estimation only.</li> </ul>
LimitError	<p>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).</p>
MaxIter	<p>Maximum number of iterations for the estimation algorithm, specified as a positive integer.            Default: 20.</p>
MaxSize	<p>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.</p> <hr/> <p><b>Note</b> The original data matrix of <math>u</math> and <math>y</math> must be smaller than MaxSize.</p> <hr/>

Property Name	Description
SearchMethod	<p>Method used by the iterative search algorithm. Assignable values:</p> <ul style="list-style-type: none"> <li>• 'Auto' — Automatically chooses from the following methods.</li> <li>• 'gn' — Subspace Gauss-Newton method.</li> <li>• 'gna' — Adaptive Gauss-Newton method.</li> <li>• 'grad' — A gradient method.</li> <li>• 'lm' — Levenberg-Marquardt method.</li> <li>• 'lsqnonlin' — Nonlinear least-squares method (requires the Optimization Toolbox product). This method only handles the 'Trace' criterion.</li> </ul>
Tolerance	<p>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.</p>
Display	<p>Toggles displaying or hiding estimation progress information in theMATLAB Command Window. Default: 'Off'. Assignable values:</p> <ul style="list-style-type: none"> <li>• 'Off' — Hide estimation information.</li> <li>• 'On' — Display estimation information.</li> </ul>
Weighting	<p>(For multiple-output models only)</p> <p>Specifies the relative importance of outputs in MIMO models (or reliability of corresponding data) as a positive semi-definite matrix <math>W</math>. Use when Criterion = 'Trace' for weighted trace minimization. By default, Weighting is an identity matrix of size equal to the number of outputs.</p>

## 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 $\text{GnPinvConst} * \max(\text{size}(J)) * \text{norm}(J) * \text{eps}$ . 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 <i>regularization</i> when using the Levenberg-Marquardt search method ( <code>Algorithm.SearchMethod='lm'</code> ). Default: 0.001. Assign a positive real value.
LMStep	(For Levenberg-Marquardt search algorithm) Try this next level of <i>regularization</i> 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 $1/\text{LMStep}$ 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 ( <code>Algorithm.SearchMethod</code> property) Default: 10. Assign a positive integer value.



Property Name	Description
MaxFunEvals	The iterations are stopped if the number of calls to the model file exceeds this value. Default: Inf. Assign a positive integer value.
MinParChange	The smallest parameter update allowed per iteration. Default: 1e-16. Assign a positive, real value.
RelImprovement	The iterations are stopped if the relative improvement of the criterion function is less than RelImprovement. Default: 0. Assign a positive real value.  <hr/> <b>Note</b> 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.  <hr/> <b>Note</b> Does not apply to Algorithm.SearchMethod='lsqnonlin'

## idnlarx EstimationInfo Properties

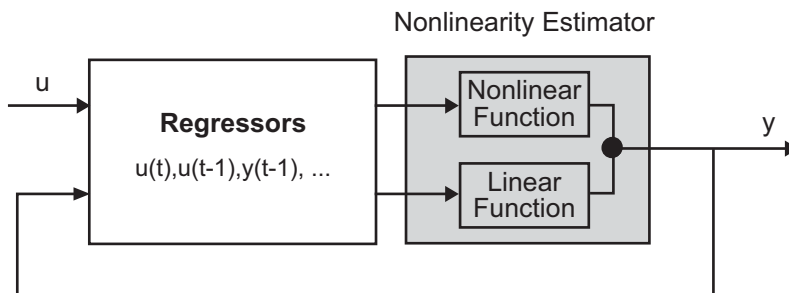
The following table summarizes the fields of the EstimationInfo model properties. The read-only fields of the EstimationInfo structure store estimation settings and results.

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.
FPE	Value of Akaike's Final Prediction Error (see <code>fpe</code> ).
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: <ul style="list-style-type: none"> <li>'zoh' means zero-order-hold, or piecewise constant.</li> <li>'foh' means first-order-hold, or piecewise linear.</li> </ul>
EstimationTime	Duration of the estimation.
InitRandState	The value of <code>randn('state')</code> at the last randomization of the initial parameter vector.
Iterations	Number of iterations performed by the estimation algorithm.
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.
Warning	Any warnings encountered during parameter estimation.
WhyStop	Reason for terminating parameter estimation iterations.

## Definitions

### Nonlinear ARX Model Structure

This block diagram represents the structure of a nonlinear ARX model in a simulation scenario:



The nonlinear ARX model computes the output  $y$  in two stages:

- 1 Computes regressors from the current and past input values and past output data.

In the simplest case, regressors are delayed inputs and outputs, such as  $u(t-1)$  and  $y(t-3)$ —called *standard* regressors. You can also specify *custom* regressors, which are nonlinear functions of delayed inputs and outputs. For example,  $\tan(u(t-1))$  or  $u(t-1)*y(t-3)$ .

By default, all regressors are inputs to both the linear and the nonlinear function blocks of the nonlinearity estimator. You can choose a subset of regressors as inputs to the nonlinear function block.

- 2 The nonlinearity estimator block maps the regressors to the model output using a combination of nonlinear and linear functions. You can select from available nonlinearity estimators, such as tree-partition networks, wavelet networks, and multi-layer neural networks. You can also exclude either the linear or the nonlinear function block from the nonlinearity estimator.

The nonlinearity estimator block can include linear and nonlinear blocks in parallel. For example:

$$F(x) = L^T(x - r) + d + g(Q(x - r))$$

$x$  is a vector of the regressors.  $L^T(x) + d$  is the output of the linear function block and is affine when  $d \neq 0$ .  $d$  is a scalar offset.  $g(Q(x - r))$  represents the output of the nonlinear function block.  $r$  is the mean of the regressors  $x$ .  $Q$  is a projection matrix that makes the calculations well conditioned. The exact form of  $F(x)$  depends on your choice of the nonlinearity estimator.

Estimating a nonlinear ARX model computes the model parameter values, such as  $L$ ,  $r$ ,  $d$ ,  $Q$ , and other parameters specifying  $g$ . Resulting models are `idnlarx` objects that store all model data, including model regressors and parameters of the nonlinearity estimator. See the `idnlarx` reference page for more information.

## Definition of `idnlarx` States

The states of an `idnlarx` object are delayed input and output variables that define the structure of the model. This toolbox requires states for simulation and prediction using `sim(idnlarx)`, `predict`, and `compare`. States are also necessary for linearization of nonlinear ARX models using `linearize(idnlarx)`.

This toolbox provides a number of options to facilitate how you specify the initial states. For example, you can use `findstates` and `data2state` to automatically search for state values in simulation and prediction applications. For linearization, use `findop`. You can also specify the states manually.

The states of an `idnlarx` model are defined by 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 at time  $t$ :  $p(t-1)$ ,  $p(t-2)$ , ...,  $p(t-D)$ .

For example, if you have a single-input, single-output `idnlarx` model:

```
m = idnlarx([2 3 0], 'wavenet', ...
            'CustomRegressors', ...
            {'y1(t-10)*u1(t-1)'});
```

This model has these regressors:

```
getreg(m)
```

Regressors:

```

y1(t-1)
y1(t-2)
u1(t)
u1(t-1)
u1(t-2)
y1(t-10)*u1(t-1)

```

The regressors show that the maximum delay in the output variable  $y_1$  is 10 samples and the maximum delay in the input  $u_1$  is 2 samples. Thus, this model has a total of 12 states:

$$X(t) = [y_1(t-1), y_2(t-2), \dots, y_1(t-10), u_1(t-1), u_1(t-2)]$$

---

**Note** The state vector includes the output variables first, followed by input variables.

---

As another example, consider the 2-output and 3-input model:

```
m = idnlarx([2 0 2 2 1 1 0 0; 1 0 1 5 0 1 1 0], ...
            [wavenet; linear])
```

getreg lists these 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  $y_1$  is 2 samples, which occurs in regressor set for output 1. The maximum delays in the three input variables are 2, 5, and 0, respectively. Thus, the state vector is:

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

Variables  $y_2$  and  $u_3$  do not contribute to the state vector because the maximum delay in these variables is zero.

A simpler way to determine states by inspecting regressors is to use `getDelayInfo`, which 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 input and output variables in the order ( $y_1, y_2, u_1, u_2, u_3$ ). The total number of model states is `sum(maxDel) = 9`.

The set of states for an `idnlarx` model are not required to be minimal.

## Examples

Create nonlinear ARX model structure with (default) wavelet network nonlinearity:

```
m = idnlarx([2 2 1]) % na=nb=2 and nk=1
```

---

Create nonlinear ARX model structure with sigmoid network nonlinearity:

```
m=idnlarx([2 3 1],sigmoidnet('Num',15))  
% number of units is 15
```

---

Create nonlinear ARX model structure with no nonlinear function in nonlinearity estimator:

```
m=idnlarx([2 2 1],[,])
```

---

Construct a nonlinear ARX model using a linear ARX model:

```
% Construct a linear ARX model.  
A = [1 -1.2 0.5];  
B = [0.8 1];  
LinearModel = idpoly(A, B, 'Ts', 0.1);
```

```
% Construct nonlinear ARX model using the linear ARX model.  
m1 = idnlarx(LinearModel)
```

## See Also

[addreg](#) | [customnet](#) | [customreg](#) | [findop\(idnlarx\)](#) | [getreg](#) | [idnlmodel](#) | [linear](#) | [linearize\(idnlarx\)](#) | [nlarx](#) | [pem](#) | [polyreg](#) | [sigmoidnet](#) | [wavenet](#)

## Tutorials

- “Example – Using nlarx to Estimate Nonlinear ARX Models”
- “Estimate Nonlinear ARX Models Using Linear ARX Models”

## **How To**

- “Identifying Nonlinear ARX Models”
- “Using Linear Model for Nonlinear ARX Estimation”



**Purpose**

Nonlinear ODE (grey-box model) with unknown parameters

**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 2-359
- “idnlgrey Properties” on page 2-360
- “idnlgrey Algorithm Properties” on page 2-364
- “idnlgrey Advanced Algorithm Properties” on page 2-367
- “idnlgrey Simulation Options” on page 2-369
- “idnlgrey Gradient Options” on page 2-372
- “idnlgrey EstimationInfo Properties” on page 2-373

**idnlgrey  
Constructor**

After you create the function or MEX-file with your model structure, you must define an idnlgrey object.

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 function or MEX-file storing the model structure (ODE file).
- `Order` — Vector with three entries  $[N_y \ N_u \ N_x]$ , specifying the number of model outputs  $N_y$ , the number of inputs  $N_u$ , and the number of states  $N_x$ .
- `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 unknown parameters and initial states of this object using `pem`. The input-output dimensions of the data must be compatible with the input and output orders you specified for the `idnlgrey` model. You can pass additional property-value pairs to `pem` to specify the properties of the model or estimation algorithm, such as `MaxIter` and `Tolerance`.

## idnlgrey Properties

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

You can include property-value pairs in the model estimator or constructor to specify the model structure and estimation algorithm properties.

Use `set` or dot notation to set a property of an existing object.

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 idn1model 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 2-364.
CovarianceMatrix	Covariance matrix of the estimated Parameters. Assignable values: <ul style="list-style-type: none"> <li>• 'None' to omit computing uncertainties and save time during parameter estimation.</li> <li>• 'Estimate' to estimation covariance. Symmetric and positive <math>N_p</math>-by-<math>N_p</math> matrix (or []) where <math>N_p</math> is the number of free model parameters.</li> </ul>
EstimationInfo	A read-only structure that stores estimation settings and results, as described in “idnlgrey EstimationInfo Properties” on page 2-373.
FileArgument	Contains auxiliary variables passed to the ODE file (function 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 a MATLAB file, P-code file or MEX-file. For more information about the file variables, see “Specifying the Nonlinear Grey-Box Model Structure”.

Property Name	Description
InitialStates	<p>An Nx-by-1 structure array with fields as follows. Here, Nx is the number of states of the model.</p> <ul style="list-style-type: none"> <li>• <b>Name:</b> Name of the state (a string). Default value is 'x#i', where #i is an integer in [1, Nx].</li> <li>• <b>Unit:</b> Unit of the state (a string). Default value is ''.</li> <li>• <b>Value:</b> Initial value of the initial state(s). Assignable values are: <ul style="list-style-type: none"> <li>▪ A finite real scalar</li> <li>▪ A finite real 1-by-Ne vector, where Ne is the number of experiments in the data set to be used for estimation</li> </ul> </li> <li>• <b>Minimum:</b> 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 &lt;= Value for all components. Default value: -Inf(size(Value)).</li> <li>• <b>Maximum:</b> 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 &lt;= Maximum for all components. Default value: Inf(size(Value)).</li> <li>• <b>Fixed:</b> 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).</li> </ul> <p>For an idnlgrey model M, the ith initial state is accessed through M.InitialStates(i) and its subfields as M.InitialStates(i).FIELDNAME.</p>

Property Name	Description
Order	<p>Structure with following fields:</p> <ul style="list-style-type: none"> <li>• <code>ny</code> — Number of outputs of the model structure.</li> <li>• <code>nu</code> — Number of inputs of the model structure.</li> <li>• <code>nx</code> — Number of states of the model structure.</li> </ul> <p>For time series, <code>nu</code> is 0. For static model structures, <code>nx</code> is 0.</p>
Parameters	<p><code>Np</code>-by-1 structure array with information about the model parameters containing the following fields:</p> <ul style="list-style-type: none"> <li>• <b>Name:</b> Name of the parameter (a string). Default value is '<code>p#i</code>', where <code>#i</code> is an integer in <code>[1, Np]</code>.</li> <li>• <b>Unit:</b> Unit of the parameter (a string). Default value is ''.</li> <li>• <b>Value:</b> Initial value of the parameter(s). Assignable values are: <ul style="list-style-type: none"> <li>▪ A finite real scalar</li> <li>▪ A finite real column vector</li> <li>▪ A 2-dimensional real matrix</li> </ul> </li> <li>• <b>Minimum:</b> Minimum value of the parameter(s). Must be a real scalar/column vector/matrix of the same size as <code>Value</code> and such that <code>Minimum &lt;= Value</code> for all components. Default value: <code>-Inf(size(Value))</code>.</li> <li>• <b>Maximum:</b> Maximum value of the parameter(s). Must be a real scalar/column vector/matrix of the same size as <code>Value</code> and such that <code>Value &lt;= Maximum</code> for all components. Default value: <code>Inf(size(Value))</code>.</li> <li>• <b>Fixed:</b> 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 <code>Value</code>.</li> </ul>

Property Name	Description
	<p>Default value: <code>false(size(Value))</code>, (estimate all parameter components).</p> <p>For an <code>idnlgrey</code> model <code>M</code>, the <code>i</code>th parameter is accessed through <code>M.Parameters(i)</code> and its subfields as <code>M.Parameters(i).FIELDNAME</code>.</p>

## 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 2-367.
Criterion	<p>Specifies criterion used during minimization. Criterion can have the following values:</p> <ul style="list-style-type: none"> <li>• 'Det': Minimize <math>\det(E' * E)</math> where <math>E</math> 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 <code>idnlgrey</code> which uses 'Trace' by default.</li> <li>• 'Trace': Minimize the trace of the weighted prediction error matrix <math>\text{trace}(E' * E * W)</math>, where <math>E</math> is the matrix of prediction errors, with one column for each output, and <math>W</math> is a positive semi-definite symmetric matrix of size equal to the number of outputs. By default, <math>W</math> is an identity matrix of size equal to the number of model outputs (so the minimization criterion becomes <math>\text{trace}(E' * E)</math>, or the traditional least-sum-of-squared-errors criterion. You can</li> </ul>

Property Name	Description
	specify the relative weighting of prediction errors for each output using the <code>Weighting</code> field of the <code>Algorithm</code> 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).
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: <ul style="list-style-type: none"> <li>• 'Auto' — Automatically chooses from the following methods.</li> <li>• 'gn' — Gauss-Newton method.</li> <li>• 'gna' — Adaptive Gauss-Newton method.</li> <li>• 'grad' — A gradient method.</li> <li>• 'lm' — Levenberg-Marquardt method.</li> <li>• 'lsqnonlin' — Nonlinear least-squares method (requires the Optimization Toolbox product). This method handles only the 'Trace' criterion.</li> </ul>
Tolerance	Specifies to terminate the iterative search when the expected improvement of the parameter values is less than <code>Tolerance</code> , 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 2-372.

Property Name	Description
SimulationOptions	A structure that specifies the simulation method and related options, as described in “idnlgrey Simulation Options” on page 2-369.
Display	Toggles displaying or hiding estimation progress information in the MATLAB Command Window. Default: 'Off'. Assignable values: <ul style="list-style-type: none"><li>• 'Off' — Hide estimation information.</li><li>• 'On' — Display estimation information.</li></ul>
Weighting	Positive semi-definite matrix $W$ used for weighted trace minimization. When <code>Criterion = 'Trace'</code> , $\text{trace}(E'E*W)$ is minimized. <code>Weighting</code> 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. <code>Weighting</code> is not useful in single-output models. By default, <code>Weighting</code> 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 to 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 $GnPinvConst * \max(\text{size}(J)) * \text{norm}(J) * \text{eps}$ . 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 <i>regularization</i> 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 <i>regularization</i> 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: 25. 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	<p>The smallest parameter update allowed per iteration.            Default: 1e-16.            Assign a positive, real value.</p>
RelImprovement	<p>The iterations are stopped if the relative improvement of the criterion function is less than RelImprovement.            Default: 0.            Assign a positive real value.</p> <hr/> <p><b>Note</b> Does not apply to            Algorithm.SearchMethod='lsqnonlin'</p> <hr/>
StepReduction	<p>(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 &gt;1.</p> <hr/> <p><b>Note</b> Does not apply to            Algorithm.SearchMethod='lsqnonlin'</p> <hr/>

## idnlgrey Simulation Options

The following table summarizes the fields of Algorithm.SimulationOptions model properties.

Property Name	Description
AbsTol	<p>Absolute error tolerance. This scalar applies to all components of the state vector. AbsTol applies only to the variable step solvers.</p> <p>Default: 1e-6.</p> <p>Assignable value: A positive real value.</p>
FixedStep	<p>(For fixed-step time-continuous solvers) Step size used by the solver.</p> <p>Default: 'Auto'.</p> <p>Assignable values:</p> <ul style="list-style-type: none"> <li>• 'Auto' — Automatically chooses the initial step.</li> <li>• A real value such that <math>0 &lt; \text{FixedStep} \leq 1</math>.</li> </ul>
InitialStep	<p>(For variable-step time-continuous solvers) Specifies the initial step at which the ODE solver starts.</p> <p>Default: 'Auto'.</p> <p>Assignable values:</p> <ul style="list-style-type: none"> <li>• 'Auto' — Automatically chooses the initial step.</li> <li>• A positive real value such that <math>\text{MinStep} \leq \text{InitialStep} \leq \text{MaxStep}</math>.</li> </ul>
MaxOrder	<p>(For ode15s) Specifies the order of the Numerical Differentiation Formulas (NDF).</p> <p>Default: 5.</p> <p>Assignable values: 1, 2, 3, 4 or 5.</p>
MaxStep	<p>(For variable-step time-continuous solvers) Specifies the largest time step of the ODE solver.</p> <p>Default: 'Auto' — 1/15 of the simulation interval.</p> <p>Assignable values:</p> <ul style="list-style-type: none"> <li>• 'Auto' — Automatically chooses the time step.</li> <li>• A positive real value <math>&gt; \text{MinStep}</math>.</li> </ul>

Property Name	Description
MinStep	<p>(For variable-step time-continuous solvers) Specifies the smallest time step of the ODE solver.            Default: 'Auto'.            Assignable values:</p> <ul style="list-style-type: none"> <li>• 'Auto' — Automatically chooses the time step.</li> <li>• A positive real value &lt; MaxStep.</li> </ul>
RelTol	<p>(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 <math> e(i)  \leq \max(\text{RelTol} \cdot \text{abs}(x(i)), \text{AbsTol}(i))</math>.            Default: 1e-3 (0.1% accuracy).            Assignable value: A positive real value.</p>
Solver	<p>ODE (Ordinary Differential/Difference Equation) solver for solving state space equations.</p> <p>A. Variable-step solvers for time-continuous idnlgrey models:</p> <ul style="list-style-type: none"> <li>• 'ode45' — Runge-Kutta (4,5) solver for nonstiff problems.</li> <li>• 'ode23' — Runge-Kutta (2,3) solver for nonstiff problems.</li> <li>• 'ode113' — Adams-Bashforth-Moulton solver for nonstiff problems.</li> <li>• 'ode15s' — Numerical Differential Formula solver for stiff problems.</li> <li>• 'ode23s' — Modified Rosenbrock solver for stiff problems.</li> <li>• 'ode23t' — Trapezoidal solver for moderately stiff problems.</li> <li>• 'ode23tb' — Implicit Runge-Kutta solver for stiff problems.</li> </ul> <p>B. Fixed-step solvers for time-continuous idnlgrey models:</p> <ul style="list-style-type: none"> <li>• 'ode5' — Dormand-Prince solver.</li> </ul>

Property Name	Description
	<ul style="list-style-type: none"> <li>'ode4' — Fourth-order Runge-Kutta solver.</li> <li>'ode3' — Bogacki-Shampine solver.</li> <li>'ode2' — Heun or improved Euler solver.</li> <li>'ode1' — Euler solver.</li> </ul> <p>C. Fixed-step solvers for time-discrete idnlgrey models: 'FixedStepDiscrete'</p> <p>D. General: 'Auto' — Automatically chooses one of the previous solvers (default).</p>

## 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: $0.01 \cdot \sqrt{\text{eps}}$ . Assignable value: A positive real value <'DiffMaxChange'.

Property Name	Description
DiffScheme	<p>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:</p> <ul style="list-style-type: none"> <li>• 'Auto' - Automatically chooses from the following methods.</li> <li>• 'Central approximation'</li> <li>• 'Forward approximation'</li> <li>• 'Backward approximation'</li> </ul>
GradientType	<p>Method used when computing derivatives (Jacobian) of the parameters or the initial states to be estimated.            Default: 'Auto'.            Assignable values:</p> <ul style="list-style-type: none"> <li>• 'Auto' — Automatically chooses from the following methods.</li> <li>• 'Basic' — Individually computes all numerical derivatives required to form each column of the Jacobian.</li> <li>• 'Refined' — Simultaneously computes all numerical derivatives required to form each column of the Jacobian.</li> </ul>

## **idnlgrey EstimationInfo Properties**

The following table summarizes the fields of the EstimationInfo model properties. The read-only fields of the EstimationInfo structure store estimation settings and results.

Property Name	Description
Status	Shows whether the model parameters were estimated.
Method	Names of the solver and the optimizer used during estimation.
LossFcn	Value of the loss function, equal to $\det(E' * E / N)$ , where E is the residual error matrix (one column for each output) and N is the total number of samples. Provides a quantitative description of the model quality.
FPE	Value of Akaike's Final Prediction Error (see <code>fpe</code> ).
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: <ul style="list-style-type: none"> <li>'zoh' means zero-order-hold, or piecewise constant.</li> <li>'foh' means first-order-hold, or piecewise linear.</li> </ul>
EstimationTime	Duration of the estimation.
InitialGuess	Structure with the fields <code>InitialStates</code> and <code>Parameters</code> , specifying the values of these quantities before the last estimation.
Iterations	Number of iterations performed by the estimation algorithm.
LastImprovement	Criterion improvement in the last iteration, shown in %. Empty when <code>SearchMethod='lsqnonlin'</code> is the search method.



---

Property Name	Description
UpdateNorm	Norm of the search vector (Gauss-Newton vector) at the last iteration. Empty when 'lsqnonlin' is the search method.
Warning	Any warnings encountered during parameter estimation.
WhyStop	Reason for terminating parameter estimation iterations.

### Definition of idnlgrey States

The states of an idnlgrey model are defined explicitly by the user in the function 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 | get | set | getinit | setinit | getpar | idnlmodel | setpar

## Purpose

Hammerstein-Wiener model

## Syntax

```
m = idnlhw([nb nf nk])  
m = idnlhw([nb nf nk],InputNL,OutputNL)  
m = idnlhw([nb nf nk],InputNL,OutputNL, 'PropertyName',  
    PropertyValue)  
m = idnlhw(LinModel)  
m = idnlhw(LinModel,InputNL,OutputNL)  
m = idnlhw(LinModel,InputNL,OutputNL, 'PropertyName',  
    PropertyValue)
```

## Description

Represents Hammerstein-Wiener models. The Hammerstein-Wiener structure represents a linear model with input-output nonlinearities.

Typically, you use the `n1hw` command to both construct the `idnlhw` object and estimate the model parameters. You can configure the model properties directly in the `n1hw` syntax. For information about the Hammerstein-Wiener model structure, see “Structure of Hammerstein-Wiener Models”.

You can also use the `idnlhw` constructor to create the Hammerstein-Wiener model structure and then estimate the parameters of this model using `pem`.

For `idnlhw` object properties, see:

- “`idnlhw` Model Properties” on page 2-379
- “`idnlhw` Algorithm Properties” on page 2-380
- “`idnlhw` Advanced Algorithm Properties” on page 2-384
- “`idnlhw` EstimationInfo Properties” on page 2-386

## Construction

`m = idnlhw([nb nf nk])` creates an `idnlhw` object using default piecewise linear functions for the input and output nonlinearity estimators. *nb*, *nf*, and *nk* are positive integers that specify model orders and delays. *nb* is the number of zeros plus 1, *nf* is the number of poles, and *nk* is the input delay.

$m = \text{idnlhw}([nb \ nf \ nk], \text{InputNL}, \text{OutputNL})$  specifies input nonlinearity *InputNL* and output nonlinearity *OutputNL*, as a nonlinearity estimator object or string representing the nonlinearity estimator type.

$m = \text{idnlhw}([nb \ nf \ nk], \text{InputNL}, \text{OutputNL}, \text{'PropertyName'}, \text{PropertyValue})$  creates the object using options specified as *idnlhw* property name and value pairs. Specify *PropertyName* inside single quotes.

$m = \text{idnlhw}(\text{LinModel})$  uses a linear model (in place of  $[nb \ nf \ nk]$ ) and default piecewise linear functions for the input and output nonlinearity estimators. *LinModel* is a discrete time input-output polynomial model of Output-Error (OE) structure (*idpoly*) or state-space model with no disturbance component (*idss* with  $K = 0$ ) for single-output systems, and *idss* model with  $K = 0$  for multi-output systems. *LinModel* sets the model orders, input delay, *B* and *F* polynomial values, input-output names and units, sampling time, and time units of *m*.

$m = \text{idnlhw}(\text{LinModel}, \text{InputNL}, \text{OutputNL})$  specifies input nonlinearity *InputNL* and output nonlinearity *OutputNL*.

$m = \text{idnlhw}(\text{LinModel}, \text{InputNL}, \text{OutputNL}, \text{'PropertyName'}, \text{PropertyValue})$  creates the object using options specified as *idnlhw* property name and value pairs.

## Input Arguments

### **nb, nf, nk**

Model orders and input delay, where *nb* is the number of zeros plus 1, *nf* is the number of poles, and *nk* is the input delay.

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, OutputNL**

Input and output nonlinearity estimators, respectively, specified as a nonlinearity estimator object or string representing the nonlinearity estimator type.

'pwnlinear' or pwnlinear object (default)	Piecewise linear function
'sigmoidnet' or sigmoidnet object	Sigmoid network
'wavenet' or wavenet object	Wavelet network
'saturation' or saturation object	Saturation
'deadzone' or deadzone object	Dead zone
'poly1d' or poly1d object	One-dimensional polynomial
'unitgain' or unitgain object	Unit gain
customnet object	Custom network

Specifying a string creates a nonlinearity estimator object with default settings. Use object representation to configure the properties of a nonlinearity estimator.

For  $n_y$  output channels, you can specify nonlinear estimators individually for each output channel by setting *InputNL* or *OutputNL* to an  $n_y$ -by-1 cell array or object array of nonlinearity estimators. To specify the same nonlinearity for all outputs, specify a single input and output nonlinearity estimator.

## LinModel

Discrete time linear model, typically estimated using the `oe` or `n4sid` command:

- Input-output polynomial model of Output-Error (OE) structure (`idpoly`) or state-space model with no disturbance component (`idss` with  $K = 0$ ), for single-output systems
- State-space model with no disturbance component (`idss` model with  $K = 0$ ), for multi-output systems

## idnlhw Model 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
```

You can specify property name-value pairs in the model estimator or constructor to specify the model structure and estimation algorithm.

Use `set` or dot notation to set a property of an existing object.

The following table summarizes `idnlhw` model properties. The general `idnlmodel` properties also apply to this nonlinear model object (see the corresponding reference page).

Property Name	Description
Algorithm	A structure that specifies the estimation algorithm options, as described in “ <code>idnlhw</code> Algorithm Properties” on page 2-380.
<code>b</code>	$B$ polynomial as a cell array of $N_y$ -by- $N_u$ elements, where $N_y$ is the number of outputs and $N_u$ is the number of inputs. An element <code>b{i,j}</code> 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.
<code>f</code>	$F$ polynomial as a cell array of $N_y$ -by- $N_u$ elements, where $N_y$ is the number of outputs and $N_u$ is the number of inputs. An element <code>f{i,j}</code> 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 in the linear block. For single output, represented as an <code>idpoly</code> object. For multiple output, represented as an <code>idss</code> object.

Property Name	Description
EstimationInfo	(Read-only) Structure that stores estimation settings and results, as described in “idnlhw EstimationInfo Properties” on page 2-386.
InputNonlinearity	<p>Nonlinearity estimator object. Assignable values include <code>pwlinear</code> (default), <code>deadzone</code>, <code>wavenet</code>, <code>saturation</code>, <code>customnet</code>, <code>sigmoidnet</code>, <code>poly1d</code>, and <code>unitgain</code>. For more information, see the corresponding reference pages.</p> <p>For <code>ny</code> outputs, <code>Nonlinearity</code> is an <code>ny</code>-by-1 array, such as <code>[sigmoidnet;wavenet]</code>. However, if you specify a scalar object, this nonlinearity object applies to all outputs.</p>
OutputNonlinearity	Same as <code>InputNonlinearity</code> .
<code>nb</code> <code>nf</code> <code>nk</code>	<p>Model orders and input delays, where <code>nb</code> is the number of zeros plus 1, <code>nf</code> is the number of poles, and <code>nk</code> is the delay from input to output in terms of the number of samples.</p> <p>For <code>nu</code> inputs and <code>ny</code> outputs, <code>nb</code>, <code>nf</code> and, <code>nk</code> are <code>ny</code>-by-<code>nu</code> matrices whose <math>i</math>-<math>j</math>th entry specifies the orders and delay of the transfer function from the <math>j</math>th input to the <math>i</math>th output.</p>

## 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 2-384.
Criterion	<p>The search method of <code>lsqnonlin</code> supports the Trace criterion only.</p> <p>Use for multiple-output models only. Criterion can have the following values:</p> <ul style="list-style-type: none"> <li>• 'Det': Minimize <math>\det(E' * E)</math>, 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 <code>idnlgrey</code> which uses 'Trace' by default.</li> <li>• 'Trace': Minimize the trace of the weighted prediction error matrix <math>\text{trace}(E' * E * W)</math>, 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 <math>\text{trace}(E' * E)</math>, 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 <code>neuralnet</code> or <code>treepartition</code> as one of its nonlinearity estimators, weighting is not applied because estimations are independent for each output.</li> </ul> <p>Both the Det and Trace criteria are derived from a general requirement of minimizing a weighted sum of least squares of prediction errors. Det 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.</p>

Property Name	Description
	If you want to achieve better accuracy for a particular channel in MIMO models, use <code>Trace</code> with weighting that favors that channel. Otherwise, use <code>Det</code> . If you use <code>Det</code> , check <code>cond(model.NoiseVariance)</code> after estimation. If the matrix is ill-conditioned, try using the <code>Trace</code> criterion. You can also use <code>compare</code> on validation data to check whether the relative error for different channels corresponds to your needs or expectations. Use the <code>Trace</code> criterion if you need to modify the relative errors, and check <code>model.NoiseVariance</code> to determine what weighting modifications to specify.
IterWavenet	(For wavenet nonlinear estimator only) Implicitly set to perform iterative estimation. Changing this setting does not impact the algorithm. Default: 'On'.
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.



Property Name	Description
MaxSize	<p>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.</p> <hr/> <p><b>Note</b> The original data matrix of <math>u</math> and <math>y</math> must be smaller than MaxSize.</p>
SearchMethod	<p>Method used by the iterative search algorithm. Assignable values:</p> <ul style="list-style-type: none"> <li>• 'Auto' — Automatically chooses from the following methods.</li> <li>• 'gn' — Gauss-Newton method.</li> <li>• 'gna' — Adaptive Gauss-Newton method.</li> <li>• 'grad' — A gradient method.</li> <li>• 'lm' — Levenberg-Marquardt method.</li> <li>• 'lsqnonlin' — Nonlinear least-squares method (requires the Optimization Toolbox product). This method handles only the 'Trace' criterion.</li> </ul>
Tolerance	<p>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 %.</p> <p>Default: 0.01.</p>

Property Name	Description
Display	<p>Toggles displaying or hiding estimation progress information in the MATLAB Command Window.                      Default: 'Off'.                      Assignable values:</p> <ul style="list-style-type: none"> <li>'Off' — Hide estimation information.</li> <li>'On' — Display estimation information.</li> </ul>
Weighting	<p>Positive semi-definite matrix <math>W</math> used for weighted trace minimization. When <code>Criterion = 'Trace'</code>, <math>\text{trace}(E' * E * W)</math> is minimized. <code>Weighting</code> can be used to specify relative importance of outputs in multiple-input multiple-output models (or reliability of corresponding data) when <math>W</math> is a diagonal matrix of nonnegative values. <code>Weighting</code> is not useful in single-output models. By default, <code>Weighting</code> is an identity matrix of size equal to the number of outputs.</p>

## 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	<p>When the search direction is computed, the algorithm discards the singular values of the Jacobian that are smaller than <math>\text{GnPinvConst} * \max(\text{size}(J)) * \text{norm}(J) * \text{eps}</math>. Singular values that are closer to 0 are included when <code>GnPinvConst</code> is decreased.                      Default: <code>1e4</code>.                      Assign a positive, real value.</p>
LMStartValue	<p>(For Levenberg-Marquardt search algorithm) The starting level of <i>regularization</i> when using the Levenberg-Marquardt search method (<code>Algorithm.SearchMethod = 'lm'</code>).</p>

Property Name	Description
	Default: 0.001. Assign a positive real value.
LMStep	(For Levenberg-Marquardt search algorithm) Try this next level of <i>regularization</i> 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	<p>The iterations are stopped if the relative improvement of the criterion function is less than RelImprovement. Default: 0. Assign a positive real value.</p> <hr/> <p><b>Note</b> This does not apply when Algorithm.SearchMethod='lsqnonlin'.</p>
StepReduction	<p>(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 &gt;1.</p> <hr/> <p><b>Note</b> This does not apply when Algorithm.SearchMethod='lsqnonlin'.</p>

## idnlhw EstimationInfo Properties

The following table summarizes the fields of the EstimationInfo model properties. The read-only fields of the EstimationInfo structure store estimation settings and results.

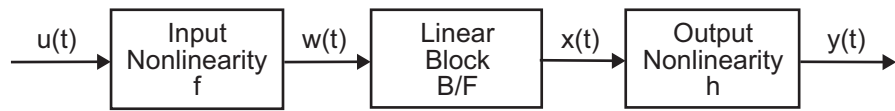
Property Name	Description
Status	Shows whether the model parameters were estimated.
Method	Shows the estimation method.
LossFcn	Value of the loss function, equal to $\det(E' * E / N)$ , where E is the residual error matrix (one column for each output) and N is the total number of samples.

Property Name	Description
FPE	Value of Akaike's Final Prediction Error (see <code>fpe</code> ).
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: <ul style="list-style-type: none"> <li>'zoh' means zero-order-hold, or piecewise constant.</li> <li>'foh' means first-order-hold, or piecewise linear.</li> </ul>
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 random number type and seed at the last randomization of the initial parameter vector.
EstimationTime	Duration of the estimation.

## Definitions

### Hammerstein-Wiener Model Structure

This block diagram represents the structure of a Hammerstein-Wiener model:



where:

- $w(t) = f(u(t))$  is a nonlinear function transforming input data  $u(t)$ .  $w(t)$  has the same dimension as  $u(t)$ .
- $x(t) = (B/F)w(t)$  is a linear transfer function.  $x(t)$  has the same dimension as  $y(t)$ .

where  $B$  and  $F$  are similar to polynomials in the linear Output-Error model, as described in “What Are Polynomial Models?”.

For  $n_y$  outputs and  $n_u$  inputs, the linear block is a transfer function matrix containing entries:

$$\frac{B_{j,i}(q)}{F_{j,i}(q)}$$

where  $j = 1, 2, \dots, n_y$  and  $i = 1, 2, \dots, n_u$ .

- $y(t) = h(x(t))$  is a nonlinear function that maps the output of the linear block to the system output.

$w(t)$  and  $x(t)$  are internal variables that define the input and output of the linear block, respectively.

Because  $f$  acts on the input port of the linear block, this function is called the *input nonlinearity*. Similarly, because  $h$  acts on the output port of the linear block, this function is called the *output nonlinearity*. If system contains several inputs and outputs, you must define the functions  $f$  and  $h$  for each input and output signal.

You do not have to include both the input and the output nonlinearity in the model structure. When a model contains only the input nonlinearity  $f$ , it is called a *Hammerstein* model. Similarly, when the model contains only the output nonlinearity  $h$ , it is called a *Wiener* model.

The nonlinearities  $f$  and  $h$  are scalar functions, one nonlinear function for each input and output channel.

The Hammerstein-Wiener model computes the output  $y$  in three stages:

- 1 Computes  $w(t) = f(u(t))$  from the input data.

$w(t)$  is an input to the linear transfer function  $B/F$ .

The input nonlinearity is a static (*memoryless*) function, where the value of the output a given time  $t$  depends only on the input value at time  $t$ .

You can configure the input nonlinearity as a sigmoid network, wavelet network, saturation, dead zone, piecewise linear function, one-dimensional polynomial, or a custom network. You can also remove the input nonlinearity.

- 2 Computes the output of the linear block using  $w(t)$  and initial conditions:  $x(t) = (B/F)w(t)$ .

You can configure the linear block by specifying the numerator  $B$  and denominator  $F$  orders.

- 3 Compute the model output by transforming the output of the linear block  $x(t)$  using the nonlinear function  $h$ :  $y(t) = h(x(t))$ .

Similar to the input nonlinearity, the output nonlinearity is a static function. Configure the output nonlinearity in the same way as the input nonlinearity. You can also remove the output nonlinearity, such that  $y(t) = x(t)$ .

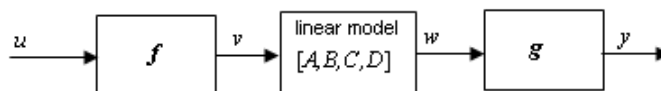
Resulting models are `idnlhw` objects that store all model data, including model parameters and nonlinearity estimator. See the `idnlhw` reference page for more information.

### **idnlhw States**

This toolbox requires states for simulation and prediction using `sim(idnlhw)`, `predict`, and `compare`. States are also necessary for

linearization of nonlinear ARX models using `linearize(idnlhw)`. This toolbox provides a number of options to facilitate how you specify the initial states. For example, you can use `findstates` and `data2state` to automatically search for state values in simulation and prediction applications. For linearization, use `findop`. You can also specify the states manually.

The states of the Hammerstein-Wiener model correspond to the states of the linear block in the Hammerstein-Wiener model structure:



The linear block contains all the dynamic elements of the model. If this linear model is not a state-space structure, the states are defined as those of model `Mss`, where `Mss = idss(Model.LinearModel)` and `Model` is the `idnlhw` object.

## Examples

Create default Hammerstein-Wiener model structure:

```
m = idnlhw([2 2 1]) % na=nb=2 and nk=1
% m has piecewise linear input and output nonlinearity
```

---

Create nonlinear ARX model structure with sigmoid network nonlinearity:

```
m=idnlarx([2 3 1],sigmoidnet('Num',15))
% number of units is 15
```

---

Create Hammerstein-Wiener model with specific input-output nonlinearities:

```
m=idnlhw([2 2 1],'sigmoidnet','deadzone')
% Equivalent to m=idnlhw([2 2 1],'sig','dead')
% Nonlinearities have default configuration
```



---

Create Hammerstein-Wiener model and configure the nonlinearity objects:

```
m=idnlhw([2 2 1],sigmoidnet('num',5),deadzone([-1,2]))
```

---

Create a Hammerstein model (no output nonlinearity):

```
m=idnlhw([2 2 1],'saturation',[])  
% [] specifies unitgain output nonlinearity
```

---

Configure the Hammerstein-Wiener model and estimate models parameters:

```
m0 = idnlhw([nb,nf,nk],[sigmoidnet;pwlinear],[]);  
m = pem(data,m0); % equivalent to m=nlhw(data,m0)
```

---

Construct default Hammerstein-Wiener model using an input-output polynomial model of Output-Error structure:

```
% Construct an input-output polynomial model of OE structure.  
B = [0.8 1];  
F = [1 -1.2 0.5];  
LinearModel = idpoly(1, B, 1,1, F, 'Ts', 0.1);  
  
% Construct Hammerstein-Wiener model using OE model  
% as its linear component.  
m1 = idnlhw(LinearModel, 'saturation', [])
```

## See Also

customnet | idnlmodel | linear | linearize(idnlhw) | nlhw | pem |  
poly1d | saturation | sigmoidnet | wavenet | saturation

## **Tutorials**

- “Example – Using nlhw to Estimate Hammerstein-Wiener Models”
- “Estimate Hammerstein-Wiener Models Using Linear OE Models”

## **How To**

- “Identifying Hammerstein-Wiener Models”
- “Using Linear Model for Hammerstein-Wiener Estimation”

**Purpose** Superclass for nonlinear models

**Description** You do not use the `idn1model` class directly. Instead, `idn1model` defines the common properties and methods inherited by its subclasses, `idn1larx`, `idn1lgrey`, and `idn1lhw`.

**idn1model Properties** The following table lists the properties shared by the `idn1larx`, `idn1lgrey`, and `idn1lhw`, defined in terms of  $N_y$  outputs and  $N_u$  inputs.

Property Name	Description
InputName	<p>Specifies the names of individual input channels. Default: {'u1'; 'u2'; ...; 'uNu'}.</p> <p>Assignable values:</p> <ul style="list-style-type: none"> <li>• For single-output models, a string. For example, 'torque'.</li> <li>• For multiple-output models, an <math>n_u</math>-by-1 cell array. For example: {'thrust'; 'aileron deflection'}</li> </ul>
InputUnit	<p>Specifies the units of each input channel. Default: ''.</p> <p>Assignable values:</p> <ul style="list-style-type: none"> <li>• For single-output models, a string. For example, 'm/s'.</li> <li>• For multiple-output models, an <math>n_u</math>-by-1 cell array.</li> </ul>
Name	Name of the model, specified as a string.
NoiseVariance	<p>Noise variance (covariance matrix) of the model innovations <math>e</math>. Assignable value is an <math>n_y</math>-by-<math>n_y</math> matrix. Typically set automatically by the estimation algorithm.</p>

# idnImodel

Property Name	Description
OutputName	<p>Specifies the names of individual output channels. Default: {'y1'; 'y2'; ...; 'yNy'}.</p> <p>Assignable values:</p> <ul style="list-style-type: none"><li>• For single-output models, a string. For example, 'torque'.</li><li>• For multiple-output models, an ny-by-1 cell array. For example: {'thrust'; 'aileron deflection'}</li></ul>
OutputUnit	<p>Specifies the units of each output channel. Default: ''.</p> <p>Assignable values:</p> <ul style="list-style-type: none"><li>• For single-output models, a string. For example, 'm/s'.</li><li>• For multiple-output models, an ny-by-1 cell array.</li></ul>
TimeUnit	<p>Unit of the sampling interval and time vector, specified as one of the following:</p> <ul style="list-style-type: none"><li>• 'nanoseconds'</li><li>• 'microseconds'</li><li>• 'milliseconds'</li><li>• 'seconds'</li><li>• 'minutes'</li><li>• 'hours'</li><li>• 'days'</li><li>• 'weeks'</li><li>• 'months'</li><li>• 'years'</li></ul> <p>Default: 'seconds'.</p>

Property Name	Description
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: <ul style="list-style-type: none"><li>• For discrete-time models, positive scalar value of the sampling interval.</li><li>• For continuous-time models, 0(idnlgrey models only).</li></ul>

### See Also

idnlarx

idnlgrey

idnlhw

# idpar

---

**Purpose** Create parameter for initial states and input level estimation

**Syntax**

```
p = idpar(paramvalue)
p = idpar(paramname,paramvalue)
```

**Description** `p = idpar(paramvalue)` creates an estimable parameter with initial value `paramvalue`. The parameter, `p`, is either scalar or array-valued, with the same dimensions as `paramvalue`. You can configure attributes of the parameter, such as which elements are fixed and which are estimated, and lower and upper bounds.

`p = idpar(paramname,paramvalue)` sets the Name property of `p` to the string `paramname`.

**Tips** Use `idpar` to create estimable parameters for:

- Initial state estimation for state-space model estimation (`ssest`), prediction (`predict`), and forecasting (`forecast`)
- Explicit initial state estimation with `findstates`
- Input level estimation for process model estimation with `pem`

Specifying estimable state values or input levels gives you explicit control over the behavior of individual state values during estimation.

## Input Arguments

### **paramvalue**

Initial parameter value.

`paramvalue` is a numeric scalar or array that determines both the dimensions and initial values of the estimable parameter `p`. For example, `p = idpar(eye(3))` creates a 3-by-3 parameter whose initial value is the identity matrix.

`paramvalue` should be:

- A column vector of length  $N_x$ , the number of states to estimate, if you are using `p` for initial state estimation.

- An  $N_x$ -by- $N_e$  array, if you are using `p` for initial state estimation with multi-experiment data.  $N_e$  is the number of experiments.
- A column vector of length  $N_u$ , the number of inputs to estimate, if you are using `p` for input level estimation.
- An  $N_u$ -by- $N_e$  array, if you are using `p` for input level estimation with multi-experiment data.

If the initial value of a parameter is unknown, use NaN.

### **paramname**

String specifying the Name property of `p`.

The Name property is not used in state estimation or input level estimation. You can optionally assign a name for convenience. For example, you can assign `x0` as the name of a parameter created for initial state estimation.

**Default:** 'par'

## **Output Arguments**

### **p**

Estimable parameter, specified as a `param.Continuous` object.

`p` can be either scalar- or array-valued. `p` takes its dimensions and initial value from `paramvalue`.

`p` contains the following fields:

- **Value** — Scalar or array value of the parameter.

The dimension and initial value of `p.Value` are taken from `paramvalue` when `p` is created.

- **Minimum** — Lower bound for the parameter value. When you use `p` in state estimation or input value estimation, the estimated value of the parameter does not drop below `p.Minimum`.

The dimensions of `p.Minimum` must match the dimensions of `p.Value`.

For array-valued parameters, you can:

- Specify lower bounds on individual array elements. For example, `p.Minimum([1 4]) = -5`.
- Use scalar expansion to set the lower bound for all array elements. For example, `p.Minimum = -5`

**Default:** `-Inf`

- **Maximum** — Upper bound for the parameter value. When you use `p` in state estimation or input value estimation, the estimated value of the parameter does not exceed `p.Maximum`.

The dimensions of `p.Maximum` must match the dimensions of `p.Value`.

For array-valued parameters, you can:

- Specify upper bounds on individual array elements. For example, `p.Maximum([1 4]) = 5`.
- Use scalar expansion to set the upper bound for all array elements. For example, `p.Maximum = 5`

**Default:** `Inf`

- **Free** — Boolean specifying whether the parameter is a free estimation variable.

The dimensions of `p.Free` must match the dimensions of `p.Value`. By default, all values are free (`p.Free = true`).

If you want to estimate `p.Value(k)`, set `p.Free(k) = true`. To fix `p.Value(k)`, set `p.Free(k) = false`. Doing so allows you to control which states or input values are estimated and which are not.

For array-valued parameters, you can:

- Fix individual array elements. For example, `p.Free([1 4]) = false; p.Free = [1 0; 0 1]`.
- Use scalar expansion to fix all array elements. For example, `p.Free = false`.

**Default:** `true (1)`

- **Scale** — Scaling factor for normalizing the parameter value.



`p.Scale` is not used in initial state estimation or input value estimation.

**Default:** 1

- `Info` — Structure array for storing parameter units and labels. The structure has `Label` and `Unit` fields.

Use these fields for your convenience, to store strings that describe parameter units and labels.

For example, `p.Info(1,1).Unit = 'rad/m';`  
`p.Info(1,1).Label = 'engine speed'.`

The dimensions of `p.Info` must match the dimensions of `p.Value`.

**Default:** '' for both `Label` and `Unit` fields

- `Name` — Parameter name.

This property is read-only. It is set to the `paramname` input argument when you create the parameter.

**Default:** ''

## Examples

### Create and Configure Parameter for State Estimation

Create and configure a parameter for estimating the initial state values of a 4-state system. Fix the first state value to 1. Limit the second and third states to values between 0 and 1.

```
paramvalue = [1; nan(3,1)];
p = idpar('x0',paramvalue);
p.Free(1) = 0;
p.Minimum([2 3]) = 0;
p.Maximum([2 3]) = 1;
```

The column vector `paramvalue` specifies an initial value of 1 for the first state. `paramvalue` further specifies unknown values for the remaining 3 states.

Setting `p.Free(1)` to false fixes `p.Value(1)` to 1. Estimation using `p` does not alter that value.

Setting `p.Minimum` and `p.Maximum` for the second and third entries in `p` limits the range that those values can take when `p` is used in estimation.

You can now use `p` in initial state estimation, such as with the `findstates` command. For example, use `opt = findstatesOptions('InitialState',p)` to create a `findstates` options set that uses `p`. Then, call `findstates` with that options set.

## See Also

`predict` | `findstates(idParametric)` | `findstatesOptions` | `forecast` | `ssest` | `pem`

**Purpose**

Polynomial model with identifiable parameters

**Syntax**

```

sys = idpoly(A,B,C,D,F,NoiseVariance,Ts)
sys = idpoly(A,B,C,D,F,NoiseVariance,Ts,Name,Value)

sys = idpoly(A)
sys = idpoly(A,[],C,D,[],NoiseVariance,Ts)
sys = idpoly(A,[],C,D,[],NoiseVariance,Ts,Name,Value)

sys = idpoly(sys0)
sys = idpoly(sys0,'split')

```

**Description**

`sys = idpoly(A,B,C,D,F,NoiseVariance,Ts)` creates a polynomial model with identifiable coefficients. `A`, `B`, `C`, `D`, and `F` specify the initial values of the coefficients. `NoiseVariance` specifies the initial value of the variance of the white noise source. `Ts` is the model sampling time.

`sys = idpoly(A,B,C,D,F,NoiseVariance,Ts,Name,Value)` creates a polynomial model using additional options specified by one or more `Name,Value` pair arguments.

`sys = idpoly(A)` creates a time series model with only an autoregressive term. In this case, `sys` represents the AR model given by  $A(q^{-1})y(t) = e(t)$ . The noise  $e(t)$  has variance 1. `A` specifies the initial values of the estimable coefficients.

`sys = idpoly(A,[],C,D,[],NoiseVariance,Ts)` creates a time series model with an autoregressive and a moving average term. The inputs `A`, `C`, and `D`, specify the initial values of the estimable coefficients. `NoiseVariance` specifies the initial value of the noise  $e(t)$ . `Ts` is the model sampling time. (Omit `NoiseVariance` and `Ts` to use their default values.)

If `D = []`, then `sys` represents the ARMA model given by:

$$A(q^{-1})y(t) = C(q^{-1})e(t).$$

`sys = idpoly(A, [], C, D, [], NoiseVariance, Ts, Name, Value)` creates a time series model using additional options specified by one or more `Name, Value` pair arguments.

`sys = idpoly(sys0)` converts any dynamic system model, `sys0`, to `idpoly` model form.

`sys = idpoly(sys0, 'split')` converts `sys0` to `idpoly` model form, and treats the last  $N_y$  input channels of `sys0` as noise channels in the returned model. `sys0` must be a numeric (non-identified) `tf`, `zpk`, or `ss` model object. Also, `sys0` must have at least as many inputs as outputs.

## Object Description

An `idpoly` model represents a system as a continuous-time or discrete-time polynomial model with identifiable (estimable) coefficients.

A polynomial model of a system with input vector  $u$ , output vector  $y$ , and disturbance  $e$  takes the following form in discrete time:

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

In continuous time, a polynomial model takes the following form:

$$A(s)Y(s) = \frac{B(s)}{F(s)}U(s) + \frac{C(s)}{D(s)}E(s)$$

$U(s)$  are the Laplace transformed inputs to `sys`.  $Y(s)$  are the Laplace transformed outputs.  $E(s)$  is the Laplace transform of the disturbance.

For `idpoly` models, the coefficients of the polynomials  $A$ ,  $B$ ,  $C$ ,  $D$ , and  $F$  can be estimable parameters. The `idpoly` model stores the values of these matrix elements in the `a`, `b`, `c`, `d`, and `f` properties of the model.

Time series models are special cases of polynomial models for systems without measured inputs. For AR models, `b` and `f` are empty, and `c` and `d` are 1 for all outputs. For ARMA models, `b` and `f` are empty, while `d` is 1.

There are three ways to obtain an `idpoly` model:

- Estimate the `idpoly` model based on output or input-output measurements of a system, using such commands as `polyest`, `arx`, `armax`, `oe`, `bj`, `iv4`, or `ivar`. These estimation commands estimate the values of the free polynomial coefficients. The estimated values are stored in the `a`, `b`, `c`, `d`, and `f` properties of the resulting `idpoly` model. The `Report` property of the resulting model stores information about the estimation, such as handling of initial conditions and options used in estimation.

When you obtain an `idpoly` model by estimation, you can extract estimated coefficients and their uncertainties from the model using commands such as `polydata`, `getpar`, or `getcov`.

- Create an `idpoly` model using the `idpoly` command.  
You can create an `idpoly` model to configure an initial parameterization for estimation of a polynomial model to fit measured response data. When you do so, you can specify constraints on the polynomial coefficients. For example, you can fix the values of some coefficients, or specify minimum or maximum values for the free coefficients. You can then use the configured model as an input argument to `polyest` to estimate parameter values with those constraints.
- Convert an existing dynamic system model to an `idpoly` model using the `idpoly` command.

## Examples

### Multi-Output ARMAX Model

Create an `idpoly` model representing the one-input, two-output ARMAX model described by the following equations:

$$\begin{aligned}
 y_1(t) + 0.5y_1(t-1) + 0.9y_2(t-1) + 0.1y_2(t-2) &= \\
 &u(t) + 5u(t-1) + 2u(t-2) + e_1(t) + 0.01e_1(t-1) \\
 y_2(t) + 0.05y_2(t-1) + 0.3y_2(t-2) &= \\
 &10u(t-2) + e_2(t) + 0.1e_2(t-1) + 0.02e_2(t-2).
 \end{aligned}$$

$y_1$  and  $y_2$  are the two outputs, and  $u$  is the input.  $e_1$  and  $e_2$  are the white noise disturbances on the outputs  $y_1$  and  $y_2$  respectively.

To create the `idpoly` model, define the A, B, and C polynomials that describe the relationships between the outputs, inputs, and noise values. (Because there are no denominator terms in the system equations, B and F are 1.)

Define the cell array containing the coefficients of the A polynomials.

```

A = cell(2,2);
A{1,1} = [1 0.5];
A{1,2} = [0 0.9 0.1];
A{2,1} = [0];
A{2,2} = [1 0.05 0.3];

```

You can read the values of each entry in the A cell array from the left side of the equations describing the system. For example, `A{1,1}` describes the polynomial that gives the dependence of  $y_1$  on itself. This polynomial is  $A_{11} = 1 + 0.5q^{-1}$ , because each factor of  $q^{-1}$  corresponds to a unit time decrement. Therefore, `A{1,1} = [1 0.5]`, giving the coefficients of  $A_{11}$  in increasing exponents of  $q^{-1}$ .

Similarly, `A{1,2}` describes the polynomial that gives the dependence of  $y_1$  on  $y_2$ . From the equations,  $A_{12} = 0 + 0.9q^{-1} + 0.1q^{-2}$ . Thus, `A{1,2} = [0 0.9 0.1]`.

The remaining entries in A are similarly constructed.

Define the cell array containing the coefficients of the B polynomials.

```

B = cell(2,1);
B{1,1} = [1 5 2];

```

```
B{2,1} = [0 0 10];
```

B describes the polynomials that give the dependence of the outputs  $y_1$  and  $y_2$  on the input  $u$ . From the equations,  $B_{11} = 1 + 5q^{-1} + 2q^{-2}$ . Therefore,  $B\{1,1\} = [1 \ 5 \ 2]$ .

Similarly, from the equations,  $B_{21} = 0 + 0q^{-1} + 10q^{-2}$ . Therefore,  $B\{2,1\} = [0 \ 0 \ 10]$ .

Define the cell array containing the coefficients of the C polynomials.

```
C = cell(2,1);
C{1,1} = [1 0.01];
C{2,1} = [1 0.1 0.02];
```

C describes the polynomials that give the dependence of the outputs  $y_1$  and  $y_2$  on the noise terms  $e_1$  and  $e_2$ . The entries of C can be read from the equations similarly to those of A and B.

Create an idpoly model with the specified coefficients.

```
sys = idpoly(A,B,C)
```

```
sys =
```

Discrete-time ARMAX model:

Model for output number 1:  $A(z)y_1(t) = -A_i(z)y_i(t) + B(z)u(t) +$   
 $A(z) = 1 + 0.5 z^{-1}$

$$A_2(z) = 0.9 z^{-1} + 0.1 z^{-2}$$

$$B(z) = 1 + 5 z^{-1} + 2 z^{-2}$$

$$C(z) = 1 + 0.01 z^{-1}$$

Model for output number 2:  $A(z)y_2(t) = B(z)u(t) + C(z)e_2(t)$   
 $A(z) = 1 + 0.05 z^{-1} + 0.3 z^{-2}$

$$B(z) = 10 z^{-2}$$

$$C(z) = 1 + 0.1 z^{-1} + 0.02 z^{-2}$$

Sample time: unspecified

Parameterization:

Polynomial orders: na=[1 2;0 2] nb=[3;1] nc=[1;2] nk=[0;2]

Number of free coefficients: 12

Use "polydata", "getpvec", "getcov" for parameters and their uncertainty

Status:

Created by direct construction or transformation. Not estimated.

The display shows all the polynomials and allows you to verify them. The display also states that there are 12 free coefficients. Leading terms of diagonal entries in A are always fixed to 1. Leading terms of all other entries in A are always fixed to 0.

You can use `sys` to specify an initial parametrization for estimation with such commands as `polyest` or `armax`.

## Tips

- Although `idpoly` supports continuous-time models, `idtf` and `idproc` allow more choices for estimation of continuous-time models. Therefore, for some continuous-time applications, these model types are preferable.

## Input Arguments

### A,B,C,D,F

Initial values of polynomial coefficients.

For SISO models, specify the initial values of the polynomial coefficients as row vectors. Specify the coefficients in order of:

- Ascending powers of  $z^{-1}$  or  $q^{-1}$  (for discrete-time polynomial models).
- Descending powers of  $s$  or  $p$  (for continuous-time polynomial models).

The leading coefficients of A, C, D, and F must be 1. Use NaN for any coefficient whose initial value is not known.



For MIMO models with  $N_y$  outputs and  $N_u$  inputs, A, B, C, D, and F are cell arrays of row vectors. Each entry in the cell array contains the coefficients of a particular polynomial that relates input, output, and noise values.

Polynomial	Dimension	Relation Described
A	$N_y$ -by- $N_y$ array of row vectors	A{ <i>i</i> , <i>j</i> } contains coefficients of relation between output $y_i$ and output $y_j$
B, F	$N_y$ -by- $N_u$ array of row vectors	B{ <i>i</i> , <i>j</i> } and F{ <i>i</i> , <i>j</i> } contain coefficients of relations between output $y_i$ and input $u_j$
C, D	$N_y$ -by-1 array of row vectors	C{ <i>i</i> } and D{ <i>i</i> } contain coefficients of relations between output $y_i$ and noise $e_i$

The leading coefficients of the diagonal entries of A (A{*i*, *i*}, *i*=1: $N_y$ ) must be 1. The leading coefficients of the off-diagonal entries of A must be zero, for causality. The leading coefficients of all entries of C, D, and F, must be 1.

Use [] for any polynomial that is not present in the desired model structure. For example, to create an ARX model, use [] for C, D, and F. For an ARMA time series, use [] for B and F.

**Default:** B = []; C = 1 for all outputs; D = 1 for all outputs;  
F = []

### Ts

Sampling time. For continuous-time models, Ts = 0. For discrete-time models, Ts is a positive scalar representing the sampling period expressed in the unit specified by the TimeUnit property of the model. To denote a discrete-time model with unspecified sampling time, set Ts = -1.

**Default:**  $-1$  (discrete-time model with unspecified sampling time)

## NoiseVariance

The variance (covariance matrix) of the model innovations  $e$ .

An identified model includes a white, Gaussian noise component  $e(t)$ . `NoiseVariance` is the variance of this noise component. Typically, a model estimation function (such as `polyest`) determines this variance. Use this input to specify an initial value for the noise variance when you create an `idpoly` model.

For SISO models, `NoiseVariance` is a scalar. For MIMO models, `NoiseVariance` is a  $N_y$ -by- $N_y$  matrix, where  $N_y$  is the number of outputs in the system.

**Default:**  $N_y$ -by- $N_y$  identity matrix

## sys0

Dynamic system.

Any dynamic system, such as `idss`, `idtf`, or `tf` to be converted into an `idpoly` object.

For the syntax `sys = idpoly(sys0, 'split')`, `sys0` must be a numeric (non-identified) `tf`, `zpk`, or `ss` model object. Also, `sys0` must have at least as many inputs as outputs. Finally, the subsystem `sys0(:, Ny+1:Nu)` must be biproper.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Use `Name, Value` arguments to specify additional properties of `idpoly` models during model creation. For example,

`idpoly(A,B,C,D,F,1,0,'InputName','Voltage')` creates an `idpoly` model with the `InputName` property set to `Voltage`.

## Properties

`idpoly` object properties include:

### **a,b,c,d,f**

Values of polynomial coefficients.

If you create an `idpoly` model `sys` using the `idpoly` command, `sys.a`, `sys.b`, `sys.c`, `sys.d`, and `sys.f` contain the initial coefficient values that you specify with the `A`, `B`, `C`, `D`, and `F` input arguments, respectively.

If you obtain an `idpoly` model by identification, then `sys.a`, `sys.b`, `sys.c`, `sys.d`, and `sys.f` contain the estimated values of the coefficients.

For an `idpoly` model `sys`, each property `sys.a`, `sys.b`, `sys.c`, `sys.d`, and `sys.f` is an alias to the corresponding `Value` entry in the `Structure` property of `sys`. For example, `sys.a` is an alias to the value of the property `sys.Structure.a.Value`.

For SISO polynomial models, the values of the numerator coefficients are stored as a row vector in order of:

- Ascending powers of  $z^{-1}$  or  $q^{-1}$  (for discrete-time transfer functions).
- Descending powers of  $s$  or  $p$  (for continuous-time transfer functions).

The leading coefficients of `A`, `C`, and `D` are fixed to 1. Any coefficient whose initial value is not known is stored as `NaN`.

For MIMO models with  $N_y$  outputs and  $N_u$  inputs, `A`, `B`, `C`, `D`, and `F` are cell arrays of row vectors. Each entry in the cell array contains the coefficients of a particular polynomial that relates input, output, and noise values.

Polynomial	Dimension	Relation Described
A	$N_y$ -by- $N_y$ array of row vectors	A{ <i>i</i> , <i>j</i> } contains coefficients of relation between output $y_i$ and output $y_j$
B, F	$N_y$ -by- $N_u$ array of row vectors	B{ <i>i</i> , <i>j</i> } and F{ <i>i</i> , <i>j</i> } contain coefficients of relations between output $y_i$ and input $u_j$
C, D	$N_y$ -by-1 array of row vectors	C{ <i>i</i> } and D{ <i>i</i> } contain coefficients of relations between output $y_i$ and noise $e_i$

The leading coefficients of the diagonal entries of A (A{*i*, *i*}, *i*=1: $N_y$ ) are fixed to 1. The leading coefficients of the off-diagonal entries of A are fixed to zero. The leading coefficients of all entries of C, D, and F, are fixed to 1.

For a time series (a model with no measured inputs), B = [] and F = [].

**Default:** B = []; C = 1 for all outputs; D = 1 for all outputs; F = []

## Variable

String specifying the polynomial model display variable. Variable can take the following values:

- 'z^-1' — Default for discrete-time models
- 'q^-1' — Equivalent to 'z^-1'
- 's' — Default for continuous-time models
- 'p' — Equivalent to 's'

The value of Variable is reflected in the display, and also affects the interpretation of the A, B, C, D, and F coefficient vectors for discrete-time

models. For `Variable = 'z^-1'` or `'q^-1'`, the coefficient vectors are ordered as ascending powers of the variable.

### **ioDelay**

Transport delays. `ioDelay` is a numeric array specifying a separate transport delay for each input/output pair.

If you create an `idpoly` model `sys` using the `idpoly` command, `sys.ioDelay` contains the initial values of the transport delay that you specify with a `Name, Value` argument pair.

If you obtain an `idpoly` model `sys` by identification, then `sys.ioDelay` contains the estimated values of the transport delay.

For an `idpoly` model `sys`, the property `sys.ioDelay` is an alias to the value of the property `sys.Structure.ioDelay.Value`.

For continuous-time systems, transport delays are expressed in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify transport are expressed as integers denoting delay of a multiple of the sampling period  $T_s$ .

For a MIMO system with  $N_y$  outputs and  $N_u$  inputs, set `ioDelay` is a  $N_y$ -by- $N_u$  array, where each entry is a numerical value representing the transport delay for the corresponding input/output pair. You can set `ioDelay` to a scalar value to apply the same delay to all input/output pairs.

**Default:** 0 for all input/output pairs

### **IntegrateNoise**

Logical vector, denoting presence or absence of integration on noise channels.

Specify `IntegrateNoise` as a logical vector of length equal to the number of outputs.

`IntegrateNoise(i) = true` indicates that the noise channel for the  $i$ th output contains an integrator. In this case, the corresponding

$D$  polynomial contains an additional term which is not represented in the property `sys.d`. This integrator term is equal to  $[1 \ 0]$  for continuous-time systems, and equal to  $[1 \ -1]$  for discrete-time systems.

**Default:** 0 for all output channels

## Structure

Information about the estimable parameters of the `idpoly` model. `sys.Structure.a`, `sys.Structure.b`, `sys.Structure.c`, `sys.Structure.d`, and `sys.Structure.f` contain information about the polynomial coefficients. `sys.Structure.ioDelay` contains information about the transport delay. `sys.Structure.IntegrateNoise` contain information about the integration terms on the noise. Each contains the following fields:

- **Value** — Parameter values. For example, `sys.Structure.a.Value` contains the initial or estimated values of the  $A$  coefficients.

NaN represents unknown parameter values.

For SISO models, each property `sys.a`, `sys.b`, `sys.c`, `sys.d`, `sys.f`, and `sys.ioDelay` is an alias to the corresponding `Value` entry in the `Structure` property of `sys`. For example, `sys.a` is an alias to the value of the property `sys.Structure.a.Value`

For MIMO models, `sys.a{i,j}` is an alias to `sys.Structure.a(i,j).Value`, and similarly for the other identifiable coefficient values.

- **Minimum** — Minimum value that the parameter can assume during estimation. For example, `sys.Structure.ioDelay.Minimum = 0.1` constrains the transport delay to values greater than or equal to 0.1.

`sys.Structure.ioDelay.Minimum` must be greater than or equal to zero.

- **Maximum** — Maximum value that the parameter can assume during estimation.

- **Free** — Logical value specifying whether the parameter is a free estimation variable. If you want to fix the value of a parameter during estimation, set the corresponding `Free = false`. For example, if  $B$  is a 3-by-3 matrix, `sys.Structure.a.Free = eyes(3)` fixes all of the off-diagonal entries in  $B$  to the values specified in `sys.Structure.b.Value`. In this case, only the diagonal entries in  $B$  are estimable.

For fixed values, such as the leading coefficients in `sys.Structure.a.Value`, the corresponding value of `Free` is always `false`.

- **Scale** — Scale of the parameter's value. `Scale` is not used in estimation.
- **Info** — Structure array for storing parameter units and labels. The structure has `Label` and `Unit` fields.

Use these fields for your convenience, to store strings that describe parameter units and labels.

For a MIMO model with  $N_y$  outputs and  $N_u$  inputs, the dimensions of the `Structure` elements are as follows:

- `sys.Structure.a` —  $N_y$ -by- $N_y$
- `sys.Structure.b` —  $N_y$ -by- $N_u$
- `sys.Structure.c` —  $N_y$ -by-1
- `sys.Structure.d` —  $N_y$ -by-1
- `sys.Structure.f` —  $N_y$ -by- $N_u$

An inactive polynomial, such as the  $B$  polynomial in a time series model, is not available as a parameter in the `Structure` property. For example, `sys = idpoly([1 -0.2 0.5])` creates an AR model. `sys.Structure` contains the fields `sys.Structure.a`, `sys.Structure.ioDelay`, and `sys.Structure.IntegrateNoise`. However, there is no field in `sys.Structure` corresponding to  $b$ ,  $c$ ,  $d$ , or  $f$ .

### NoiseVariance

The variance (covariance matrix) of the model innovations  $e$ .

An identified model includes a white, Gaussian noise component  $e(t)$ . `NoiseVariance` is the variance of this noise component. Typically, the model estimation function (such as `arx`) determines this variance.

For SISO models, `NoiseVariance` is a scalar. For MIMO models, `NoiseVariance` is a  $N_y$ -by- $N_y$  matrix, where  $N_y$  is the number of outputs in the system.

### **Report**

Information about the estimation process.

`Report` contains the following fields:

- `InitialCondition` — Whether estimation estimated initial conditions or fixed them at zero.
- `Fit` — Quantitative quality assessment of estimation, including percent fit to data and final prediction error.
- `Parameters` — Estimated values of model parameters and their covariance.
- `OptionsUsed` — Options used during estimation (see `ssestOptions` or `n4sidOptions`).
- `RandState` — Random number stream state at start of estimation.
- `Status` — Whether model was obtained by construction, estimated, or modified after estimation.
- `Method` — Name of estimation method used.
- `DataUsed` — Attributes of data used for estimation, such as name and sampling time.
- `Termination` — Termination conditions for the iterative search scheme used for prediction error minimization, such as final cost value or stopping criterion. Not available when the model is estimated using `arx` or instrument variable approaches.



### InputDelay

Input delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. For continuous-time systems, specify input delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify input delays in integer multiples of the sampling period `Ts`. For example, `InputDelay = 3` means a delay of three sampling periods.

For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector, where each entry is a numerical value representing the input delay for the corresponding input channel. You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0 for all input channels

### OutputDelay

Output delays.

For identified systems, like `idpoly`, `OutputDelay` is fixed to zero.

### Ts

Sampling time. For continuous-time models, `Ts = 0`. For discrete-time models, `Ts` is a positive scalar representing the sampling period expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sampling time, set `Ts = -1`.

Changing this property does not discretize or resample the model. Use `c2d` and `d2c` to convert between continuous- and discrete-time representations. Use `d2d` to change the sampling time of a discrete-time system.

**Default:** -1 (discrete-time model with unspecified sampling time)

### TimeUnit

String representing the unit of the time variable, any time delays in the model (for continuous-time models), and the sampling time  $T_s$  (for discrete-time models). `TimeUnit` can take the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

## **InputName**

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

Alternatively, use automatic vector expansion to assign input names for multi-input models. For example, if `sys` is a two-input model, enter:

```
sys.InputName = 'controls';
```

The software automatically expands the input names to `{'controls(1)'; 'controls(2)'}`.

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `sys.u` is equivalent to `sys.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** Empty string `''` for all input channels

### InputUnit

Input channel units. Use `InputUnit` to keep track of input signal units. Set `InputUnit` to a string for single-input model, or to a cell array of strings for a multi-input model. `InputUnit` has no effect on system behavior.

**Default:** Empty string `''` for all input channels

### InputGroup

Input channel groups. The `InputGroup` property lets you assign the input channels of MIMO systems into groups and refer to each group by name. Specify input groups as a structure whose field names are the group names and whose field values are the input channels belong to each group. For example:

```
sys.InputGroup.controls = [1 2];  
sys.InputGroup.noise = [3 5];
```

creates input groups named `controls` and `noise` that include input channels 1, 2 and 3, 5, respectively. You can then extract the subsystem from the `controls` inputs to all outputs using:

```
sys(:, 'controls')
```

**Default:** Struct with no fields

## OutputName

Output channel names. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

Alternatively, use automatic vector expansion to assign output names for multi-output models. For example, if `sys` is a two-output model, enter:

```
sys.OutputName = 'measurements';
```

The software automatically expands the output names to `{'measurements(1)'; 'measurements(2)'}`.

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `sys.y` is equivalent to `sys.OutputName`.

Output channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** Empty string '' for all input channels

## OutputUnit

Output channel units. Use `OutputUnit` to keep track of output signal units. Set `OutputUnit` to a string for single-input model, or to a cell array of strings for a multi-input model. `OutputUnit` has no effect on system behavior.

**Default:** Empty string '' for all input channels

## OutputGroup

Output channel groups. The `OutputGroup` property lets you assign the output channels of MIMO systems into groups and refer to each group by name. Specify output groups as a structure whose field names are

the group names and whose field values are the output channels belong to each group. For example:

```
sys.OutputGroup.temperature = [1];  
sys.InputGroup.measurement = [3 5];
```

creates output groups named `temperature` and `measurement` that include output channels 1, and 3, 5, respectively. You can then extract the subsystem from all inputs to the measurement outputs using:

```
sys('measurement',:)
```

**Default:** Struct with no fields

### **Name**

System name. Set `Name` to a string to label the system.

**Default:** ''

### **Notes**

Any text that you wish to associate with the system. Set `Notes` to a string or a cell array of strings.

**Default:** {}

### **UserData**

Any type of data you wish to associate with system. Set `UserData` to any MATLAB data type.

**Default:** []

## **See Also**

`polydata` | `arx` | `armax` | `bj` | `oe` | `ar` | `polyest` |  
`setPolyFormat` | `idss` | `idproc` | `idtf` | `iv4` | `ivar`

## **Related Examples**

- “How to Estimate Polynomial Models in the GUI”
- “How to Estimate Polynomial Models at the Command Line”
- “Polynomial Sizes and Orders of Multi-Output Polynomial Models”

## **Concepts**

- “What Are Polynomial Models?”
- “Dynamic System Models”

**Purpose** Continuous-time process model with identifiable parameters

**Syntax**  
`sys = idproc(type)`  
`sys = idproc(type,Name,Value)`

**Description** `sys = idproc(type)` creates a continuous-time process model with identifiable parameters. `type` is a string that specifies aspects of the model structures, such as the number of poles in the model, whether the model includes an integrator, and whether the model includes a time delay.

`sys = idproc(type,Name,Value)` creates a process model with additional attributes specified by one or more `Name,Value` pair arguments.

## Object Description

An `idproc` model represents a system as a continuous-time process model with identifiable (estimable) coefficients.

A simple SISO process model has a gain, a time constant, and a delay:

$$sys = \frac{K_p}{1 + T_{p1}s} e^{-T_d s}.$$

$K_p$  is a proportional gain.  $T_{p1}$  is the time constant of the real pole, and  $T_d$  is the transport delay (dead time).

More generally, `idproc` can represent process models with up to three poles and a zero:

$$sys = K_p \frac{1 + T_z s}{(1 + T_{p1}s)(1 + T_{p2}s)(1 + T_{p3}s)} e^{-T_d s}.$$

Two of the poles can be a complex conjugate (underdamped) pair. In that case, the general form of the process model is:

$$\text{sys} = K_p \frac{1 + T_z s}{(1 + 2\zeta T_\omega s + (T_\omega s)^2)(1 + T_{p3} s)} e^{-T_d s}.$$

$T_\omega$  is the time constant of the complex pair of poles, and  $\zeta$  is the associated damping constant.

In addition, any `idproc` model can have an integrator. For example, the following is a process model that you can represent with `idproc`:

$$\text{sys} = K_p \frac{1}{s(1 + 2\zeta T_\omega s + (T_\omega s)^2)} e^{-T_d s}.$$

This model has no zero ( $T_z = 0$ ). The model has a complex pair of poles. The model also has an integrator, represented by the  $1/s$  term.

For `idproc` models, all the time constants, the delay, the proportional gain, and the damping coefficient can be estimable parameters. The `idproc` model stores the values of these parameters in properties of the model such as `Kp`, `Tp1`, and `Zeta`. (See “Properties” on page 2-428 for more information.)

A MIMO process model contains a SISO process model corresponding to each input-output pair in the system. For `idproc` models, the form of each input-output pair can be independently specified. For example, a two-input, one-output process can have one channel with two poles and no zero, and another channel with a zero, a pole, and an integrator. All the coefficients are independently estimable parameters.

There are two ways to obtain an `idproc` model:

- Estimate the `idproc` model based on output or input-output measurements of a system, using the `procest` command. `procest` estimates the values of the free parameters such as gain, time constants, and time delay. The estimated values are stored as properties of the resulting `idproc` model. For example, the properties `sys.Tz` and `sys.Kp` of an `idproc` model `sys` store the zero time constant and the proportional gain, respectively. (See “Properties” on page 2-428 for more information.) The `Report` property of the



resulting model stores information about the estimation, such as handling of initial conditions and options used in estimation.

When you obtain an `idproc` model by estimation, you can extract estimated coefficients and their uncertainties from the model using commands such as `getpar` and `getcov`.

- Create an `idproc` model using the `idproc` command.

You can create an `idproc` model to configure an initial parameterization for estimation of a process model. When you do so, you can specify constraints on the parameters. For example, you can fix the values of some coefficients, or specify minimum or maximum values for the free coefficients. You can then use the configured model as an input argument to `procest` to estimate parameter values with those constraints.

## Examples

### SISO Process Model with Complex Poles and Time Delay

Create a process model with a pair of complex poles and a time delay. Set the initial value of the model to the following:

$$\text{sys} = \frac{0.01}{1 + 2(0.1)(10)s + (10s)^2} e^{-5s}.$$

Create a process model with the specified structure.

```
sys = idproc('P2DU')
```

```
sys =
Process model with transfer function:
      Kp
G(s) = ----- * exp(-Td*s)
      1+2*Zeta*Tw*s+(Tw*s)^2
```

```
Kp = NaN
```

```
Tw = NaN
```

```
Zeta = NaN  
Td = NaN
```

Parameterization:

```
'P2DU'
```

Number of free coefficients: 4

Use "getpvec", "getcov" for parameters and their uncertainties.

Status:

Created by direct construction or transformation. Not estimated.

The input string 'P2DU' specifies an underdamped pair of poles and a time delay. The display shows that `sys` has the desired structure. The display also shows that the four free parameters, `Kp`, `Tw`, `Zeta`, and `Td` are all initialized to `NaN`.

Set the initial values of all parameters to the desired values.

```
sys.Kp = 0.01;  
sys.Tw = 10;  
sys.Zeta = 0.1;  
sys.Td = 5;
```

You can use `sys` to specify this parametrization and these initial guesses for process model estimation with `procest`.

## **MIMO Process Model**

Create a one-input, three-output process model, where each channel has two real poles and a zero, but only the first channel has a time delay, and only the first and third channels have an integrator.

```
type = {'P2ZDI'; 'P2Z'; 'P2ZI'};  
sys = idproc(type);
```

Providing an array of type strings causes `idproc` to create a MIMO model where each type string in the array defines the structure of the corresponding I/O pair. Since `type` is a column vector of strings, `sys`

is a one-input, 3-output model having the specified parametrization structure. The string `type{k,1}` specifies the structure of the subsystem `sys(k,1)`. All identifiable parameters are initialized to NaN.

### Array of Process Models

Create a 3-by-1 array of process models, each containing one output and two input channels.

Create cell array of type strings.

```
type1 = {'P1D', 'P2DZ'};
type2 = {'P0', 'P3UI'};
type3 = {'P2D', 'P2DI'};
type = cat(3, type1, type2, type3);
size(type)
```

```
ans =
```

```
     1     2     3
```

Use `type` to create the array.

```
sysarr = idproc(type);
```

The first two dimensions of the cell array `type` set the output and input dimensions of each model in the array of process models. The remaining dimensions of the cell array set the array dimensions. Thus, `sysarr` is a 3-model array of 2-input, one-output process models.

Select a model from the array.

```
sysarr(:, :, 2)
```

```
Process model with 2 inputs: y = G11(s)u1 + G12(s)u2
```

```
From input 1 to output 1:
```

```
G11(s) = Kp
```

```
Kp = NaN
```

From input 2 to output 1:

$$G_{12}(s) = \frac{K_p}{s(1+2*\text{Zeta}*T_w*s+(T_w*s)^2)(1+T_{p3}*s)}$$

Kp = NaN  
 Tw = NaN  
 Zeta = NaN  
 Tp3 = NaN

Parameterization:

'P0' 'P3IU'

Number of free coefficients: 5

Use "getpvec", "getcov" for parameters and their uncertainties.

Status:

Created by direct construction or transformation. Not estimated.

This two-input, one-output model corresponds to the `type2` entry in the `type` cell array.

## Input Arguments

### **type**

String or cell array of strings characterizing the model structure.

For SISO models, `type` is a string made up of a series of characters that specify aspects of the model structure.

Characters	Meaning
Pk	A process model with $k$ poles (not including an integrator). $k$ must be 0, 1, 2, or 3.
Z	The process model includes a zero ( $T_z \neq 0$ ). A <code>type</code> string with P0 cannot include Z (a process model with no poles cannot include a zero).

Characters	Meaning
D	The process model includes a time delay (deadtime) ( $T_d \neq 0$ ).
I	The process model includes an integrator ( $1/s$ ).
U	The process model is underdamped. In this case, the process model includes a complex pair of poles

Every type string must begin with one of P0, P1, P2, or P3. All other components of the string are optional.

Example type strings include:

- 'P1D' specifies a process model with one pole and a time delay (deadtime) term:

$$\text{sys} = \frac{K_p}{1 + T_{p1}s} e^{-T_d s}.$$

$K_p$ ,  $T_{p1}$ , and  $T_d$  are the identifiable parameters of this model.

- 'P2U' creates a process model with a pair of complex poles:

$$\text{sys} = \frac{K_p}{\left(1 + 2\zeta T_\omega s + (T_\omega s)^2\right)}.$$

$K_p$ ,  $T_\omega$ , and Zeta are the identifiable parameters of this model.

- 'P3ZDI' creates a process model with three poles. All poles are real, because the string does not include U. The model also includes a zero, a time delay, and an integrator:

$$\text{sys} = K_p \frac{1 + T_z s}{s(1 + T_{p1}s)(1 + T_{p2}s)(1 + T_{p3}s)} e^{-T_d s}.$$

The identifiable parameters of this model are `Kp`, `Tz`, `Tp1`, `Tp2`, `Tp3`, and `Td`.

The values of all parameters in a particular model structure are initialized to NaN. You can change them to finite values by setting the values of the corresponding `idproc` model properties after you create the model. For example, `sys.Td = 5` sets the initial value of the time delay of `sys` to 5.

For a MIMO process model with `Ny` outputs and `Nu` inputs, `type` is an `Ny`-by-`Nu` cell array of strings specifying the structure of each input/output pair in the model. For example, `type{i,j}` specifies the type of the subsystem `sys(i,j)` from the `j`th input to the `y`th output.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Use `Name`, `Value` arguments to specify parameter initial values and additional properties of `idproc` models during model creation. For example, `sys = idproc('p2z','InputName','Voltage','Kp',10,'Tz',0);` creates an `idtf` model with the `InputName` property set to `Voltage`. The command also initializes the parameter `Kp` to a value of 10, and `Tz` to 0.

## Properties

`idproc` object properties include:

### Type

Cell array of strings characterizing the model structure.

For a SISO model `sys`, the property `sys.Type` contains a single string specifying the structure of the system.

For a MIMO model with `Ny` outputs and `Nu` inputs, `sys.Type` is an `Ny`-by-`Nu` cell array of strings specifying the structure of each input/output pair in the model. For example, `type{i,j}` specifies the

structure of the subsystem  $\text{sys}(i, j)$  from the  $j$ th input to the  $i$ th output.

The strings are made up of a series of characters that specify aspects of the model structure, as follows.

Characters	Meaning
Pk	A process model with $k$ poles (not including an integrator). $k$ is 0, 1, 2, or 3.
Z	The process model includes a zero ( $T_z \neq 0$ ).
D	The process model includes a time delay (deadtime) ( $T_d \neq 0$ ).
I	The process model includes an integrator (1/s).
U	The process model is underdamped. In this case, the process model includes a complex pair of poles

If you create an `idproc` model `sys` using the `idproc` command, `sys.Type` contains the strings that you specify with the `type` input argument.

If you obtain an `idproc` model by identification using `procest`, then `sys.Type` contains the strings describing the model structures that you specified for that identification.

In general, you cannot change the type string of an existing model. However, you can change whether the model contains an integrator using the property `sys.Integration`.

### **Kp, Tp1, Tp2, Tp3, Tz, Tw, Zeta, Td**

Values of process model parameters.

If you create an `idproc` model using the `idproc` command, the values of all parameters present in the model structure initialize by default to NaN. The values of parameters not present in the model structure are fixed to 0. For example, if you create a model, `sys`, of type 'P1D', then `Kp`, `Tp1`, and `Td` are initialized to NaN and are identifiable (free) parameters. All remaining parameters, such as `Tp2` and `Tz`, are inactive

in the model. The values of inactive parameters are fixed to zero and cannot be changed.

For a MIMO model with  $N_y$  outputs and  $N_u$  inputs, each parameter value is an  $N_y$ -by- $N_u$  cell array of strings specifying the corresponding parameter value for each input/output pair in the model. For example, `sys.Kp(i,j)` specifies the  $K_p$  value of the subsystem `sys(i,j)` from the  $j$ th input to the  $i$ th output.

For an `idproc` model `sys`, each parameter value property such as `sys.Kp`, `sys.Tp1`, `sys.Tz`, and the others is an alias to the corresponding Value entry in the Structure property of `sys`. For example, `sys.Tp3` is an alias to the value of the property `sys.Structure.Tp3.Value`.

**Default:** For each parameter value, NaN if the process model structure includes the particular parameter; 0 if the structure does not include the parameter.

## Integration

Logical value or matrix denoting the presence or absence of an integrator in the transfer function of the process model.

For a SISO model `sys`, `sys.Integration = true` if the model contains an integrator.

For a MIMO model, `sys.Integration(i,j) = true` if the transfer function from the  $j$ th input to the  $i$ th output contains an integrator.

When you create a process model using the `idproc` command, the value of `sys.Integration` is determined by whether the corresponding type string contains I.

## NoiseTF

Coefficients of the noise transfer function.

`sys.NoiseTF` stores the coefficients of the numerator and the denominator polynomials for the noise transfer function  $H(s) = N(s)/D(s)$ .



`sys.NoiseTF` is a structure with fields `num` and `den`. Each field is a cell array of  $N_y$  row vectors, where  $N_y$  is the number of outputs of `sys`. These row vectors specify the coefficients of the noise transfer function numerator and denominator in order of decreasing powers of  $s$ .

Typically, the noise transfer function is automatically computed by the estimation function `procest`. You can specify a noise transfer function that `procest` uses as an initial value. For example:

```
NoiseNum = {[1 2.2]; [1 0.54]};
NoiseDen = {[1 1.3]; [1 2]};
NoiseTF = struct('num', {NoiseNum}, 'den', {NoiseDen});
sys = idproc({'p2'; 'p1di'}); % 2-output, 1-input process model
sys.NoiseTF = NoiseTF;
```

Each vector in `sys.NoiseTF.num` and `sys.NoiseTF.den` must be of length 3 or less (second-order in  $s$  or less). Each vector must start with 1. The length of a numerator vector must be equal to that of the corresponding denominator vector, so that  $H(s)$  is always biproper.

**Default:**

```
struct('num', {num2cell(ones(Ny,1))}, 'den', {num2cell(ones(Ny,1))})
```

**Structure**

Information about the estimable parameters of the `idproc` model.

`sys.Structure` includes one entry for each parameter in the model structure of `sys`. For example, if `sys` is of type 'P1D', then `sys` includes identifiable parameters `Kp`, `Tp1`, and `Td`. Correspondingly, `sys.Structure.Kp`, `sys.Structure.Tp1`, and `sys.Structure.Td` contain information about each of these parameters, respectively.

Each of these parameter entries in `sys.Structure` contains the following fields:

- **Value** — Parameter values. For example, `sys.Structure.Kp.Value` contains the initial or estimated values of the  $K_p$  parameter.  
NaN represents unknown parameter values.

For SISO models, each parameter value property such as `sys.Kp`, `sys.Tp1`, `sys.Tz`, and the others is an alias to the corresponding Value entry in the Structure property of `sys`. For example, `sys.Tp3` is an alias to the value of the property `sys.Structure.Tp3.Value`.

For MIMO models, `sys.Kp{i,j}` is an alias to `sys.Structure(i,j).Kp.Value`, and similarly for the other identifiable coefficient values.

- **Minimum** — Minimum value that the parameter can assume during estimation. For example, `sys.Structure.Kp.Minimum = 1` constrains the proportional gain to values greater than or equal to 1.
- **Maximum** — Maximum value that the parameter can assume during estimation.
- **Free** — Logical value specifying whether the parameter is a free estimation variable. If you want to fix the value of a parameter during estimation, set the corresponding `Free = false`. For example, to fix the dead time to 5:

```
sys.Td = 5;  
sys.Structure.Td.Free = false;
```

- **Scale** — Scale of the parameter's value. `Scale` is not used in estimation.
- **Info** — Structure array for storing parameter units and labels. The structure has `Label` and `Unit` fields.

Use these fields for your convenience, to store strings that describe parameter units and labels.

`Structure` also includes a field `Integration` that stores a logical array indicating whether each corresponding process model has an integrator. `sys.Structure.Integration` is an alias to `sys.Integration`.

For a MIMO model with `Ny` outputs and `Nu` input, `Structure` is an `Ny`-by-`Nu` array. The element `Structure(i,j)` contains information corresponding to the process model for the `(i,j)` input-output pair.

### NoiseVariance

The variance (covariance matrix) of the model innovations  $e$ .

An identified model includes a white, Gaussian noise component  $e(t)$ . `NoiseVariance` is the variance of this noise component. Typically, the model estimation function (such as `procest`) determines this variance.

For SISO models, `NoiseVariance` is a scalar. For MIMO models, `NoiseVariance` is a  $N_y$ -by- $N_y$  matrix, where  $N_y$  is the number of outputs in the system.

### Report

Information about the estimation process.

`Report` contains the following fields:

- `InitialCondition` — Whether estimation estimated initial conditions or fixed them at zero.
- `Fit` — Quantitative quality assessment of estimation, including percent fit to data and final prediction error.
- `Parameters` — Estimated values of model parameters and input offset, and their covariances.
- `OptionsUsed` — Options used during estimation (see `procestOptions`).
- `RandState` — Random number stream state at start of estimation.
- `Status` — Whether model was obtained by construction, estimated, or modified after estimation.
- `Method` — Name of estimation method used.
- `DataUsed` — Attributes of data used for estimation, such as name and sampling time.
- `Termination` — Termination conditions for the iterative search scheme used for prediction error minimization, such as final cost value or stopping criterion.

## **InputDelay**

Input delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. Specify input delays in the time unit stored in the `TimeUnit` property.

For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector, where each entry is a numerical value representing the input delay for the corresponding input channel. You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0 for all input channels

## **OutputDelay**

Output delays.

For identified systems, like `idproc`, `OutputDelay` is fixed to zero.

## **Ts**

Sampling time. For `idproc`, `Ts` is fixed to zero because all `idproc` models are continuous time.

## **TimeUnit**

String representing the unit of the time variable, any time delays in the model (for continuous-time models), and the sampling time `Ts` (for discrete-time models). `TimeUnit` can take the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'

- 'weeks'
- 'months'
- 'years'

Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### InputName

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

Alternatively, use automatic vector expansion to assign input names for multi-input models. For example, if `sys` is a two-input model, enter:

```
sys.InputName = 'controls';
```

The software automatically expands the input names to `{'controls(1)'; 'controls(2)'}`.

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `sys.u` is equivalent to `sys.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** Empty string '' for all input channels

### InputUnit

Input channel units. Use `InputUnit` to keep track of input signal units. Set `InputUnit` to a string for single-input model, or to a cell array of

strings for a multi-input model. `InputUnit` has no effect on system behavior.

**Default:** Empty string '' for all input channels

## InputGroup

Input channel groups. The `InputGroup` property lets you assign the input channels of MIMO systems into groups and refer to each group by name. Specify input groups as a structure whose field names are the group names and whose field values are the input channels belong to each group. For example:

```
sys.InputGroup.controls = [1 2];  
sys.InputGroup.noise = [3 5];
```

creates input groups named `controls` and `noise` that include input channels 1, 2 and 3, 5, respectively. You can then extract the subsystem from the `controls` inputs to all outputs using:

```
sys(:, 'controls')
```

**Default:** Struct with no fields

## OutputName

Output channel names. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

Alternatively, use automatic vector expansion to assign output names for multi-output models. For example, if `sys` is a two-output model, enter:

```
sys.OutputName = 'measurements';
```

The software automatically expands the output names to `{ 'measurements(1)'; 'measurements(2) '}`.

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `sys.y` is equivalent to `sys.OutputName`.

Output channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** Empty string `''` for all input channels

### OutputUnit

Output channel units. Use `OutputUnit` to keep track of output signal units. Set `OutputUnit` to a string for single-input model, or to a cell array of strings for a multi-input model. `OutputUnit` has no effect on system behavior.

**Default:** Empty string `''` for all input channels

### OutputGroup

Output channel groups. The `OutputGroup` property lets you assign the output channels of MIMO systems into groups and refer to each group by name. Specify output groups as a structure whose field names are the group names and whose field values are the output channels belong to each group. For example:

```
sys.OutputGroup.temperature = [1];  
sys.InputGroup.measurement = [3 5];
```

creates output groups named `temperature` and `measurement` that include output channels 1, and 3, 5, respectively. You can then extract the subsystem from all inputs to the `measurement` outputs using:

```
sys('measurement',:)
```

**Default:** Struct with no fields

**Name**

System name. Set `Name` to a string to label the system.

**Default:** ''

**Notes**

Any text that you wish to associate with the system. Set `Notes` to a string or a cell array of strings.

**Default:** {}

**UserData**

Any type of data you wish to associate with system. Set `UserData` to any MATLAB data type.

**Default:** []

**See Also**

`idtf` | `procest` | `idss` | `tfest` | `ssest` | `pem`



---

<b>Purpose</b>	Resample time-domain data by decimation or interpolation
<b>Syntax</b>	<pre>datar = idresamp(data,R) datar = idresamp(data,R,order,tol) [datar,res_fact] = idresamp(data,R,order,tol)</pre>
<b>Description</b>	<p><code>datar = idresamp(data,R)</code> resamples data on a new sample interval <code>R</code> and stores the resampled data as <code>datar</code>.</p> <p><code>datar = idresamp(data,R,order,tol)</code> filters the data by applying a filter of specified order before interpolation and decimation. Replaces <code>R</code> by a rational approximation that is accurate to a tolerance <code>tol</code>.</p> <p><code>[datar,res_fact] = idresamp(data,R,order,tol)</code> returns <code>res_fact</code>, which corresponds to the value of <code>R</code> approximated by a rational expression.</p>
<b>Input Arguments</b>	<p><code>data</code> Name of time-domain <code>iddata</code> object or a matrix of data. Can be input-output or time-series data.</p> <p>Data must be sampled at equal time intervals.</p> <p><code>R</code> Resampling factor, such that <math>R &gt; 1</math> results in decimation and <math>R &lt; 1</math> results in interpolation.</p> <p>Any positive number you specify is replaced by the rational approximation, <math>Q/P</math>.</p> <p><code>order</code> Order of the filters applied before interpolation and decimation.</p> <p>Default: 8</p> <p><code>tol</code> Tolerance of the rational approximation for the resampling factor <code>R</code>.</p>

# idresamp

---

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 Arguments

`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**

State-space model with identifiable parameters

**Syntax**

```
sys = idss(A,B,C,D)
sys = idss(A,B,C,D,K)
sys = idss(A,B,C,D,K,x0)
sys = idss(A,B,C,D,K,x0,Ts)
sys = idss( __ ,Name,Value)

sys = idss(sys0)
sys = idss(sys0,'split')
```

**Description**

`sys = idss(A,B,C,D)` creates a state-space model with identifiable parameters. A, B, C, and D are the initial values of the state-space matrices. By default, `sys` is discrete-time model with unspecified sampling time and no state disturbance element.

`sys = idss(A,B,C,D,K)` creates a state-space model with a disturbance element given by the matrix K.

`sys = idss(A,B,C,D,K,x0)` creates a state-space model with initial state values given by the vector `x0`.

`sys = idss(A,B,C,D,K,x0,Ts)` creates a state-space model with sampling time `Ts`. Use `Ts = 0` to create a continuous-time model.

`sys = idss( __ ,Name,Value)` creates a state-space model using additional options specified by one or more `Name,Value` pair arguments.

`sys = idss(sys0)` converts any dynamic system model, `sys0`, to `idss` model form. When you convert a numeric (non-identified) model to `idss` form, the state-space data of `sys0` define the A, B, C, and D matrices of the converted model. The disturbance matrix K is fixed to zero. The

NoiseVariance value defaults to `eye(Ny)`, where `Ny` is the number of outputs of `sys`.

`sys = idss(sys0, 'split')` converts `sys0` to `idss` model form, and treats the last `Ny` input channels of `sys0` as noise channels in the returned model. `sys0` must be a numeric (non-identified) `tf`, `zpk`, or `ss` model object. Also, `sys0` must have at least as many inputs as outputs.

## Object Description

An `idss` model represents a system as a continuous-time or discrete-time state-space model with identifiable (estimable) coefficients.

A state-space model of a system with input vector  $u$ , output vector  $y$ , and disturbance  $e$  takes the following form in continuous time:

$$\begin{aligned}\frac{dx(t)}{dt} &= Ax(t) + Bu(t) + Ke(t) \\ y(t) &= Cx(t) + Du(t) + e(t).\end{aligned}$$

In discrete time, the state-space model takes the form:

$$\begin{aligned}x[k+1] &= Ax[k] + Bu[k] + Ke[k] \\ y[k] &= Cx[k] + Du[k] + e[k].\end{aligned}$$

For `idss` models, the elements of the state-space matrices  $A$ ,  $B$ ,  $C$ , and  $D$  can be estimable parameters. The elements of the state disturbance  $K$  can also be estimable parameters. The `idss` model stores the values of these matrix elements in the `a`, `b`, `c`, `d`, and `k` properties of the model.

There are three ways to obtain an `idss` model.

- Estimate the `idss` model based on input-output measurements of a system, using `n4sid` or `ssest`. These estimation commands estimate the values of the estimable elements of the state-space matrices. The estimated values are stored in the `a`, `b`, `c`, `d`, and `k` properties of the resulting `idss` model. The `Report` property of the resulting model stores information about the estimation, such as handling of initial state values and options used in estimation.

When you obtain an `idss` model by estimation, you can extract estimated coefficients and their uncertainties from the model using commands such as `idssdata`, `getpar`, or `getcov`.

- Create an `idss` model using the `idss` command.

You can create an `idss` model to configure an initial parameterization for estimation of a state-space model to fit measured response data. When you do so, you can specify constraints on one or more of the state-space matrix elements. For example, you can fix the values of some elements, or specify minimum or maximum values for the free elements. You can then use the configured model as an input argument to an estimation command (`n4sid` or `ssest`) to estimate parameter values with those constraints.

- Convert an existing dynamic system model to an `idss` model using the `idss` command.

To configure an `idss` model in a desired form, such as a companion or modal form, use state transformation commands such as `canon` and `ss2ss`.

## Examples

### Create State-Space Model with Identifiable Parameters

Create a 4th-order SISO state-space model with identifiable parameters. Initialize the initial state values to 0.1 for all entries. Set the sampling time to 0.1 s as well.

```
A = blkdiag([-0.1 0.4; -0.4 -0.1],[-1 5; -5 -1]);
B = [1; zeros(3,1)];
C = [1 0 1 0];
D = 0;
K = zeros(4,1);
x0 = [0.1,0.1,0.1,0.1];
Ts = 0.1;

sys = idss(A,B,C,D,K,x0,Ts);
```

`sys` is a 4th-order, SISO `idss` model. The number of states and input-output dimensions are determined by the dimensions of the

state-space matrices. By default, all entries in the matrices A, B, C, D, and K are identifiable parameters.

You can use `sys` to specify an initial parametrization for state-space model estimation with `ssest` or `n4sid`.

## Specify Additional Attributes of State-Space Model

Create a 4th-order SISO state-space model with identifiable parameters. Name the input and output channels of the model, and specify minutes for the model time units.

You can use `Name, Value` pair arguments to specify additional model properties on model creation.

```
A = blkdiag([-0.1 0.4; -0.4 -0.1],[-1 5; -5 -1]);
B = [1; zeros(3,1)];
C = [1 0 1 0];
D = 0;

sys = idss(A,B,C,D,'InputName','Drive','TimeUnit','minutes');
```

To change or specify most attributes of an existing model, you can use dot notation. For example:

```
sys.OutputName = 'Torque';
```

## Configure Identifiable Parameters of State-Space Model

Configure an `idss` model so that it has no state disturbance element and only the non-zero entries of the A matrix are estimable. Additionally, fix the values of the B matrix.

You can configure individual parameters of an `idss` model to specify constraints for state-space model estimation with `ssest` or `n4sid`.

Create an `idss` model.

```
A = blkdiag([-0.1 0.4; -0.4 -0.1],[-1 5; -5 -1]);
B = [1; zeros(3,1)];
C = [1 0 1 0];
```

```
D = 0;
K = zeros(4,1);
x0 = [0.1,0.1,0.1,0.1];

sys = idss(A,B,C,D,K,x0,0);
```

Setting all entries of `K = 0` creates an `idss` model with no state disturbance element.

Use the `Structure` property of the model to fix the values of some of the parameters.

```
sys.Structure.a.Free = (A~=0);
sys.Structure.b.Free = false;
sys.Structure.k.Free = false;
```

The entries in `sys.Structure.a.Free` determine whether the corresponding entries in `sys.a` are free (identifiable) or fixed. The first line sets `sys.Structure.a.Free` to a logical matrix that is true wherever `A` is non-zero, and false everywhere else. Doing so fixes the value of the zero entries in `sys.a`.

The remaining lines fix all the values in `sys.b` and `sys.k` to the values you specified when you created the model.

## Array of State-Space Models

Create an array of state-space models.

There are several ways to create arrays of state-space models:

- Direct array construction using  $n$ -dimensional state-space arrays
- Array-building by indexed assignment
- Array-building using the `stack` command
- Sampling an identified model using the `rsample` command

Create an array by providing  $n$ -dimensional arrays as an input argument to `idss`, instead of 2-dimensional matrices.

```
A = rand(2,2,3,4);  
sysarr = idss(A,[2;1],[1 1],0);
```

When you provide a multi-dimensional array to `idss` in place of one of the state-space matrices, the first two dimensions specify the numbers of states, inputs, or outputs of each model in the array. The remaining dimensions specify the dimensions of the array itself. `A` is a 2-by-2-by-3-by-4 array. Therefore, `sysarr` is a 3-by-4 array of `idss` models. Each model in `sysarr` has two states, specified by the first two dimensions of `A`. Further, each model in `sysarr` has the same `B`, `C`, and `D` values.

Create an array by indexed assignment.

```
sysarr = idss(zeros(1,1,2));  
sysarr(:,:,1) = idss([4 -3;-2 0],[2;1],[1 1],0);  
sysarr(:,:,2) = idss(rand(2),rand(2,1),rand(1,2),1);
```

The first command preallocates the array. The first two dimensions of the array are the I/O dimensions of each model in the array. Therefore, `sysarr` is a 2-element vector of SISO models.

The remaining commands assign an `idss` model to each position in `sysarr`. Each model in an array must have the same I/O dimensions.

Add another model to `sysarr` using `stack`.

`stack` is an alternative to building an array by indexing.

```
sysarr = stack(1,sysarr,idss([1 -2;-4 9],[0;-1],[1 1],0));
```

This command adds another `idss` model along the first array dimension of `sysarr`. `sysarr` is now a 3-by-1 array of SISO `idss` models

## Input Arguments

### **A,B,C,D**

Initial values of the state-space matrices.

For a system with  $N_y$  outputs,  $N_u$  inputs, and  $N_x$  states, specify initial values of the state-space matrix elements as follows:



- A —  $N_x$ -by- $N_x$  matrix.
- B —  $N_x$ -by- $N_u$  matrix.
- C —  $N_y$ -by- $N_x$  matrix.
- D —  $N_y$ -by- $N_u$  matrix.

Use NaN for any matrix element whose initial value is not known.

### **K**

Initial value of the state disturbance matrix.

Specify K as an  $N_x$ -by- $N_y$  matrix.

Use NaN for any matrix element whose initial value is not known.

**Default:**  $N_x$ -by- $N_y$  zero matrix.

### **x0**

Initial state values.

Specify the initial condition as a column vector of  $N_x$  values.

**Default:**  $N_x$  column vector of zeros.

### **Ts**

Sampling time. For continuous-time models,  $T_s = 0$ . For discrete-time models,  $T_s$  is a positive scalar representing the sampling period expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sampling time, set  $T_s = -1$ .

**Default:**  $-1$  (discrete-time model with unspecified sampling time)

### **sys0**

Dynamic system.

Any dynamic system, such as `idtf` or `idpoly`, to convert to an `idss` model.

For the syntax `sys = idss(sys0, 'split')`, `sys0` must be a numeric (non-identified) `tf`, `zpk`, or `ss` model object. Also, `sys0` must have at least as many inputs as outputs. Finally, the subsystem `sys0(:, Ny+1:Ny+Nu)` must contain a non-zero feedthrough term (the subsystem must be biproper).

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Use `Name`, `Value` arguments to specify additional properties of `idss` models during model creation. For example, `idss(A,B,C,D, 'InputName', 'Voltage')` creates an `idss` model with the `InputName` property set to `Voltage`.

## Properties

Specify optional comma-separated pairs of `Name`, `Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

`idss` objects properties include:

### **a,b,c,d**

Values of state-space matrices.

- `a` — State matrix  $A$ , an  $N_x$ -by- $N_x$  matrix.
- `b` —  $N_x$ -by- $N_u$  matrix.
- `c` —  $N_y$ -by- $N_x$  matrix.
- `d` —  $N_y$ -by- $N_u$  matrix.

If you create an `idss` model `sys` using the `idss` command, `sys.a`, `sys.b`, `sys.c`, and `sys.d` contain the initial values of the state-space matrices that you specify with the `A,B,C,D` input arguments.

If you obtain an `idss` model `sys` by identification using `ssest` or `n4sid`, then `sys.a`, `sys.b`, `sys.c`, and `sys.d` contain the estimated values of the matrix elements.

For an `idss` model `sys`, each property `sys.a`, `sys.b`, `sys.c`, and `sys.d` is an alias to the corresponding `Value` entry in the `Structure` property of `sys`. For example, `sys.a` is an alias to the value of the property `sys.Structure.a.Value`.

### **k**

Value of state disturbance matrix  $K$ , an  $N_x$ -by- $N_y$  matrix.

If you create an `idss` model `sys` using the `idss` command, `sys.k` contains the initial values of the state-space matrices that you specify with the `K` input argument.

If you obtain an `idss` model `sys` by identification using `ssest` or `n4sid`, then `sys.k` contains the estimated values of the matrix elements.

For an `idss` model `sys`, `sys.k` is an alias to the value of the property `sys.Structure.k.Value`.

**Default:**  $N_x$ -by- $N_y$  zero matrix.

### **StateName**

State names. Set `StateName` to a string for first-order models, or to a cell array of strings for models with two or more states. Use an empty string `''` for unnamed states.

**Default:** Empty string `''` for all states

### **StateUnit**

State units. Use `StateUnit` to keep track of the units each state is expressed in. Set `StateUnit` to a string for first-order models, or to a

cell array of strings for models with two or more states. `StateUnit` has no effect on system behavior.

**Default:** Empty string '' for all states

## Structure

Information about the estimable parameters of the `idss` model. `Structure.a`, `Structure.b`, `Structure.c`, `Structure.d`, and `Structure.k` contain information about the  $A$ ,  $B$ ,  $C$ ,  $D$ , and  $K$  matrices, respectively. Each contains the following fields:

- **Value** — Parameter values. For example, `sys.Structure.a.Value` contains the initial or estimated values of the  $A$  matrix.

NaN represents unknown parameter values.

Each property `sys.a`, `sys.b`, `sys.c`, and `sys.d` is an alias to the corresponding `Value` entry in the `Structure` property of `sys`. For example, `sys.a` is an alias to the value of the property `sys.Structure.a.Value`

- **Minimum** — Minimum value that the parameter can assume during estimation. For example, `sys.Structure.k.Minimum = 0` constrains all entries in the  $K$  matrix to be greater than or equal to zero.
- **Maximum** — Maximum value that the parameter can assume during estimation.
- **Free** — Boolean specifying whether the parameter is a free estimation variable. If you want to fix the value of a parameter during estimation, set the corresponding `Free = false`. For example, if  $A$  is a 3-by-3 matrix, `sys.Structure.a.Free = eyes(3)` fixes all of the off-diagonal entries in  $A$ , to the values specified in `sys.Structure.a.Value`. In this case, only the diagonal entries in  $A$  are estimable.
- **Scale** — Scale of the parameter's value. `Scale` is not used in estimation.

- **Info** — Structure array for storing parameter units and labels. The structure has `Label` and `Unit` fields.

Use these fields for your convenience, to store strings that describe parameter units and labels.

### **NoiseVariance**

The variance (covariance matrix) of the model innovations  $e$ .

An identified model includes a white, Gaussian noise component  $e(t)$ . `NoiseVariance` is the variance of this noise component. Typically, the model estimation function (such as `ssest`) determines this variance.

For SISO models, `NoiseVariance` is a scalar. For MIMO models, `NoiseVariance` is a  $N_y$ -by- $N_y$  matrix, where  $N_y$  is the number of outputs in the system.

### **Report**

Information about the estimation process.

`Report` contains the following fields:

- `N4Weight` — Subspace algorithm option value used by `n4sid` estimator (see `n4sidOptions`).
- `N4Horizon` — Forward and backward prediction horizons used by `n4sid` (see `n4sidOptions`).
- `InitialState` — Whether initial state values were estimated or fixed.
- `Fit` — Quantitative quality assessment of estimation, including percent fit to data and final prediction error.
- `Parameters` — Estimated values of model parameters and initial states, and their covariances.
- `OptionsUsed` — Options used during estimation (see `ssestOptions` or `n4sidOptions`).
- `RandState` — Random number stream state at start of estimation.

- **Status** — Whether model was obtained by construction, estimated, or modified after estimation.
- **Method** — Name of estimation method used.
- **DataUsed** — Attributes of data used for estimation, such as name and sampling time.
- **Termination** — Termination conditions for the iterative search scheme used for prediction error minimization, such as final cost value or stopping criterion. Available only when the model is estimated using `ssest` or `pem`.

## **InputDelay**

Input delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. For continuous-time systems, specify input delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify input delays in integer multiples of the sampling period `Ts`. For example, `InputDelay = 3` means a delay of three sampling periods.

For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector, where each entry is a numerical value representing the input delay for the corresponding input channel. You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0 for all input channels

## **OutputDelay**

Output delays.

For identified systems, like `idss`, `OutputDelay` is fixed to zero.

## **Ts**

Sampling time. For continuous-time models, `Ts = 0`. For discrete-time models, `Ts` is a positive scalar representing the sampling period expressed in the unit specified by the `TimeUnit` property of the model.

To denote a discrete-time model with unspecified sampling time, set  $T_s = -1$ .

Changing this property does not discretize or resample the model. Use `c2d` and `d2c` to convert between continuous- and discrete-time representations. Use `d2d` to change the sampling time of a discrete-time system.

**Default:** `-1` (discrete-time model with unspecified sampling time)

### **TimeUnit**

String representing the unit of the time variable, any time delays in the model (for continuous-time models), and the sampling time  $T_s$  (for discrete-time models). `TimeUnit` can take the following values:

- `'nanoseconds'`
- `'microseconds'`
- `'milliseconds'`
- `'seconds'`
- `'minutes'`
- `'hours'`
- `'days'`
- `'weeks'`
- `'months'`
- `'years'`

Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** `'seconds'`

### **InputName**

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

Alternatively, use automatic vector expansion to assign input names for multi-input models. For example, if `sys` is a two-input model, enter:

```
sys.InputName = 'controls';
```

The software automatically expands the input names to `{'controls(1)'; 'controls(2)'}`.

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `sys.u` is equivalent to `sys.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** Empty string '' for all input channels

## InputUnit

Input channel units. Use `InputUnit` to keep track of input signal units. Set `InputUnit` to a string for single-input model, or to a cell array of strings for a multi-input model. `InputUnit` has no effect on system behavior.

**Default:** Empty string '' for all input channels

## InputGroup

Input channel groups. The `InputGroup` property lets you assign the input channels of MIMO systems into groups and refer to each group by name. Specify input groups as a structure whose field names are the group names and whose field values are the input channels belong to each group. For example:

```
sys.InputGroup.controls = [1 2];
```



```
sys.InputGroup.noise = [3 5];
```

creates input groups named `controls` and `noise` that include input channels 1, 2 and 3, 5, respectively. You can then extract the subsystem from the `controls` inputs to all outputs using:

```
sys(:, 'controls')
```

**Default:** Struct with no fields

### OutputName

Output channel names. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

Alternatively, use automatic vector expansion to assign output names for multi-output models. For example, if `sys` is a two-output model, enter:

```
sys.OutputName = 'measurements';
```

The software automatically expands the output names to `{'measurements(1)'; 'measurements(2)'}`.

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `sys.y` is equivalent to `sys.OutputName`.

Output channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** Empty string `''` for all input channels

### OutputUnit

Output channel units. Use `OutputUnit` to keep track of output signal units. Set `OutputUnit` to a string for single-input model, or to a cell

array of strings for a multi-input model. `OutputUnit` has no effect on system behavior.

**Default:** Empty string '' for all input channels

## **OutputGroup**

Output channel groups. The `OutputGroup` property lets you assign the output channels of MIMO systems into groups and refer to each group by name. Specify output groups as a structure whose field names are the group names and whose field values are the output channels belong to each group. For example:

```
sys.OutputGroup.temperature = [1];  
sys.InputGroup.measurement = [3 5];
```

creates output groups named `temperature` and `measurement` that include output channels 1, and 3, 5, respectively. You can then extract the subsystem from all inputs to the measurement outputs using:

```
sys('measurement',:)
```

**Default:** Struct with no fields

## **Name**

System name. Set `Name` to a string to label the system.

**Default:** ''

## **Notes**

Any text that you wish to associate with the system. Set `Notes` to a string or a cell array of strings.

**Default:** {}

## **UserData**

Any type of data you wish to associate with system. Set UserData to any MATLAB data type.

**Default:** []

**See Also**

`idssdata` | `sstest` | `sstestOptions` | `n4sid` | `pem` | `idgrey` | `idpoly` | `idproc` | `idtf`

**Concepts**

- “Dynamic System Models”

# idssdata

---

**Purpose** State-space data of identified system

**Syntax**

```
[A,B,C,D,K] = idssdata(sys)
[A,B,C,D,K,x0] = idssdata(sys)
[A,B,C,D,K,x0,dA,dB,dC,dD,dK,dx0] = idssdata(sys)
[A,B,C,D,K, ___ ] = idssdata(sys,j1,...,jN)
[A,B,C,D,K, ___ ] = idssdata(sys,'cell')
```

**Description** [A,B,C,D,K] = idssdata(sys) returns the A,B,C,D and K matrices of the identified state-space model **sys**.

[A,B,C,D,K,x0] = idssdata(sys) returns the initial state values, **x0**.

[A,B,C,D,K,x0,dA,dB,dC,dD,dK,dx0] = idssdata(sys) returns the uncertainties in the system matrices for **sys**.

[A,B,C,D,K, \_\_\_ ] = idssdata(sys,j1,...,jN) returns data for the **j1, ..., jn** entries in the model array **sys**.

[A,B,C,D,K, \_\_\_ ] = idssdata(sys,'cell') returns data for all the entries in the model array **sys** as separate cells in cell arrays.

## Input Arguments

### **sys**

Identified model.

If **sys** is not an identified state-space model (**idssor idgrey**), then it is first converted to an **idss** model.

**sys** may be an array of identified models.

### **j1,...,jN**

Integer indices of **N** entries in the array **sys** of identified systems.

## Output Arguments

### **A,B,C,D,K**

State-space matrices that represent **sys** as:

$$x[k+1] = Ax[k] + Bu[k] + Ke[k]; x[0] = x0;$$

$$y[k] = Cx[k] + Du[k] + e[k];$$

If **sys** is an array of identified models, then **A, B, C, D, K** are multi-dimension arrays. To access the state-space matrix, say **A**, for the  $k$ -th entry of **sys**, use **A(:, :, k)**.

### **x0**

Initial state.

If **sys** is an **idss** or **idgrey** model, then **x0** is the value obtained during estimation. It is also stored using the **Report.Parameters** property of **sys**.

For other model types, **x0** is zero.

If **sys** is an array of identified models, then **x0** contains a column for each entry in **sys**.

### **dA, dB, dC, dD, dK**

Uncertainties associated with the state-space matrices **A, B, C, D, K**.

The uncertainty matrices represents 1 standard deviation of uncertainty.

If **sys** is an array of identified models, then **dA, dB, dC, dD, dK** are multi-dimension arrays. To access the state-space matrix, say **A**, for the  $k$ -th entry of **sys**, use **A(:, :, k)**.

### **dx0**

Uncertainty associated with the initial state.

**dx0** represents 1 standard deviation of uncertainty.

If **sys** is an array of identified models, then **dx0** contains a column for each entry in **sys**.

## Examples

### Obtain Identified State-Space Matrices

Obtain the identified state-space matrices for a model estimated from data.

Identify a model using data.

```
load icEngine.mat
data = iddata(y,u,0.04);
sys = n4sid(data,4,'InputDelay',2);
```

`data` is an `iddata` object representing data sampled at a sampling rate of 0.04 seconds.

`sys` is an `idss` model representing the identified system.

Obtain identified state-space matrices of `sys`.

```
[A,B,C,D,K] = idssdata(sys);
```

`A,B,C,D` and `K` represent the state-space matrices of the identified model `sys`.

### Obtain Initial State of Identified Model

Obtain the initial state associated with an identified model.

Identify a model using data.

```
load icEngine.mat
data = iddata(y,u,0.04);
sys = n4sid(data,4,'InputDelay',2);
```

`data` is an `iddata` object representing data sampled at a sampling rate of 0.04 seconds.

`sys` is an `idss` model representing the identified system.

Obtain the initial state associated with `sys`.

```
[A,B,C,D,K,x0] = idssdata(sys);
```

A,B,C,D and K represent the state-space matrices of the identified model `sys`.

`x0` is the initial state identified for `sys`.

### **Obtain Uncertainty Data of State-Space Matrices of Identified Model**

Obtain the uncertainty matrices of the state-space matrices of an identified model.

Identify a model using data.

```
load icEngine.mat
data = iddata(y,u,0.04);
sys = n4sid(data,4,'InputDelay',2);
```

`data` is an `iddata` object representing data sampled at a sampling rate of 0.04 seconds.

`sys` is an `idss` model representing the identified system.

Obtain the uncertainty matrices associated with the state-space matrices of `sys`.

```
[A,B,C,D,K,x0,dA,dB,dC,dD,dx0] = idssdata(sys);
```

`dA,dB,dC,dD` and `dK` represent the uncertainty associated with the state-space matrices of the identified model `sys`.

`dx0` represents the uncertainty associated with the estimated initial state.

### **Obtain State-Space Matrices for Multiple Identified Models**

Obtain the state-space matrices for multiple models from an array of identified models.

Identify multiple models using data.

```
load icEngine.mat
data = iddata(y,u,0.04);
```

```
sys2 = n4sid(data,2,'InputDelay',2);  
sys3 = n4sid(data,3,'InputDelay',2);  
sys4 = n4sid(data,4,'InputDelay',2);  
sys = stack(1,sys2,sys3,sys4);
```

`data` is an `iddata` object representing data sampled at a sampling rate of 0.04 seconds.

`sys` is an array of `idss` models. The first entry of `sys` is a second order identified system. The second and third entries of `sys` are third and fourth order identified systems, respectively.

Obtain the state-space matrices for the first and third entries of `sys`.

```
[A,B,C,D,K,x0] = idssdata(sys,1,3);
```

## **Obtain State-Space Matrices for Identified Model as Cell Array**

Obtain the state-space matrices of an array of identified models in cell arrays.

Identify multiple models using data.

```
load icEngine.mat  
data = iddata(y,u,0.04);  
sys3 = n4sid(data,3,'InputDelay',2);  
sys4 = n4sid(data,4,'InputDelay',2);  
sys = stack(1,sys3,sys4);
```

`data` is an `iddata` object representing data sampled at a sampling rate of 0.04 seconds.

`sys` is an array of `idss` models. The first entry of `sys` is a third order identified system and the second entry is a fourth order identified system.

Obtain the state-space matrices of `sys` in cell arrays.

```
[A,B,C,D,K,x0] = idssdata(sys,'cell');
```



A,B,C,D and K are cell arrays containing the state-space matrices of the individual entries of the identified model arraysys.

x0 is a cell array containing the estimated initial state of the individual entries of the identified model array sys.

**See Also**

[ssdata](#) | [idss](#) | [tfdata](#) | [zpkdata](#) | [polydata](#)

# idtf

---

**Purpose** Transfer function model with identifiable parameters

**Syntax**

```
sys = idtf(num,den)
sys = idtf(num,den,Ts)
sys = idtf( ___,Name,Value)

sys = idtf(sys0)
```

**Description** `sys = idtf(num,den)` creates a continuous-time transfer function with identifiable parameters (an `idtf` model). `num` specifies the current values of the transfer function numerator coefficients. `den` specifies the current values of the transfer function denominator coefficients.

`sys = idtf(num,den,Ts)` creates a discrete-time transfer function with identifiable parameters. `TS` is the sampling time.

`sys = idtf( ___,Name,Value)` creates a transfer function with properties specified by one or more `Name, Value` pair arguments.

`sys = idtf(sys0)` converts any dynamic system model, `sys0`, to `idtf` model form.

**Object Description** An `idtf` model represents a system as a continuous-time or discrete-time transfer function with identifiable (estimable) coefficients.

A SISO transfer function is a ratio of polynomials with an exponential term. In continuous time,

$$G(s) = e^{-\tau s} \frac{b_n s^n + b_{n-1} s^{n-1} + \dots + b_0}{s^m + a_{m-1} s^{m-1} + \dots + a_0}.$$

In discrete time,

$$G(z^{-1}) = z^{-k} \frac{b_n z^{-n} + b_{n-1} z^{-n+1} + \dots + b_0}{z^{-m} + a_{m-1} z^{-m+1} + \dots + a_0}.$$

In discrete time,  $z^{-k}$  represents a time delay of  $kT_s$ , where  $T_s$  is the sampling time.

For `idtf` models, the denominator coefficients  $a_0, \dots, a_{m-1}$  and the numerator coefficients  $b_0, \dots, b_n$  can be estimable parameters. (The leading denominator coefficient is always fixed to 1.) The time delay  $\tau$  (or  $k$  in discrete time) can also be an estimable parameter. The `idtf` model stores the polynomial coefficients  $a_0, \dots, a_{m-1}$  and  $b_0, \dots, b_n$  in the `den` and `num` properties of the model, respectively. The time delay  $\tau$  or  $k$  is stored in the `ioDelay` property of the model.

A MIMO transfer function contains a SISO transfer function corresponding to each input-output pair in the system. For `idtf` models, the polynomial coefficients and transport delays of each input-output pair are independently estimable parameters.

There are three ways to obtain an `idtf` model.

- Estimate the `idtf` model based on input-output measurements of a system, using `tfest`. The `tfest` command estimates the values of the transfer function coefficients and transport delays. The estimated values are stored in the `num`, `den`, and `ioDelay` properties of the resulting `idtf` model. The `Report` property of the resulting model stores information about the estimation, such as handling of initial conditions and options used in estimation.

When you obtain an `idtf` model by estimation, you can extract estimated coefficients and their uncertainties from the model using commands such as `tfdata`, `getpar`, or `getcov`.

- Create an `idtf` model using the `idtf` command.

You can create an `idtf` model to configure an initial parameterization for estimation of a transfer function to fit measured response data. When you do so, you can specify constraints on such values as the numerator and denominator coefficients and transport delays. For example, you can fix the values of some parameters, or specify

minimum or maximum values for the free parameters. You can then use the configured model as an input argument to `tfest` to estimate parameter values with those constraints.

- Convert an existing dynamic system model to an `idtf` model using the `idtf` command.

## Examples

### Continuous-Time Transfer Function

Specify a continuous-time, single-input, single-output (SISO) transfer function with estimable parameters. The initial values of the transfer function are:

$$G(s) = \frac{s + 4}{s^2 + 20s + 5}$$

```
num = [1 4];  
den = [1 20 5];  
G = idtf(num,den);
```

`G` is an `idtf` model. `num` and `den` specify the initial values of the numerator and denominator polynomial coefficients in descending powers of  $s$ . The numerator coefficients having initial values 1 and 4 are estimable parameters. The denominator coefficient having initial values 20 and 5 are also estimable parameters. The leading denominator coefficient is always fixed to 1.

You can use `G` to specify an initial parametrization for estimation with `tfest`.

### Transfer Function with Known Input Delay and Specified Attributes

Specify a continuous-time, SISO transfer function with known input delay. The transfer function initial values are given by:

$$G(s) = e^{-5.8s} \frac{5}{s + 5}$$

Label the input of the transfer function with the name 'Voltage' and specify the input units as volt.

Use Name, Value input pairs to specify the delay, input name, and input unit.

```
num = 5;
den = [1 5];
input_delay = 5.8;
input_name = 'Voltage';
input_unit = 'volt';
G = idtf(num,den,'InputDelay',input_delay,...
         'InputName',input_name,'InputUnit',input_unit);
```

$G$  is an idtf model. You can use  $G$  to specify an initial parametrization for estimation with tfest. If you do so, model properties such as InputDelay, InputName, and InputUnit are applied to the estimated model. Estimation treats InputDelay as a fixed value. If you want to estimate the delay and specify an initial value of 5.8 s, use the ioDelay property instead.

## Discrete-Time Transfer Function

Specify a discrete-time SISO transfer function with estimable parameters. The initial values of the transfer function are:

$$H(z) = \frac{z - 0.1}{z + 0.8}$$

Specify the sampling time as 0.2 seconds.

```
num = [1 -0.1];
den = [1 0.8];
Ts = 0.2
H = idtf(num,den,Ts);
```

num and den are the initial values of the numerator and denominator polynomial coefficients. For discrete-time systems, specify the coefficients in ascending powers of  $z^{-1}$ .

Ts specifies the sampling time for the transfer function as 0.2 seconds.

H is an `idtf` model. The numerator and denominator coefficients are estimable parameters (except for the leading denominator coefficient, which is fixed to 1).

### MIMO Discrete-Time Transfer Function

Specify a discrete-time, two-input, two-output transfer function. The initial values of the MIMO transfer function are:

$$H(z) = \begin{bmatrix} \frac{1}{z+0.2} & \frac{z}{z+0.7} \\ \frac{-z+2}{z-0.3} & \frac{3}{z+0.3} \end{bmatrix}$$

Specify the sampling time as 0.2 seconds.

```
nums = {1, [1,0]; [-1,2], 3};
dens = {[1,0.2], [1,0.7]}; [1, -0.3], [1, 0.3]};
Ts = 0.2
H = idtf(nums,dens,Ts);
```

`nums` and `dens` specify the initial values of the coefficients in cell arrays. Each entry in the cell array corresponds to the numerator or denominator of the transfer function of one input-output pair. For example, the first row of `nums` is `{1, [1,0]}`. This specifies the numerators across the first row of transfer functions in H. Likewise, the first row of `dens`, `{[1,0.2], [1,0.7]}`, specifies the denominators across the first row of H.

Ts specifies the sampling time for the transfer function as 0.2 seconds.

H is an `idtf` model. All of the polynomial coefficients are estimable parameters, except for the leading coefficient of each denominator polynomial. These coefficients are always fixed to 1.

### Specify $q^{-1}$ as Transfer Function Variable

Specify the following discrete-time transfer function in terms of  $q^{-1}$ :

$$H(q^{-1}) = \frac{1 + .4q^{-1}}{1 + .1q^{-1} - .3q^{-2}}$$

Specify the sampling time as 0.1 seconds.

```
num = [1 .4];
den = [1 .1 -.3];
Ts = 0.1;
convention_variable = 'q^-1';
H = idtf(num,den,Ts,'Variable',convention_variable);
```

Use a Name, Value pair argument to specify the variable  $q^{-1}$ .

num and den are the numerator and denominator polynomial coefficients in ascending powers of  $q^{-1}$ .

Ts specifies the sampling time for the transfer function as 0.1 seconds.

H is an idtf model.

### Gain Matrix Transfer Function

Specify a transfer function with estimable coefficients whose initial value is the static gain matrix:

$$H(s) = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 3 & 0 & 2 \end{bmatrix}$$

```
M = [1 0 1; 1 1 0; 3 0 2];
H = idtf(M);
```

H is an idtf model that describes a three input (Nu=3), three output (Ny=3) transfer function. Each input/output channel is an estimable static gain. The initial values of the gains are given by the values in the matrix M.

## Convert Identifiable State-Space Model to Identifiable Transfer Function

Convert a state-space model with identifiable parameters to a transfer function with identifiable parameters.

Convert the following identifiable state-space model to an identifiable transfer function.

$$\begin{aligned}\tilde{x}(t) &= \begin{bmatrix} -0.2 & 0 \\ 0 & -0.3 \end{bmatrix} x(t) + \begin{bmatrix} -2 \\ 4 \end{bmatrix} u(t) + \begin{bmatrix} .1 \\ .2 \end{bmatrix} e(t) \\ y(t) &= [1 \quad 1]x(t)\end{aligned}$$

```
A = [-0.2, 0; 0, -0.3]; B = [2;4]; C=[1, 1]; D = 0; K = [.1; .2];  
sys0 = idss(A,B,C,D,K,'NoiseVariance',0.1);  
sys = idtf(sys0);
```

A,B,C,D and K are matrices that specify sys0, an identifiable state-space model with a noise variance of 0.1.

sys = idtf(sys0) creates an idtf model, sys.

## Obtain a Transfer Function by Estimation

Identify a transfer function containing a specified number of poles for given data.

Load time domain system response data and use it to identify a transfer function for the system.

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

z1 is an iddata object that contains time domain input-output data.

np specifies the number of poles in the identified transfer function.

sys is an idtf model containing the identified transfer function.



To see the numerator and denominator coefficients of the resulting estimated model `sys`, enter:

```
sys.num  
sys.den
```

### **Obtain a Transfer Function with Prior Knowledge of Model Structure and Constraints**

Identify a transfer function for given data by providing its expected structure and coefficient constraints

Load time domain data.

```
load iddata1 z1;  
z1.y = cumsum(z1.y);
```

This command integrates the output data of `z1`. The identified transfer function should therefore contain an integrator.

Create a transfer function model with the expected structure.

```
init_sys = idtf([100 1500],[1 10 10 0]);
```

`int_sys` is an `idtf` model with three poles and one zero. The denominator coefficient for the  $s^0$  term is zero. Therefore, `int_sys` contains an integrator.

Specify constraints on the numerator and denominator coefficients of the transfer function model. To do so, configure fields in the `Structure` property of

```
init_sys.Structure.num.Minimum = eps;  
init_sys.Structure.den.Minimum = eps;  
init_sys.Structure.den.Free(end) = false;
```

The constraints specify that the numerator and denominator coefficients are nonnegative. Additionally, the last element of the denominator coefficients (associated with the  $s^0$  term) is not a free estimation variable. This constraint forces one of the estimated poles to be at  $s = 0$ .

Create an estimation option set that specifies using the Levenberg–Marquardt search method.

```
opt = tfestOptions('SearchMethod', 'lm');
```

Estimate a transfer function for `z1` using `init_sys` and the estimation option set.

```
sys = tfest(z1,init_sys,opt);
```

`tfest` uses the coefficients of `init_sys` to initialize the estimation of `sys`. Additionally, the estimation is constrained by the constraints you specify in the `Structure` property of `init_sys`. The resulting `idtf` model `sys` contains the parameter values that result from the estimation.

### **Array of Transfer Function Models**

Create an array of transfer function models with identifiable coefficients. Each transfer function in the array is of the form:

$$H(s) = \frac{a}{s + a}.$$

The initial value of the coefficient  $a$  varies across the array, from 0.1 to 1.0, in increments of 0.1.

```
H = idtf(zeros(1,1,10));  
for k = 1:10  
    num = k/10;  
    den = [1 k/10];  
    H(:,:,k) = idtf(num,den);  
end
```

The first command pre-allocates a one-dimensional, 10-element array `H` and fills it with empty `idtf` models.

The first two dimensions of a model array are the output and input dimensions. The remaining dimensions are the array dimensions. `H(:,:,k)` represents the  $k$ th model in the array. Thus, the `for` loop

replaces the  $k$ th entry in the array with a transfer function whose coefficients are initialized with  $a = k/10$ .

## Input Arguments

### **num**

Initial values of transfer function numerator coefficients.

For SISO transfer functions, specify the initial values of the numerator coefficients **num** as a row vector. Specify the coefficients in order of:

- Descending powers of  $s$  or  $p$  (for continuous-time transfer functions).
- Ascending powers of  $z^{-1}$  or  $q^{-1}$  (for discrete-time transfer functions).

Use NaN for any coefficient whose initial value is not known.

For MIMO transfer functions with  $N_y$  outputs and  $N_u$  inputs, **num** is a  $N_y$ -by- $N_u$  cell array of the numerator coefficients for each input/output pair.

### **den**

Initial values of transfer function denominator coefficients.

For SISO transfer functions, specify the initial values of the denominator coefficients **den** as a row vector. Specify the coefficients in order of:

- Descending powers of  $s$  or  $p$  (for continuous-time transfer functions).
- Ascending powers of  $z^{-1}$  or  $q^{-1}$  (for discrete-time transfer functions).

The leading coefficient in **den** must be 1. Use NaN for any coefficient whose initial value is not known.

For MIMO transfer functions with  $N_y$  outputs and  $N_u$  inputs, **den** is a  $N_y$ -by- $N_u$  cell array of the numerator coefficients for each input/output pair.

### **Ts**

Sampling time. For continuous-time models,  $T_s = 0$ . For discrete-time models,  $T_s$  is a positive scalar representing the sampling period

expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sampling time, set `Ts = -1`.

Changing this property does not discretize or resample the model. Use `c2d` and `d2c` to convert between continuous- and discrete-time representations. Use `d2d` to change the sampling time of a discrete-time system.

**Default:** 0 (continuous time)

## **sys0**

Dynamic system.

Any dynamic system, such as `idss` or `idpoly`, to convert to an `idtf` model.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Use `Name`, `Value` arguments to specify additional properties of `idtf` models during model creation. For example, `idtf(num,den,'InputName','Voltage')` creates an `idtf` model with the `InputName` property set to `Voltage`.

## **Properties**

`idtf` object properties include:

### **num**

Values of transfer function numerator coefficients.

If you create an `idtf` model `sys` using the `idtf` command, `sys.num` contains the initial values of numerator coefficients that you specify with the `num` input argument.

If you obtain an `idtf` model by identification using `tfest`, then `sys.num` contains the estimated values of the numerator coefficients.

For an `idtf` model `sys`, the property `sys.num` is an alias to the value of the property `sys.Structure.num.Value`.

For SISO transfer functions, the values of the numerator coefficients are stored as a row vector in order of:

- Descending powers of  $s$  or  $p$  (for continuous-time transfer functions).
- Ascending powers of  $z^{-1}$  or  $q^{-1}$  (for discrete-time transfer functions).

Any coefficient whose initial value is not known is stored as NaN.

For MIMO transfer functions with  $N_y$  outputs and  $N_u$  inputs, `num` is a  $N_y$ -by- $N_u$  cell array of the numerator coefficients for each input/output pair.

### **den**

Values of transfer function denominator coefficients.

If you create an `idtf` model `sys` using the `idtf` command, `sys.den` contains the initial values of denominator coefficients that you specify with the `den` input argument.

If you obtain an `idtf` model `sys` by identification using `tfest`, then `sys.den` contains the estimated values of the denominator coefficients.

For an `idtf` model `sys`, the property `sys.den` is an alias to the value of the property `sys.Structure.den.Value`.

For SISO transfer functions, the values of the denominator coefficients are stored as a row vector in order of:

- Descending powers of  $s$  or  $p$  (for continuous-time transfer functions).
- Ascending powers of  $z^{-1}$  or  $q^{-1}$  (for discrete-time transfer functions).

The leading coefficient in `den` is fixed to 1. Any coefficient whose initial value is not known is stored as NaN.

For MIMO transfer functions with  $N_y$  outputs and  $N_u$  inputs, `den` is a  $N_y$ -by- $N_u$  cell array of the numerator coefficients for each input/output pair.

## Variable

String specifying the transfer function display variable. `Variable` can take the following values:

- 's' — Default for continuous-time models
- 'p' — Equivalent to 's'
- 'z<sup>-1</sup>' — Default for discrete-time models
- 'q<sup>-1</sup>' — Equivalent to 'z<sup>-1</sup>'

The value of `Variable` is reflected in the display, and also affects the interpretation of the `num` and `den` coefficient vectors for discrete-time models. For `Variable` = 'z<sup>-1</sup>' or 'q<sup>-1</sup>', the coefficient vectors are ordered as ascending powers of the variable.

## ioDelay

Transport delays. `ioDelay` is a numeric array specifying a separate transport delay for each input/output pair.

If you create an `idtf` model `sys` using the `idtf` command, `sys.ioDelay` contains the initial values of the transport delay that you specify with a `Name, Value` argument pair.

If you obtain an `idtf` model `sys` by identification using `tfest`, then `sys.ioDelay` contains the estimated values of the transport delay.

For an `idtf` model `sys`, the property `sys.ioDelay` is an alias to the value of the property `sys.Structure.ioDelay.Value`.

For continuous-time systems, transport delays are expressed in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify transport are expressed as integers denoting delay of a multiple of the sampling period  $T_s$ .

For a MIMO system with  $N_y$  outputs and  $N_u$  inputs, set `ioDelay` is a  $N_y$ -by- $N_u$  array, where each entry is a numerical value representing the transport delay for the corresponding input/output pair. You can set `ioDelay` to a scalar value to apply the same delay to all input/output pairs.

**Default:** 0 for all input/output pairs

### Structure

Information about the estimable parameters of the `idtf` model. `Structure.num`, `Structure.den`, and `Structure.ioDelay` contain information about the numerator coefficients, denominator coefficients, and transport delay, respectively. Each contains the following fields:

- **Value** — Parameter values. For example, `sys.Structure.num.Value` contains the initial or estimated values of the numerator coefficients.

NaN represents unknown parameter values. For denominators, the value of the leading coefficient, specified by `sys.Structure.den.Value(1)` is fixed to 1.

For SISO models, `sys.num`, `sys.den`, and `sys.ioDelay` are aliases to `sys.Structure.num.Value`, `sys.Structure.den.Value`, and `sys.Structure.ioDelay.Value`, respectively.

For MIMO models, `sys.num{i,j}` is an alias to `sys.Structure(i,j).num.Value`, and `sys.den{i,j}` is an alias to `sys.Structure(i,j).den.Value`. Additionally, `sys.ioDelay(i,j)` is an alias to `sys.Structure(i,j).ioDelay.Value`

- **Minimum** — Minimum value that the parameter can assume during estimation. For example, `sys.Structure.ioDelay.Minimum = 0.1` constrains the transport delay to values greater than or equal to 0.1.

`sys.Structure.ioDelay.Minimum` must be greater than or equal to zero.

- **Maximum** — Maximum value that the parameter can assume during estimation.

- **Free** — Boolean specifying whether the parameter is a free estimation variable. If you want to fix the value of a parameter during estimation, set the corresponding `Free = false`. For example, `sys.Structure.den.Free = false` fixes all of the denominator coefficients in `sys` to the values specified in `sys.Structure.den.Value`.

For denominators, the value of `Free` for the leading coefficient, specified by `sys.Structure.den.Free(1)`, is always `false` (the leading denominator coefficient is always fixed to 1).

- **Scale** — Scale of the parameter's value. `Scale` is not used in estimation.
- **Info** — Structure array for storing parameter units and labels. The structure has `Label` and `Unit` fields.

Use these fields for your convenience, to store strings that describe parameter units and labels.

For a MIMO model with  $N_y$  outputs and  $N_u$  input, `Structure` is an  $N_y$ -by- $N_u$  array. The element `Structure(i, j)` contains information corresponding to the transfer function for the  $(i, j)$  input-output pair.

### **NoiseVariance**

The variance (covariance matrix) of the model innovations  $e$ .

An identified model includes a white, Gaussian noise component  $e(t)$ . `NoiseVariance` is the variance of this noise component. Typically, the model estimation function (such as `tfest`) determines this variance.

For SISO models, `NoiseVariance` is a scalar. For MIMO models, `NoiseVariance` is a  $N_y$ -by- $N_y$  matrix, where  $N_y$  is the number of outputs in the system.

### **Report**

Information about the estimation process.

`Report` contains the following fields:



- **InitMethod** — Method used to initialize model coefficients before iterative prediction error minimization.
- **N4Weight** — Subspace algorithm option value used by `n4sidestimator` (see `n4sidOptions`).
- **N4Horizon** — Forward and backward prediction horizons used by `n4sid` (see `n4sidOptions`).
- **InitialCondition** — Whether estimation estimated initial conditions or fixed them at zero.
- **Fit** — Quantitative quality assessment of estimation, including percent fit to data and final prediction error.
- **Parameters** — Estimated values of model parameters and their covariance.
- **OptionsUsed** — Options used during estimation (see `tfestOptions`).
- **RandState** — Random number stream state at start of estimation.
- **Status** — Whether model was obtained by construction, estimated, or modified after estimation.
- **Method** — Name of estimation method used.
- **DataUsed** — Attributes of data used for estimation, such as name and sampling time.
- **Termination** — Termination conditions for the iterative search scheme used for prediction error minimization, such as final cost value or stopping criterion.

### **InputDelay**

Input delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. For continuous-time systems, specify input delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify input delays in integer multiples of the sampling period `Ts`. For example, `InputDelay = 3` means a delay of three sampling periods.

For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector, where each entry is a numerical value representing the input delay for the corresponding input channel. You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

Estimation treats `InputDelay` as a fixed constant of the model. Estimation uses the `ioDelay` property for estimating time delays. To specify initial values and constraints for estimation of time delays, use `sys.Structure.ioDelay`.

**Default:** 0 for all input channels

## **OutputDelay**

Output delays.

For identified systems, like `idtf`, `OutputDelay` is fixed to zero.

## **Ts**

Sampling time. For continuous-time models, `Ts = 0`. For discrete-time models, `Ts` is a positive scalar representing the sampling period expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sampling time, set `Ts = -1`.

Changing this property does not discretize or resample the model. Use `c2d` and `d2c` to convert between continuous- and discrete-time representations. Use `d2d` to change the sampling time of a discrete-time system.

**Default:** 0 (continuous time)

## **TimeUnit**

String representing the unit of the time variable, any time delays in the model (for continuous-time models), and the sampling time `Ts` (for discrete-time models). `TimeUnit` can take the following values:

- 'nanoseconds'

- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

Alternatively, use automatic vector expansion to assign input names for multi-input models. For example, if `sys` is a two-input model, enter:

```
sys.InputName = 'controls';
```

The software automatically expands the input names to `{'controls(1)'; 'controls(2)'}`.

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `sys.u` is equivalent to `sys.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems

- Specifying connection points when interconnecting models

**Default:** Empty string '' for all input channels

### **InputUnit**

Input channel units. Use `InputUnit` to keep track of input signal units. Set `InputUnit` to a string for single-input model, or to a cell array of strings for a multi-input model. `InputUnit` has no effect on system behavior.

**Default:** Empty string '' for all input channels

### **InputGroup**

Input channel groups. The `InputGroup` property lets you assign the input channels of MIMO systems into groups and refer to each group by name. Specify input groups as a structure whose field names are the group names and whose field values are the input channels belong to each group. For example:

```
sys.InputGroup.controls = [1 2];  
sys.InputGroup.noise = [3 5];
```

creates input groups named `controls` and `noise` that include input channels 1, 2 and 3, 5, respectively. You can then extract the subsystem from the `controls` inputs to all outputs using:

```
sys(:, 'controls')
```

**Default:** Struct with no fields

### **OutputName**

Output channel names. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

Alternatively, use automatic vector expansion to assign output names for multi-output models. For example, if `sys` is a two-output model, enter:

```
sys.OutputName = 'measurements';
```

The software automatically expands the output names to `{ 'measurements(1)'; 'measurements(2) '}`.

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `sys.y` is equivalent to `sys.OutputName`.

Output channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** Empty string `''` for all input channels

### OutputUnit

Output channel units. Use `OutputUnit` to keep track of output signal units. Set `OutputUnit` to a string for single-input model, or to a cell array of strings for a multi-input model. `OutputUnit` has no effect on system behavior.

**Default:** Empty string `''` for all input channels

### OutputGroup

Output channel groups. The `OutputGroup` property lets you assign the output channels of MIMO systems into groups and refer to each group by name. Specify output groups as a structure whose field names are the group names and whose field values are the output channels belong to each group. For example:

```
sys.OutputGroup.temperature = [1];  
sys.InputGroup.measurement = [3 5];
```

creates output groups named `temperature` and `measurement` that include output channels 1, and 3, 5, respectively. You can then extract the subsystem from all inputs to the measurement outputs using:

```
sys('measurement',:)
```

**Default:** Struct with no fields

## Name

System name. Set `Name` to a string to label the system.

**Default:** ''

## Notes

Any text that you wish to associate with the system. Set `Notes` to a string or a cell array of strings.

**Default:** {}

## UserData

Any type of data you wish to associate with system. Set `UserData` to any MATLAB data type.

**Default:** []

## Tips

- Unlike `idss` and `idpoly`, `idtf` uses a trivial noise model, with  $H = 1$  in

$$y = Gu + He$$

and does not let you parameterize the noise.

## See Also

`tfdata` | `getcov` | `getpar` | `idpoly` | `idss` | `idproc` | `idfrd`  
| `oe` | `tfest`

## Concepts

- “Dynamic System Models”

**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  $T_s$ , then

`Datf.Frequency = [0:df:F]`, where  $F$  is  $\pi/T_s$  if  $N$  is odd and  $F = \pi/T_s \cdot (1 - 1/N)$  if  $N$  is even.

**See Also** `iddata` | `fft`

# impulse

---

**Purpose** Impulse response plot of dynamic system; impulse response data

**Syntax**

```
impulse(sys)
impulse(sys,Tfinal)
impulse(sys,t)
impulse(sys1,sys2,...,sysN)
impulse(sys1,sys2,...,sysN,Tfinal)
impulse(sys1,sys2,...,sysN,t)
[y,t] = impulse(sys)
[y,t] = impulse(sys,Tfinal)
y = impulse(sys,t)
[y,t,x] = impulse(sys)
[y,t,x,yzd] = impulse(sys)
```

**Description** `impulse` calculates the unit impulse response of a dynamic system model. For continuous-time dynamic systems, the impulse response is the response to a Dirac input  $\delta(t)$ . For discrete-time systems, the impulse response is the response to a unit area pulse of length  $T_s$  and height  $1/T_s$ , where  $T_s$  is the sampling time of the system. (This pulse approaches  $\delta(t)$  as  $T_s$  approaches zero.) For state-space models, `impulse` assumes initial state values are zero.

`impulse(sys)` plots the impulse response of the dynamic system model `sys`. This model can be continuous or discrete, and SISO or MIMO. The impulse response of multi-input systems is the collection of impulse responses for each input channel. The duration of simulation is determined automatically to display the transient behavior of the response.

`impulse(sys,Tfinal)` simulates the impulse response from  $t = 0$  to the final time  $t = T_{\text{final}}$ . Express `Tfinal` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time systems with unspecified sampling time ( $T_s = -1$ ), `impulse` interprets `Tfinal` as the number of sampling periods to simulate.

`impulse(sys,t)` uses the user-supplied time vector `t` for simulation. Express `t` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time models, `t` should be of the form `Ti:Ts:Tf`,



where  $T_s$  is the sample time. For continuous-time models,  $t$  should be of the form  $T_i:dt:T_f$ , where  $dt$  becomes the sample time of a discrete approximation to the continuous system (see “Algorithms” on page 2-490). The `impulse` command always applies the impulse at  $t=0$ , regardless of  $T_i$ .

To plot the impulse responses of several models `sys1,..., sysN` on a single figure, use:

```
impulse(sys1,sys2,...,sysN)
impulse(sys1,sys2,...,sysN,Tfinal)
impulse(sys1,sys2,...,sysN,t)
```

As with `bode` or `plot`, you can specify a particular color, linestyle, and/or marker for each system, for example,

```
impulse(sys1, 'y:', sys2, 'g- -')
```

See "Plotting and Comparing Multiple Systems" and the `bode` entry in this section for more details.

When invoked with output arguments:

```
[y,t] = impulse(sys)
[y,t] = impulse(sys,Tfinal)
y = impulse(sys,t)
```

`impulse` returns the output response  $y$  and the time vector  $t$  used for simulation (if not supplied as an argument to `impulse`). No plot is drawn on the screen. For single-input systems,  $y$  has as many rows as time samples (length of  $t$ ), and as many columns as outputs. In the multi-input case, the impulse responses of each input channel are stacked up along the third dimension of  $y$ . The dimensions of  $y$  are then

For state-space models only:

```
[y,t,x] = impulse(sys)
```

(length of  $t$ )  $\times$  (number of outputs)  $\times$  (number of inputs)

and  $y(:, :, j)$  gives the response to an impulse disturbance entering the  $j$ th input channel. Similarly, the dimensions of  $x$  are

(length of  $t$ )  $\times$  (number of states)  $\times$  (number of inputs)

`[y,t,x,yzd] = impulse(sys)` returns the standard deviation YSD of the response  $Y$  of an identified system  $SYS$ . YSD is empty if  $SYS$  does not contain parameter covariance information.

## Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

## Examples

### Example 1

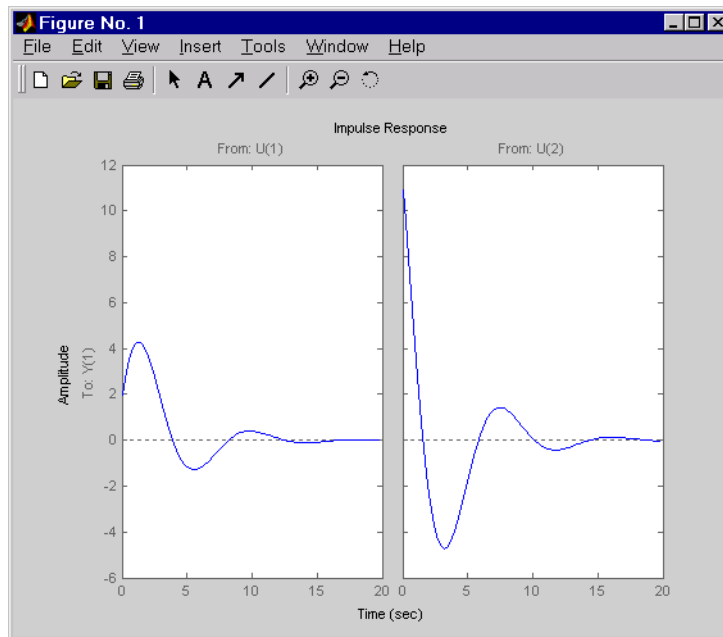
#### Impulse Response Plot of Second-Order State-Space Model

Plot the impulse response of the second-order state-space model

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -0.5572 & -0.7814 \\ 0.7814 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$
$$y = [1.9691 \quad 6.4493] \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

use the following commands.

```
a = [-0.5572 -0.7814;0.7814 0];  
b = [1 -1;0 2];  
c = [1.9691 6.4493];  
sys = ss(a,b,c,0);  
impulse(sys)
```



The left plot shows the impulse response of the first input channel, and the right plot shows the impulse response of the second input channel.

You can store the impulse response data in MATLAB arrays by

```
[y,t] = impulse(sys)
```

Because this system has two inputs, y is a 3-D array with dimensions

```
size(y)
```

```
ans =
    101     1     2
```

(the first dimension is the length of t). The impulse response of the first input channel is then accessed by

```
y(:, :, 1)
```

## Example 2

Fetch the impulse response and the corresponding 1 std uncertainty of an identified linear system.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'dcmotor')
z = iddata(y, u, 0.1, 'Name', 'DC-motor');
set(z, 'InputName', 'Voltage', 'InputUnit', 'V');
set(z, 'OutputName', {'Angular position', 'Angular velocity'});
set(z, 'OutputUnit', {'rad', 'rad/s'});
set(z, 'Tstart', 0, 'TimeUnit', 's');

model = tfest(z, 2);
[y, t, ~, ysd] = impulse(model, 2);

% Plot 3 std uncertainty
subplot(211)
plot(t, y(:, 1), t, y(:, 1) + 3 * ysd(:, 1), 'k:', t, y(:, 1) - 3 * ysd(:, 1), 'k:')
subplot(212)
plot(t, y(:, 2), t, y(:, 2) + 3 * ysd(:, 2), 'k:', t, y(:, 2) - 3 * ysd(:, 2), 'k:')
```

## Algorithms

Continuous-time models are first converted to state space. The impulse response of a single-input state-space model

$$\dot{x} = Ax + bu$$

$$y = Cx$$

is equivalent to the following unforced response with initial state  $b$ .

$$\dot{x} = Ax, \quad x(0) = b$$

$$y = Cx$$

To simulate this response, the system is discretized using zero-order hold on the inputs. The sampling period is chosen automatically based

on the system dynamics, except when a time vector  $t = 0:dt:Tf$  is supplied ( $dt$  is then used as sampling period).

## Limitations

The impulse response of a continuous system with nonzero  $D$  matrix is infinite at  $t = 0$ . `impulse` ignores this discontinuity and returns the lower continuity value  $Cb$  at  $t = 0$ .

## See Also

`ltiview` | `step` | `initial` | `lsim` | `impz` | `impzvar` | `impzvar`

# impulseest

---

**Purpose** Nonparametric impulse response estimation

**Syntax**

```
sys = impulseest(data)
sys = impulseest(data,N)
sys = impulseest(data,N,NK)
sys = impulseest(___,options)
```

**Description** `sys = impulseest(data)` estimates an impulse response model, `sys`, using time- or frequency-domain data, `data`. The model order (number of nonzero impulse response coefficients) is determined automatically using persistence of excitation analysis on the input data.

`sys = impulseest(data,N)` estimates an Nth order impulse response model, corresponding to the time range  $0 : T_s : (N-1) * T_s$ , where  $T_s$  is the data sampling time.

`sys = impulseest(data,N,NK)` specifies a transport delay of NK samples in the estimated impulse response.

`sys = impulseest(___,options)` specifies estimation options using the options set `options`.

Use nonparametric impulse response to analyze `data` for feedback effects, delays and significant time constants.

- Tips**
- To view the impulse or step response of `sys`, use either `impzplot` or `stepplot`, respectively.
  - A significant value of the impulse response of `sys` for negative time values indicates the presence of feedback in the data.
  - To view the region of insignificant impulse response (statistically zero) in a plot, right-click on the plot and select **Characteristics > Confidence Region**. A patch depicting the zero-response region appears on the plot. The impulse response at any time value is significant only if it lies outside the zero response region. The level of significance depends on the number of standard deviations specified in `ShowConfidence` or `options` in the property

editor. A common choice is 3 standard deviations, which gives 99.7% significance.

## Input Arguments

### **data**

Estimation data with at least one input signal and nonzero sample time.

For time domain estimation, **data** is an `iddata` object containing the input and output signal values.

For frequency domain estimation, **data** can be one of the following:

- Frequency response data (`frd` or `idfrd`)
- `iddata` object with its properties specified as follows:
  - `InputData` — Fourier transform of the input signal
  - `OutputData` — Fourier transform of the output signal
  - `Domain` — 'Frequency'

### **N**

Order of the FIR model. Must be one of the following:

- A positive integer.

For data containing  $N_u$  inputs and  $N_y$  outputs, you can also specify **N** as an  $N_y$ -by- $N_u$  matrix of positive integers, such that  $N(i,j)$  represents the length of impulse response from input  $j$  to output  $i$ .

- `[]` — Determines the order automatically using persistence of excitation analysis on the input data.

### **NK**

Transport delay in the estimated impulse response, specified as a scalar integer. For data containing  $N_u$  inputs and  $N_y$  outputs, you can also specify a  $N_y$ -by- $N_u$  matrix.

- To generate the impulse response coefficients for negative time values, which is useful for feedback analysis, use a negative integer.

If you specify a negative value, the value must be the same across all output channels.

You can also use `NK = 'negative'` to automatically pick negative lags for all input/output channels of the model.

- Specify `NK = 0` if the delay is unknown. The true delay is then be indicated by insignificant impulse response values in the beginning of the response.
- Specify `NK = 1` to create a system whose leading numerator coefficient is zero.

Positive values of `NK` greater than 1 are stored in the `ioDelay` property of `sys` (`sys.ioDelay = max(NK-1,0)`). Negative values are stored in the `InputDelay` property.

The impulse response (input `j` to output `i`) coefficients correspond to the time span  $NK(i,j) \cdot T_s : T_s : (N(i,j) + NK(i,j) - 1) \cdot T_s$ .

**Default:** `zeros(Ny, Nu)`

## **options**

Estimation options that specify the following:

- Prefilter order
- Regularization algorithm
- Input and output data offsets

Use `impulseestOptions` to create the options set.

## **Output Arguments**

### **sys**

Estimated impulse response model.

`sys` is an `idtf` model, which encapsulates an FIR model.



## Examples

### Identify Nonparametric Impulse Response Model from Data

Compute a nonparametric impulse response model using data from a hair dryer. The input is the voltage applied to the heater and the output is the heater temperature. Use the first 500 samples for estimation.

```
load dry2
ze = dry2(1:500);
sys = impulseest(ze);
```

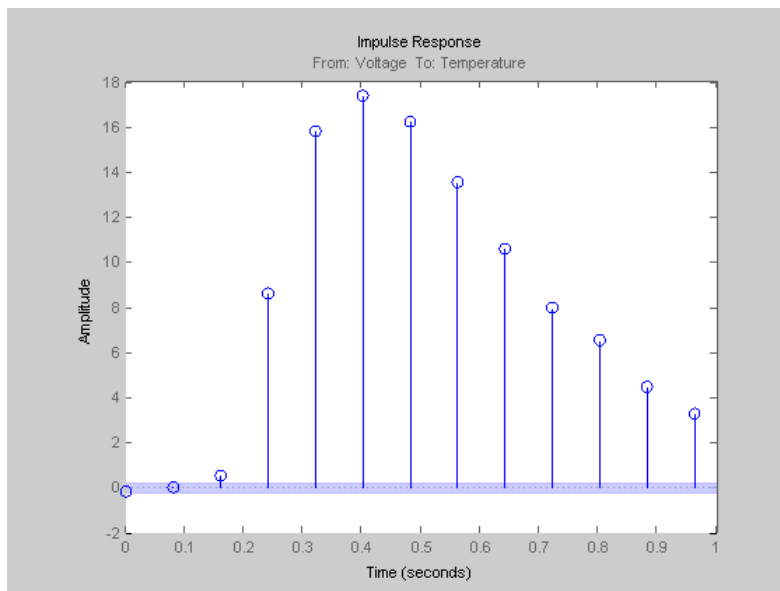
`ze` is an `iddata` object that contains time-domain data.

`sys`, the identified nonparametric impulse response model, is an `idtf` model.

Analyze the impulse response of the identified model from time 0 to time 1.

```
impzplot(sys,1);
```

Right-click the plot and select **Characteristics > Confidence Region** to view the statistically zero-response region.



The first significantly nonzero response value occurs at 0.24 seconds, or, the third lag. This implies that the transport delay is 3 samples. To generate a model where the 3-sample delay is imposed, set the transport delay to 3:

```
sys = impulseest(ze,[],3)
```

### Specify Order of FIR Model

Estimate an impulse response model with a specific order.

```
load iddata3 z3  
sys = impulseest(z3,35);
```

### Specify Transport Delay in FIR Model

Estimate an impulse response model with transport delay of 10 samples.

```
load iddata3 z3
```

```
sys = impulseest(z3,35,-10);
```

## **Obtain Regularized Estimate of Impulse Response Model**

Obtain regularized estimates of impulse response model using the regularizing kernel estimation option.

Estimate a model using regularization.

```
load iddata3 z3;  
sys1 = impulseest(z3);
```

By default, tuned and correlated kernel ('TC') is used for regularization.

Estimate a model with no regularization.

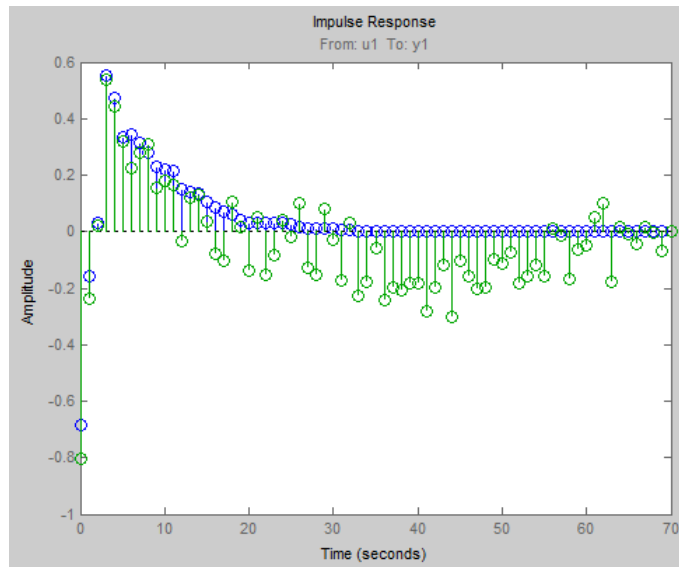
```
opt = impulseestOptions('RegulKernel','none');  
sys2 = impulseest(z3,opt);
```

Compare the impulse response of both models.

```
h = impulseplot(sys1,sys2,70);
```

# impulseest

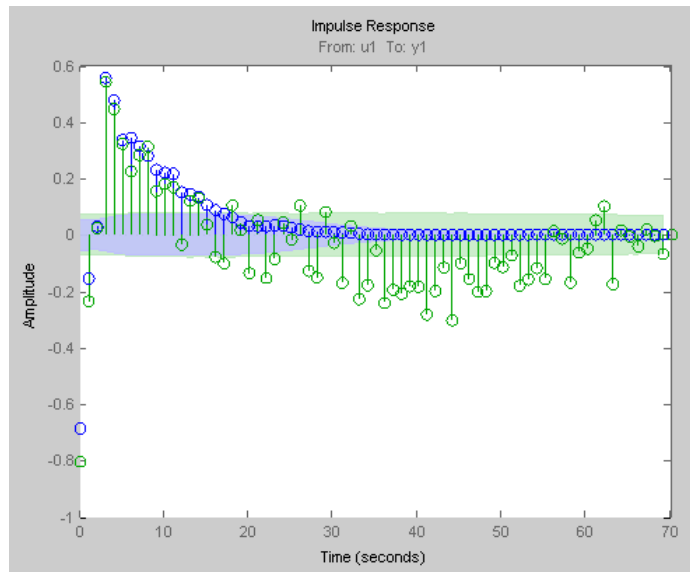
---



As the plot shows, using regularization makes the response smoother.

Plot the confidence interval.

```
showConfidence(h);
```



The uncertainty in the computed response is reduced at larger lags for the model using regularization. Regularization decreases variance at the price of some bias. The tuning of the regularization is such that the bias is dominated by the variance error though.

### Test Measured Data for Feedback Effects

Use the empirical impulse response of the measured data to verify whether there are feedback effects. Significant amplitude of the impulse response for negative time values indicates feedback effects in data.

Compute the noncausal impulse response using a fourth-order prewhitening filter, automatically chosen order and negative lag using nonregularized estimation.

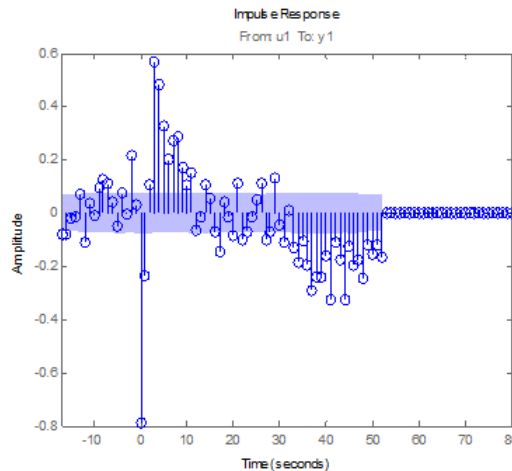
```
load iddata3 z3;
opt = impulseestOptions('pw',4,'RegulKernel','none');
sys = impulseest(z3,[],'negative',opt);
```

`sys` is a noncausal model containing response values for negative time.

Analyze the impulse response of the identified model.

```
impzplot(sys);
```

View the statistically zero-response region by right-clicking on the plot and selecting **Characteristics > Confidence Region**.



The large response value at  $t=0$  (zero lag) suggests that the data comes from a process containing feedthrough. That is, the input affects the output instantaneously. It could also be that there is a direct feedback effect (proportional control without some delay that  $u(t)$  is determined partly by  $y(t)$ ).

Also, the response values are significant for some negative time lags, such as at  $-7$  seconds and  $-9$  seconds. Such significant negative values suggest the possibility of feedback in the data.

## Compute Impulse Response on Frequency Response Data

Compute an impulse response model for frequency response data.

```
load demofr;  
zfr = AMP.*exp(1i*PHA*pi/180);
```

```
Ts = 0.1;  
data = idfrd(zfr,W,Ts);  
sys = impulseest(data);
```

## **Compare Identified Nonparametric and Parametric Models**

Identify parametric and nonparametric models for a data set, and compare their step response.

Identify the impulse response model (nonparametric) and state-space model (parametric), based on a data set.

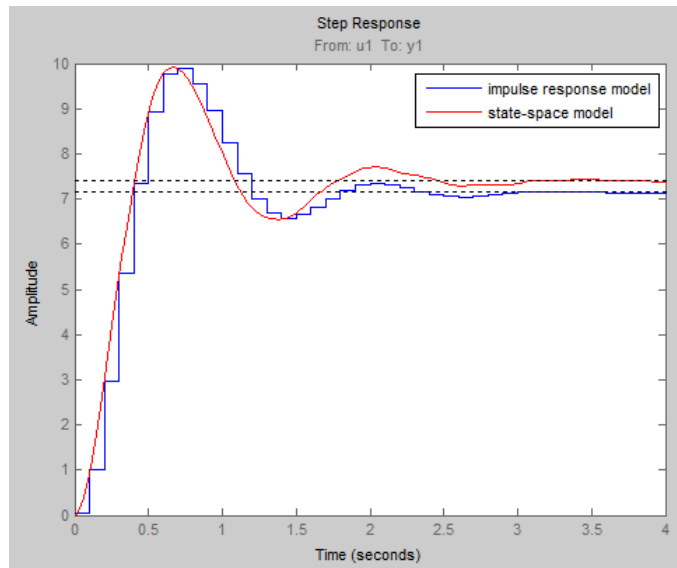
```
load iddata1 z1;  
sys1 = impulseest(z1);  
sys2 = ssest(z1,4);
```

sys1 is a discrete-time identified transfer function model.

sys2 is a continuous-time identified state-space model.

Compare the step response for sys1 and sys2.

```
step(sys1,'b',sys2,'r');  
legend('impulse response model','state-space model');
```



## Algorithms

*Correlation analysis* refers to methods that estimate the impulse response of a linear model, without specific assumptions about model orders.

The impulse response,  $g$ , is the system's output when the input is an impulse signal. The output response to a general input,  $u(t)$ , is obtained as the convolution with the impulse response. In continuous time:

$$y(t) = \int_{-\infty}^t g(\tau)u(t-\tau)d\tau$$

In discrete-time:

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

The values of  $g(k)$  are the *discrete time impulse response coefficients*.



You can estimate the values from observed input-output data in several different ways. `impulseest` estimates the first  $n$  coefficients using the least-squares method to obtain a finite impulse response (FIR) model of order  $n$ .

Several important options are associated with the estimate:

- **Prewhitening** — The input can be pre-whitened by applying an input-whitening filter of order `PW` to the data. This minimizes the effect of the neglected tail ( $k > n$ ) of the impulse response.

- 1 A filter of order `PW` is applied such that it whitens the input signal  $u$ :

$$1/A = A(u)e, \text{ where } A \text{ is a polynomial and } e \text{ is white noise.}$$

- 2 The inputs and outputs are filtered using the filter:

$$uf = Au, yf = Ay$$

- 3 The filtered signals  $uf$  and  $yf$  are used for estimation.

You can specify prewhitening using the `PW` name-value pair argument of `impulseestOptions`.

- **Regularization** — The least-squares estimate can be regularized. This means that a prior estimate of the decay and mutual correlation among  $g(k)$  is formed and used to merge with the information about  $g$  from the observed data. This gives an estimate with less variance, at the price of some bias. You can choose one of the several kernels to encode the prior estimate.

This option is essential because, often, the model order  $n$  can be quite large. In cases where there is no regularization,  $n$  can be automatically decreased to secure a reasonable variance.

You can specify the regularizing kernel using the `RegulKernel` Name-Value pair argument of `impulseestOptions`.

- **Autoregressive Parameters** — The basic underlying FIR model can be complemented by `NA` autoregressive parameters, making it an ARX model.

# impulseest

---

$$y(t) = \sum_{k=1}^n g(k)u(t-k) - \sum_{k=1}^{NA} a_k y(t-k)$$

This gives both better results for small  $n$  and allows unbiased estimates when data are generated in closed loop. `impulseest` uses  $NA = 5$  for  $t > 0$  and  $NA = 0$  (no autoregressive component) for  $t < 0$ .

- **Noncausal effects** — Response for negative lags. It may happen that the data has been generated partly by output feedback:

$$u(t) = \sum_{k=0}^{\infty} h(k)y(t-k) + r(t)$$

where  $h(k)$  is the impulse response of the regulator and  $r$  is a setpoint or disturbance term. The existence and character of such feedback  $h$  can be estimated in the same way as  $g$ , simply by trading places between  $y$  and  $u$  in the estimation call. Using `impulseest` with an indication of negative delays, `mi = impulseest(data,nk,nb)`,  $nk < 0$ , returns a model `mi` with an impulse response

$$[h(-nk), h(-nk-1), \dots, h(0), g(1), g(2), \dots, g(nb+nk)]$$

aligned so that it corresponds to lags  $[nk, nk+1, \dots, 0, 1, 2, \dots, nb+nk]$ . This is achieved because the input delay (`InputDelay`) of model `mi` is  $nk$ .

For a multi-input multi-output system, the impulse response  $g(k)$  is an  $n_y$ -by- $n_u$  matrix, where  $n_y$  is the number of outputs and  $n_u$  is the number of inputs. The  $i$ - $j$  element of the matrix  $g(k)$  describes the behavior of the  $i$ th output after an impulse in the  $j$ th input.

## See Also

`impulseestOptions` | `impulse` | `step` | `cra` | `covf` | `spa`

## Concepts

- “What Is Time-Domain Correlation Analysis?”

## Purpose

Options set for impulseest

## Syntax

```
options = impulseestOptions  
options = impulseestOptions(Name,Value)
```

## Description

`options = impulseestOptions` creates a default options set for `impulseest`.

`options = impulseestOptions(Name,Value)` creates an options set with the options specified by one or more `Name,Value` pair arguments.

## Tips

- A linear model cannot describe arbitrary input-output offsets. Therefore, before using the data, you must either detrend it or remove the levels using `InputOffset` and `OutputOffset`. You can reintroduce the removed data during simulations by using the `InputOffset` and `OutputOffset` simulation options. For more information, see `simOptions`.

- Estimating the impulse response by specifying either `InputOffset`, `OutputOffset` or both is equivalent to detrending the data using `getTrend` and `detrend`. For example:

```
opt = impulseestOptions('InputOffset',in_off,'OutputOffset',out_off)  
impulseest(data,opt);
```

is the same as:

```
Tr = getTrend(data),  
Tr.InputOffset = in_off  
Tr.OutputOffset = out_off  
dataT = detrend(data,Tr)  
impulseest(dataT)
```

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can

# impulseestOptions

---

specify several name and value pair arguments in any order as Name1, Value1, . . . , NameN, ValueN.

## **RegulKernel**

Regularizing kernel, used for regularized estimates of impulse response for all input-output channels. Regularization reduces variance of estimated model coefficients and produces a smoother response by trading variance for bias. For more information, see [1].

Must be one of the following strings:

- 'TC' — Tuned and correlated kernel
- 'none' — No regularization is used
- 'CS' — Cubic spline kernel
- 'SE' — Squared exponential kernel
- 'SS' — Stable spline kernel
- 'HF' — High frequency stable spline kernel
- 'DI' — Diagonal kernel
- 'DC' — Diagonal and correlated kernel

**Default:** 'TC'

## **PW**

Order of the input prewhitening filter. Must be one of the following:

- 'auto' — Uses a filter of order 10 when RegulKernel is 'none'; otherwise, 0.
- Nonnegative integer

Use a nonzero value of prewhitening only for unregularized estimation (RegulKernel is 'none').

**Default:** 'auto'

## InputOffset

Input signal offset level of time-domain estimation data. Must be one of the following:

- An  $N_u$ -element column vector, where  $N_u$  is the number of inputs. For multi-experiment data, specify a  $N_u$ -by- $N_e$  matrix, where  $N_e$  is the number of experiments. The offset value `InputOffset(i,j)` is subtracted from the  $i^{\text{th}}$  input signal of the  $j^{\text{th}}$  experiment.
- `[]` — No offsets.

**Default:** `[]`

## OutputOffset

Output signal offset level of time-domain estimation data. Must be one of the following:

- An  $N_y$ -element column vector, where  $N_y$  is the number of outputs. For multi-experiment data, specify a  $N_y$ -by- $N_e$  matrix, where  $N_e$  is the number of experiments. The offset value `OutputOffset(i,j)` is subtracted from the  $i^{\text{th}}$  output signal of the  $j^{\text{th}}$  experiment.
- `[]` — No offsets.

**Default:** `[]`

## Advanced

Structure with the following fields:

- `MaxSize` — Maximum allowable size of Jacobian matrices formed during estimation. Specify a large positive number.

**Default:** `250e3`

- `SearchMethod` — Search method for estimating regularization parameters. Must be one of the following strings:
  - `'gn'`: Quasi-Newton line search

# impulseestOptions

---

- 'fmincon': Trust-region-reflective constrained minimizer. Requires Optimization Toolbox software.

In general, 'fmincon' is better than 'gn' for handling bounds on regularization parameters that are imposed automatically during estimation. Thus, if you have the Optimization Toolbox software, use 'fmincon'.

SearchMethod is used only when RegulKernel is not 'none'.

**Default:** 'gn'

## Output Arguments

### options

Option set containing the specified options for impulseest.

## Examples

### Create Default Options Set for Impulse Response Estimation

Create a default options set for impulseest.

```
options = impulseestOptions;
```

### Specify Regularizing Kernel and Prewhitening Options for Impulse Response Estimation

Specify 'HF' regularizing kernel and order of prewhitening filter for impulseest.

```
options = impulseestOptions('RegulKernel','HF','PW',5);
```

Alternatively, use dot notation to specify these options.

```
options = impulseestOptions;  
options.RegulKernel = 'HF';  
options.PW = 5;
```

## References

[1] T. Chen, H. Ohlsson, and L. Ljung. "On the Estimation of Transfer Functions, Regularizations and Gaussian Processes - Revisited", *Automatica*, Volume 48, August 2012.

**See Also** impulseest

**Purpose** Plot impulse response and return plot handle

**Syntax**

```
impzplot(sys)
impzplot(sys,Tfinal)
impzplot(sys,t)
impzplot(sys1,sys2,...,sysN)
impzplot(sys1,sys2,...,sysN,Tfinal)
impzplot(sys1,sys2,...,sysN,t)
impzplot(AX,...)
impzplot(..., plotoptions)
h = impzplot(...)
```

**Description** `impzplot` plots the impulse response of the dynamic system model `sys`. For multi-input models, independent impulse commands are applied to each input channel. The time range and number of points are chosen automatically. For continuous systems with direct feedthrough, the infinite pulse at  $t=0$  is disregarded. `impzplot` can also return the plot handle, `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands. Type

```
help timeoptions
```

for a list of available plot options.

`impzplot(sys)` plots the impulse response of the LTI model without returning the plot handle.

`impzplot(sys,Tfinal)` simulates the impulse response from  $t = 0$  to the final time  $t = Tfinal$ . Express `Tfinal` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time systems with unspecified sampling time ( $T_s = -1$ ), `impzplot` interprets `Tfinal` as the number of sampling intervals to simulate.

`impzplot(sys,t)` uses the user-supplied time vector `t` for simulation. Express `t` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time models, `t` should be of the form `Ti:Ts:Tf`, where `Ts` is the sample time. For continuous-time models, `t` should be of the form `Ti:dt:Tf`, where `dt` becomes the

# impzplot

---

sample time of a discrete approximation to the continuous system (see `impz`). The `impzplot` command always applies the impulse at  $t=0$ , regardless of  $T_i$ .

To plot the impulse response of multiple LTI models `sys1,sys2,...` on a single plot, use:

```
impzplot(sys1,sys2,...,sysN)
impzplot(sys1,sys2,...,sysN,Tfinal)
impzplot(sys1,sys2,...,sysN,t)
```

You can also specify a color, line style, and marker for each system, as in

```
impzplot(sys1,'r',sys2,'y--',sys3,'gx')
```

`impzplot(AX,...)` plots into the axes with handle `AX`.

`impzplot(..., plotoptions)` plots the impulse response with the options specified in `plotoptions`. Type

```
help timeoptions
```

for more detail.

`h = impzplot(...)` plots the impulse response and returns the plot handle `h`.

## Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

## Examples

### Example 1

Normalize the impulse response of a third-order system.

```
sys = rss(3);
h = impzplot(sys);
% Normalize responses
setoptions(h,'Normalize','on');
```



## Example 2

Plot the impulse response and the corresponding 1 std "zero interval" of an identified linear system.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'dcmo...
z = iddata(y, u, 0.1, 'Name', 'DC-motor');
set(z, 'InputName', 'Voltage', 'InputUnit', 'V');
set(z, 'OutputName', {'Angular position', 'Angular velocity'});
set(z, 'OutputUnit', {'rad', 'rad/s'});
set(z, 'Tstart', 0, 'TimeUnit', 's');
model = n4sid(z,4,n4sidOptions('Focus', 'simulation'));
h = impulseplot(model,2);
showConfidence(h);
```

## See Also

[getoptions](#) | [impulse](#) | [setoptions](#) | [showConfidence](#)

# init

---

**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 linear or nonlinear identified model. It does not support `idnlgrey` models. `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  $\text{pars}(k) + e \cdot \sqrt{R(k)}$ , 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` | `rsample` | `simsd`

**Purpose** Interpolate FRD model

**Syntax** `isys = interp(sys,freqs)`

**Description** `isys = interp(sys,freqs)` interpolates the frequency response data contained in the FRD model `sys` at the frequencies `freqs`. `interp`, which is an overloaded version of the MATLAB function `interp`, uses linear interpolation and returns an FRD model `isys` containing the interpolated data at the new frequencies `freqs`. If `sys` is an IDFRD model, the noise spectrum, if non-empty, is also interpolated. The response and noise covariance data, if available, are also interpolated.

You should express the frequency values `freqs` in the same units as `sys.frequency`. The frequency values must lie between the smallest and largest frequency points in `sys` (extrapolation is not supported).

**See Also** `freqresp` | `frd` | `idfrd`

# iopzmap

---

**Purpose** Plot pole-zero map for I/O pairs of model

**Syntax** `iopzmap(sys)`  
`iopzmap(sys1,sys2,...)`

**Description** `iopzmap(sys)` computes and plots the poles and zeros of each input/output pair of the dynamic system model `sys`. The poles are plotted as x's and the zeros are plotted as o's.

`iopzmap(sys1,sys2,...)` shows the poles and zeros of multiple models `sys1,sys2,...` on a single plot. You can specify distinctive colors for each model, as in `iopzmap(sys1,'r',sys2,'y',sys3,'g')`.

The functions `sgrid` or `zgrid` can be used to plot lines of constant damping ratio and natural frequency in the  $s$  or  $z$  plane.

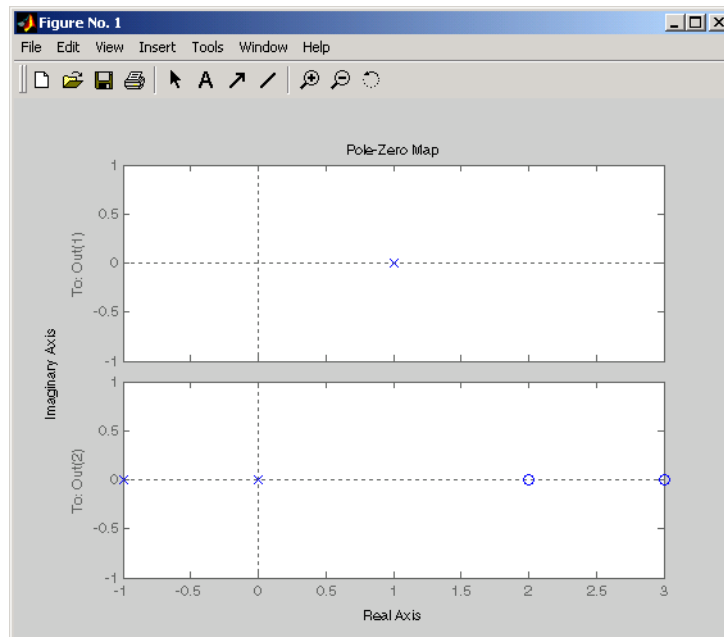
For model arrays, `iopzmap` plots the poles and zeros of each model in the array on the same diagram.

**Tips** You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

## Examples **Example 1**

Create a one-input, two-output system and plot pole-zero maps for I/O pairs.

```
H = [tf(-5,[1 -1]); tf([1 -5 6],[1 1 0])];  
iopzmap(H)
```



## Example 2

View the poles and zeros of an over-parameterized state-space model estimated using input-output data.

```
load iddata1
sys = ssest(z1,6,ssestOptions('focus','simulation'))
iopzmap(sys)
```

The plot shows that there are two pole-zero pairs that almost overlap, which hints are their potential redundancy.

## See Also

pzmap | pole | zero | sgrid | zgrid | iopzplot

# iopzplot

---

**Purpose** Plot pole-zero map for I/O pairs and return plot handle

**Syntax**

```
h = iopzplot(sys)
iopzplot(sys1,sys2,...)
iopzplot(AX,...)
iopzplot(..., plotoptions)
```

**Description** `h = iopzplot(sys)` computes and plots the poles and zeros of each input/output pair of the LTI model `SYS`. The poles are plotted as `x`'s and the zeros are plotted as `o`'s. It also returns the plot handle `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands. Type

```
help pzoptions
```

for a list of available plot options.

`iopzplot(sys1,sys2,...)` shows the poles and zeros of multiple LTI models `SYS1,SYS2,...` on a single plot. You can specify distinctive colors for each model, as in

```
iopzplot(sys1, 'r', sys2, 'y', sys3, 'g')
```

`iopzplot(AX,...)` plots into the axes with handle `AX`.

`iopzplot(..., plotoptions)` plots the poles and zeros with the options specified in `plotoptions`. Type

```
help pzoptions
```

for more detail.

The function `sgrid` or `zgrid` can be used to plot lines of constant damping ratio and natural frequency in the `s` or `z` plane.

For arrays `sys` of LTI models, `iopzplot` plots the poles and zeros of each model in the array on the same diagram.

**Tips**

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

**Examples****Example 1**

Use the plot handle to change the I/O grouping of a pole/zero map.

```
sys = rss(3,2,2);  
h = iopfplot(sys);  
% View all input-output pairs on a single axis.  
setoptions(h, 'IOGrouping', 'all')
```

**Example 2**

View the poles and zeros of an over-parameterized state-space model estimated using input-output data.

```
load iddata1  
sys = ssest(z1,6,ssestOptions('focus','simulation'));  
h = iopfplot(sys);  
showConfidence(h)
```

There is at least one pair of complex-conjugate poles whose locations overlap with those of a complex zero, within 1-std confidence region. This suggests their redundancy. Hence a lower (4th) order model might be more robust for the given data.

```
sys2 = ssest(z1,4,ssestOptions('focus','simulation'));  
h = iopfplot(sys,sys2);  
showConfidence(h)  
axis([-20, 10 -30 30])
```

The variability in the pole-zero locations of the second model `sys2` are reduced.

**See Also**

`getoptions` | `iopfmap` | `setoptions` | `showConfidence`

# isct

---

<b>Purpose</b>	Determine if dynamic system model is in continuous time
<b>Syntax</b>	<code>bool = isct(sys)</code>
<b>Description</b>	<code>bool = isct(sys)</code> returns a logical value of 1 ( <b>true</b> ) if the dynamic system model <code>sys</code> is a continuous-time model. The function returns a logical value of 0 ( <b>false</b> ) otherwise.
<b>Input Arguments</b>	<b>sys</b> Dynamic system model or array of such models.
<b>Output Arguments</b>	<b>bool</b> Logical value indicating whether <code>sys</code> is a continuous-time model. <code>bool = 1 (true)</code> if <code>sys</code> is a continuous-time model ( <code>sys.Ts = 0</code> ). If <code>sys</code> is a discrete-time model, <code>bool = 0 (false)</code> .  For a static gain, both <code>isct</code> and <code>isdT</code> return <code>true</code> unless you explicitly set the sampling time to a nonzero value. If you do so, <code>isdT</code> returns <code>true</code> and <code>isct</code> returns <code>false</code> .  For arrays of models, <code>bool</code> is <code>true</code> if the models in the array are continuous.
<b>See Also</b>	<code>isdT</code>   <code>isstable</code>



---

<b>Purpose</b>	Determine if dynamic system model is in discrete time
<b>Syntax</b>	<code>bool = isdt(sys)</code>
<b>Description</b>	<code>bool = isdt(sys)</code> returns a logical value of 1 (true) if the dynamic system model <code>sys</code> is a discrete-time model. The function returns a logical value of 0 (false) otherwise.
<b>Input Arguments</b>	<b>sys</b> Dynamic system model or array of such models.
<b>Output Arguments</b>	<b>bool</b> Logical value indicating whether <code>sys</code> is a discrete-time model. <code>bool = 1</code> (true) if <code>sys</code> is a discrete-time model ( <code>sys.Ts &gt; 0</code> ). If <code>sys</code> is a continuous-time model, <code>bool = 0</code> (false).  For a static gain, both <code>isct</code> and <code>isdt</code> return true unless you explicitly set the sampling time to a nonzero value. If you do so, <code>isdt</code> returns true and <code>isct</code> returns false.  For arrays of models, <code>bool</code> is true if the models in the array are discrete.
<b>See Also</b>	<code>isct</code>   <code>isstable</code>

# isempty

---

**Purpose** Determine whether dynamic system model is empty

**Syntax** `isempty(sys)`

**Description** `isempty(sys)` returns TRUE (logical 1) if the dynamic system model `sys` has no input or no output, and FALSE (logical 0) otherwise. Where `sys` is a FRD model, `isempty(sys)` returns TRUE when the frequency vector is empty. Where `sys` is a model array, `isempty(sys)` returns TRUE when the array has empty dimensions or when the LTI models in the array are empty.

**Examples** Both commands

```
isempty(tf) % tf by itself returns an empty transfer function  
isempty(ss(1,2,[],[]))
```

return TRUE (logical 1) while

```
isempty(ss(1,2,3,4))
```

returns FALSE (logical 0).

**See Also** `issiso` | `size`

<b>Purpose</b>	Determine if dynamic system model is proper
<b>Syntax</b>	<pre>B = isproper(sys) B = isproper(sys,'elem') [B, sysr] = isproper(sys)</pre>
<b>Description</b>	<p><code>B = isproper(sys)</code> returns TRUE (logical 1) if the dynamic system model <code>sys</code> is proper and FALSE (logical 0) otherwise.</p> <p>A proper model has relative degree <math>\leq 0</math> and is causal. SISO transfer functions and zero-pole-gain models are proper if the degree of their numerator is less than or equal to the degree of their denominator (in other words, if they have at least as many poles as zeroes). MIMO transfer functions are proper if all their SISO entries are proper. Regular state-space models (state-space models having no E matrix) are always proper. A descriptor state-space model that has an invertible E matrix is always proper. A descriptor state-space model having a singular (non-invertible) E matrix is proper if the model has at least as many poles as zeroes.</p> <p>If <code>sys</code> is a model array, then <code>B</code> is TRUE if all models in the array are proper.</p> <p><code>B = isproper(sys,'elem')</code> checks each model in a model array <code>sys</code> and returns a logical array of the same size as <code>sys</code>. The logical array indicates which models in <code>sys</code> are proper.</p> <p>If <code>sys</code> is a proper descriptor state-space model with a non-invertible E matrix, <code>[B, sysr] = isproper(sys)</code> also returns an equivalent model <code>sysr</code> with fewer states (reduced order) and a non-singular E matrix. If <code>sys</code> is not proper, <code>sysr = sys</code>.</p>
<b>Examples</b>	<p><b>Example 1</b></p> <p>The following commands</p> <pre>isproper(tf([1 0],1))           % transfer function s isproper(tf([1 0],[1 1]))      % transfer function s/(s+1)</pre>

return FALSE (logical 0) and TRUE (logical 1), respectively.

## Example 2

Combining state-space models can yield results that include more states than necessary. Use `isproper` to compute an equivalent lower-order model.

```
H1 = ss(tf([1 1],[1 2 5]));  
H2 = ss(tf([1 7],[1]));  
H = H1*H2
```

```
a =  
      x1    x2    x3    x4  
x1   -2  -2.5  0.5  1.75  
x2    2    0    0    0  
x3    0    0    1    0  
x4    0    0    0    1
```

```
b =  
      u1  
x1    0  
x2    0  
x3    0  
x4   -4
```

```
c =  
      x1    x2    x3    x4  
y1    1  0.5    0    0
```

```
d =  
      u1  
y1    0
```

```
e =  
      x1    x2    x3    x4  
x1    1    0    0    0  
x2    0    1    0    0
```

```

x3    0    0    0    0.5
x4    0    0    0    0

```

H is proper and reducible:

```
[isprop, Hr] = isproper(H)
```

```
isprop =
```

```
1
```

```
a =
```

```

           x1           x2
x1         0    0.1398
x2 -0.06988 -0.0625

```

```
b =
```

```

           u1
x1  -0.125
x2  -0.1398

```

```
c =
```

```

           x1           x2
y1  -0.5    -1.118

```

```
d =
```

```

           u1
y1    1

```

```
e =
```

```

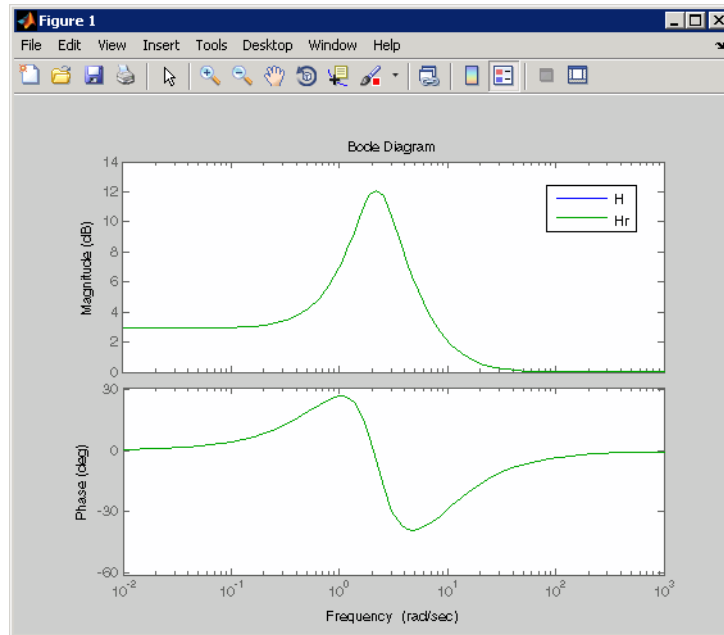
           x1           x2
x1  0.0625    0
x2    0    0.03125

```

Continuous-time model.

H and Hr are equivalent, as a Bode plot demonstrates:

bode(H, Hr)



**See Also**

ss | dss

**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`

# issiso

---

**Purpose** Determine if dynamic system model is single-input/single-output (SISO)

**Syntax** `issiso(sys)`

**Description** `issiso(sys)` returns 1 (true) if the dynamic system model `sys` is SISO and 0 (false) otherwise.

**See Also** `isempty` | `size`



**Purpose** Determine whether system is stable

**Syntax**  
`B = isstable(sys)`  
`B = isstable(sys, 'elem')`

**Description** `B = isstable(sys)` returns 1 (true) if the dynamic system model `sys` has stable dynamics, and 0 (false) otherwise. If `sys` is a model array, then `B` is true if all models in `sys` are stable.

`B = isstable(sys, 'elem')` returns a logical array of the same size as the model array `sys`. The logical array indicates which models in `sys` are stable.

`isstable` is only supported for analytical models with a finite number of poles.

**See Also** pole

**Purpose** AR model estimation using instrumental variable method

**Syntax**

```
sys = ivar(data,na)
sys = ivar(data,na,nc)
sys = ivar(data,na,nc,max_size)
```

**Description** `sys = ivar(data,na)` estimates an AR polynomial model, `sys`, using the instrumental variable method and the time series data `data`. `na` specifies the order of the  $A$  polynomial.

An AR model is represented by the equation:

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

In the above model,  $e(t)$  is an arbitrary process, assumed to be a moving average process of order `nc`, possibly time varying. `nc` is assumed to be equal to `na`. Instruments are chosen as appropriately filtered outputs, delayed `nc` steps.

`sys = ivar(data,na,nc)` specifies the value of the moving average process order, `nc`, separately.

`sys = ivar(data,na,nc,max_size)` specifies the maximum size of matrices formed during estimation.

## Input Arguments

### **data**

Estimation time series data.

`data` must be an `iddata` object with scalar output data only.

### **na**

Order of the  $A$  polynomial

### **nc**

Order of the moving average process representing  $e(t)$ .

### **max\_size**

Maximum matrix size.

`max_size` specifies the maximum size of any matrix formed by the algorithm for estimation.

Specify `max_size` as a reasonably large positive integer.

**Default:** 250000

## Output Arguments

**sys**

Identified polynomial model.

`sys` is an AR `idpoly` model which encapsulates the identified polynomial model.

## 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);
spectrum(miv,mls)
```

## References

[1] Stoica, P., et al. *Optimal Instrumental Variable Estimates of the AR-parameters of an ARMA Process*, IEEE Trans. Autom. Control, Volume AC-30, 1985, pp. 1066–1074.

## See Also

`ar` | `arx` | `etfe` | `idpoly` | `polyest` | `spa` | `step` | `spectrum`

# ivstruc

---

**Purpose** Loss functions for sets of ARX 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.

`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.

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.

```
load iddata1 z1;  
v = ivstruc(z1,z1, struc(1:3,1:2,2:4));  
nn = selstruc(v)  
m = iv4(z1,nn);
```

**Algorithms**

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** ARX model estimation using instrumental variable method with arbitrary instruments

**Syntax**  
`sys = ivx(data,[na nb nk],x)`  
`sys = ivx(data,[na nb nk],x,max_size)`

**Description** `sys = ivx(data,[na nb nk],x)` estimates an ARX polynomial model, `sys`, using the instrumental variable method with arbitrary instruments. The model is estimated for the time series data `data`. `[na nb nk]` specifies the ARX structure orders of the  $A$  and  $B$  polynomials and the input to output delay, expressed in the number of samples.

An ARX model is represented as:

$$A(q)y(t) = B(q)u(t - nk) + v(t)$$

`sys = ivx(data,[na nb nk],x,max_size)` specifies the maximum size of matrices formed during estimation.

**Tips**

- Use `iv4` first for IV estimation to identify ARX polynomial models where the instruments `x` are chosen automatically. Use `ivx` for nonstandard situations. For example, when there is feedback present in the data, or, when other instruments need to be tried. You can also use `iv` to automatically generate instruments from certain custom defined filters.

**Input Arguments**

**data**  
Estimation time series data.

`data` must be an `iddata` object and can represent either time- or frequency-domain data. If using frequency domain data, the number of outputs must be 1.

**[na nb nk]**  
ARX model orders.

For more details on the ARX model structure, see `arx`.

**x**

Instrument variable matrix.

`x` is a matrix containing the arbitrary instruments for use in the instrumental variable method.

`x` must be of the same size as the output data, `data.y`. For multi-experiment data, specify `x` as a cell array with one entry for each experiment.

The instruments used are analogous to the regression vector, with `y` replaced by `x`.

**max\_size**

Maximum matrix size.

`max_size` specifies the maximum size of any matrix formed by the algorithm for estimation.

Specify `max_size` as a reasonably large positive integer.

**Default:** 250000

**Output Arguments****sys**

Identified polynomial model.

`sys` is an ARX `idpoly` model which encapsulates the identified polynomial model.

`ivx` does not return any estimated covariance information for `sys`.

**References**

[1] Ljung, L. *System Identification: Theory for the User*, page 222, Upper Saddle River, NJ, Prentice-Hal PTR, 1999.

**See Also**

`arx` | `arxstruc` | `idpoly` | `iv4` | `ivar` | `polyest`

**Purpose** ARX model estimation using four-stage instrumental variable method.

**Syntax**

```
sys = iv4(data,[na nb nk])  
sys = iv4(data,'na',na,'nb',nb,'nk',nk)  
sys = iv4(data,[na nb nk],Name,Value)  
sys = iv4(data,[na nb nk], ___,opt)
```

**Description** `sys = iv4(data,[na nb nk])` estimates an ARX polynomial model, `sys`, using the four-stage instrumental variable method, for the data object `data`. `[na nb nk]` specifies the ARX structure orders of the  $A$  and  $B$  polynomials and the input to output delay. The estimation algorithm is insensitive to the color of the noise term.

`sys` is an ARX model:

$$A(q)y(t) = B(q)u(t - nk) + v(t)$$

Alternatively, you may also use the following syntax:

```
sys = iv4(data,'na',na,'nb',nb,'nk',nk)
```

`sys = iv4(data,[na nb nk],Name,Value)` estimates an ARX polynomial with additional options specified by one or more `Name,Value` pair arguments.

`sys = iv4(data,[na nb nk], ___,opt)` uses the option set, `opt`, to configure the estimation behavior.

## Input Arguments

### **data**

Estimation time series data.

`data` must be an `iddata` object.

### **[na nb nk]**

ARX polynomial orders.

For multi-output model, `[na nb nk]` contains one row for every output. In particular, specify `na` as an  $N_y$ -by- $N_y$  matrix, where each



entry is the polynomial order relating the corresponding output pair. Here,  $N_y$  is the number of outputs. Specify `nb` and `nk` as  $N_y$ -by- $N_u$  matrices, where  $N_u$  is the number of inputs. For more details on the ARX model structure, see `arx`.

### **opt**

Estimation options.

`opt` is an options set that configures the estimation options. These options include:

- estimation focus
- handling of initial conditions
- handling of data offsets

Use `iv4options` to create the options set.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **InputDelay**

Input delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. For continuous-time systems, specify input delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify input delays in integer multiples of the sampling period  $T_s$ . For example, `InputDelay = 3` means a delay of three sampling periods.

For a system with  $N_u$  inputs, set `InputDelay` to an  $N_u$ -by-1 vector, where each entry is a numerical value representing the input delay for the corresponding input channel. You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0 for all input channels

### **ioDelay**

Transport delays. `ioDelay` is a numeric array specifying a separate transport delay for each input/output pair.

For continuous-time systems, specify transport delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify transport delays as integers denoting delay of a multiple of the sampling period  $T_s$ .

For a MIMO system with  $N_y$  outputs and  $N_u$  inputs, set `ioDelay` to a  $N_y$ -by- $N_u$  array, where each entry is a numerical value representing the transport delay for the corresponding input/output pair. You can also set `ioDelay` to a scalar value to apply the same delay to all input/output pairs.

**Default:** 0 for all input/output pairs

## **Output Arguments**

### **sys**

Identified polynomial model of ARX structure.

`sys` is an `idpoly` model which encapsulates the identified polynomial model.

## **Examples**

Estimate 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]);
```

## **Algorithms**

Estimation is performed in 4 stages. 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

[1] Ljung, L. *System Identification: Theory for the User*, equations (15.21) through (15.26), Upper Saddle River, NJ, Prentice-Hal PTR, 1999.

## See Also

iv4options | arx | armax | bj | idpoly | ivx | n4sid | oe  
| polyst

# iv4Options

---

**Purpose** Option set for iv4

**Syntax**  
`opt = iv4options`  
`opt = iv4options(Name,Value)`

**Description** `opt = iv4options` creates the default options set for iv4.  
`opt = iv4options(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### InitialCondition

Specify handling of initial conditions during estimation.

`InitialCondition` takes one of the following values:

- 'zero' — The initial condition is set to zero.
- 'estimate' — The initial condition is treated as an independent estimation parameter.
- 'backcast' — The initial condition is estimated using the best least squares fit.
- 'auto' — `ssest` chooses the initial condition handling method based on the estimation data.

**Default:** 'auto'

### Focus

Defines how the errors  $e$  between the measured outputs and the modeled outputs are weighed at specific frequencies during the minimization of the prediction error.

Higher weighting at specific frequencies emphasizes the requirement for a good fit at these frequencies.

**Focus** can take the following values:

- '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.

This method provides a stable model.

- 'prediction' — Automatically calculates the weighting function as a product of the input spectrum and the inverse of the noise model. The weighting function 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.
- 'stability' — Same as 'prediction' but with model stability enforced.
- Passbands — Row vector or matrix containing frequency values that define desired passbands. For example:

```
[w1,wh]
[w11,w1h;w21,w2h;w31,w3h;...]
```

where  $w1$  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.

- SISO filter — Enter any SISO linear filter in any of the following ways:
  - A single-input-single-output (SISO) linear system.

- The {A,B,C,D} format, which specifies the state-space matrices of the filter.
- The {numerator, denominator} format, 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. To obtain a good model fit for a specific frequency range, you must choose the filter with a passband in this range. The estimation result is the same if you first prefilter the data using `idfilt`.

- Weighting vector — 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.

**Default:** 'prediction'

## **EstCovar**

Controls whether parameter covariance data is generated or not.

If `EstCovar` is true, then use `getcov` to fetch the covariance matrix from the estimated model.

**Default:** true

## **Display**

Specifies whether estimation progress should be displayed.

`Display` requires one of the following strings:

- 'on' — Information on model structure and estimation results are displayed in a progress viewer window
- 'off' — No progress or results information is displayed

**Default:** 'off'

**InputOffset**

Removes offset from time domain input data during estimation.

Specify as a column vector of length  $Nu$ , where  $Nu$  is the number of inputs.

Use `[]` to indicate no offset.

For multiexperiment data, specify `InputOffset` as a  $Nu$ -by- $Ne$  matrix.  $Nu$  is the number of inputs, and  $Ne$  is the number of experiments.

Each entry specified by `InputOffset` is subtracted from the corresponding input data.

**Default:** `[]`

**OutputOffset**

Removes offset from time domain output data during estimation.

Specify as a column vector of length  $Ny$ , where  $Ny$  is the number of outputs.

Use `[]` to indicate no offset.

For multiexperiment data, specify `OutputOffset` as a  $Ny$ -by- $Ne$  matrix.  $Ny$  is the number of outputs, and  $Ne$  is the number of experiments. Each entry specified by `OutputOffset` is subtracted from the corresponding output data.

**Default:** `[]`

**OutputWeight**

Weight of prediction errors in multi-output estimation.

Specify `OutputWeight` as a positive semi-definite symmetric matrix ( $W$ ). The software will minimize the trace of the weighted prediction error matrix  $\text{trace}(E' * E * W)$ . Where  $E$  is the matrix of prediction errors, with one column for each output, and  $W$  is the positive semi-definite symmetric matrix of size equal to the number of outputs. Use

W to specify the relative importance of outputs in multiple-input multiple-output models, or the reliability of corresponding data.

This option is relevant only for multi-output models.

## Advanced

Advanced is a structure with the following fields:

- **MaxSize** — Specifies the maximum number of elements in a segment when input-output data is split into segments.

MaxSize must be a positive integer.

**Default:** 250000

- **StabilityThreshold** — Specifies thresholds for stability tests.

StabilityThreshold is a structure with the following fields:

- **s** — Specifies the location of the right-most pole to test the stability of continuous-time models. A model is considered stable when its right-most pole is to the left of s.

**Default:** 0

- **z** — 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 z from the origin.

**Default:**  $1 + \sqrt{\text{eps}}$

## Output Arguments

**opt**

Option set containing the specified options for iv4.

## Examples

### Create Default Options Set for ARX Model Estimation Using 4-Stage Instrument Variable Method

Create a default options set for iv4.

```
opt = iv4options;
```



## Specify Options for ARX Model Estimation Using 4-Stage Instrument Variable Method

Create an options set for `iv4` using the 'backcast' algorithm to initialize the state and set the Display to 'on'.

```
opt = iv4Options('InitialState','backcast','Display','on');
```

Alternatively, use dot notation to set the values of `opt`.

```
opt = iv4Options;  
opt.InitialState = 'backcast';  
opt.Display = 'on';
```

**See Also** `iv4`

# linapp

---

<b>Purpose</b>	Linear approximation of nonlinear ARX and Hammerstein-Wiener models for given input
<b>Syntax</b>	<code>lm = linapp(nlmodel,u)</code> <code>lm = linapp(nlmodel,umin,umax,nsample)</code>
<b>Input Arguments</b>	<p><code>nlmodel</code> Name of the <code>idnlarx</code> or <code>idnlhw</code> model object you want to linearize.</p> <p><code>u</code> Input signal as an <code>iddata</code> object or a real matrix.</p> <p>Dimensions of <code>u</code> must match the number of inputs in <code>nlmodel</code>.</p> <p><code>[umin,umax]</code> Minimum and maximum input values for generating white-noise input with a magnitude in this rectangular range. The sample length of this signal is <code>nsample</code>.</p> <p><code>nsample</code> Optional argument when you specify <code>[umin,umax]</code>. Specifies the length of the white-noise input. Default: 1024.</p>
<b>Description</b>	<p><code>lm = linapp(nlmodel,u)</code> computes a linear approximation of a nonlinear ARX or Hammerstein-Wiener model by simulating the model output for the input signal <code>u</code>, and estimating a linear model <code>lm</code> from <code>u</code> and the simulated output signal.</p> <p><code>lm = linapp(nlmodel,umin,umax,nsample)</code> computes a linear approximation of a nonlinear ARX or Hammerstein-Wiener model by first generating the input signal as a uniformly distributed white noise from the magnitude range <code>umin</code> and <code>umax</code> and (optionally) the number of samples.</p> <p>The following table summarizes the linear model objects that store the linear approximation for each type of nonlinear model and the number of outputs.</p>

Nonlinear Model Type	Number of Outputs	Linear Model Object
idnlarx	Single output	idpoly
idnlarx	Multiple outputs	idarx
idnlhw	Single output	idpoly
idnlhw	Multiple outputs	idss

**See Also**

`idnlarx` | `idnlhw` | `findop(idnlarx)` | `findop(idnlhw)` |  
`linearize(idnlarx)` | `linearize(idnlhw)`

**How To**

- “Linear Approximation of Nonlinear Black-Box Models”

# linear

---

**Purpose** Class representing linear nonlinearity estimator for nonlinear ARX models

**Syntax** `lin=linear`  
`lin=linear('Parameters',Par)`

**Description** `linear` is an object that stores the linear nonlinearity estimator for estimating nonlinear ARX models.

`lin=linear` instantiates the `linear` object.

`lin=linear('Parameters',Par)` instantiates the `linear` object and specifies optional values in the `Par` structure. For more information about this structure, see “linear Properties” on page 2-546.

**Tips**

- `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  $x$ .
- When creating a nonlinear ARX model using the constructor (`idnlarx`) or estimator (`nlarx`), you can specify a linear nonlinearity estimator using `[]`, instead of entering `linear` explicitly. For example:

```
m=idnlarx(orders,[]);
```

## 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: <ul style="list-style-type: none"> <li>LinearCoef: <math>m</math>-by-1 vector <math>L</math>.</li> <li>OutputOffset: Scalar <math>d</math>.</li> </ul>

## Examples

Estimate a nonlinear ARX model using the `linear` estimator with custom regressors for the following system:

$$y(t) = a_1y(t-1) + a_2y(t-2) + a_3u(t-1) + a_4y(t-1)u(t-2) + a_5|u(t)|u(t-3) + a_6,$$

where  $u$  is the input and  $y$  is the output.

```
% Create regressors y(t-1), y(t-2) and u(t-1).
```

```
orders = [2 1 1];
```

```
% Create an idnlarx model using linear estimator with custom regressors
model = idnlarx(orders, linear, 'InputName', 'u', 'OutputName', 'y', ...
    'CustomRegressors', {'y(t-1)*u(t-2)', 'abs(u(t))*u(t-3)'});
```

```
% Estimate the model parameters a1, a2, ... a6.
```

```
EstimatedModel = nlarx(data, model)
```

---

**Note** The nonlinearity in the model is described by custom regressors only.

---

## Algorithms

When the `idnlarx` property `Focus` is `'Prediction'`, `linear` uses a fast, noniterative initialization and iterative search technique for estimating parameters. In most cases, iterative search requires only a few iterations.

When the `idnlarx` property `Focus` is `'Simulation'`, `linear` uses an iterative technique for estimating parameters.

# linear

---

## **Tutorials**

“How to Estimate Nonlinear ARX Models at the Command Line”

## **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 2-354.

**Input Arguments**

- `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` reference page.

---

**Note** To estimate `U0` and `X0` from operating point specifications, use the `findop(idnlarx)` command.

---

**Output Arguments**

- `SYS` is an `idss` model.  
When the Control System Toolbox product is installed, `SYS` is an LTI object.

**Algorithms**

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

# linearize(idnlarx)

---

$$y^* = f(X^*, u^*)$$

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

- $\Delta X(t) = X(t) - X^*(t)$
- $\Delta u(t) = u(t) - u^*(t)$
- $\Delta y(t) = y(t) - y^*(t)$
- $B\tilde{U} = [B_1, B_2] \begin{bmatrix} Y \\ U \end{bmatrix} = B_1 Y + B_2 U$
- $f_X = \left. \frac{\partial}{\partial X} f(X, U) \right|_{X^*, U^*}$
- $f_U = \left. \frac{\partial}{\partial U} f(X, U) \right|_{X^*, U^*}$

---

**Note** For linear approximations over larger input ranges, use `linapp`. For more information, see the `linapp` reference page.

---

## Examples

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
```

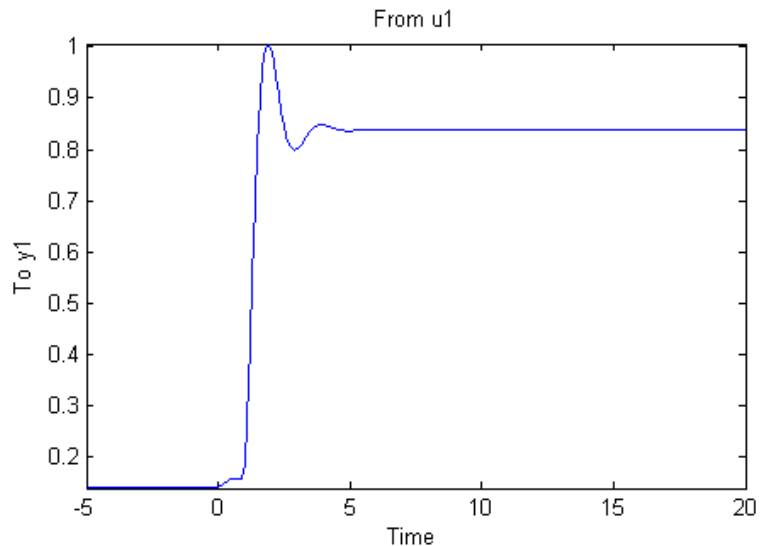


**2** Estimate idnlarx model from sample data:

```
nlsys = 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(nlsys, 20)
```



The model step response is a steady-state value of 0.8383 at  $T = 20$  seconds.

**4** Compute the operating point corresponding to  $T = 20$ .

```
stepinput = iddata([], [zeros(10,1); ones(200,1)], ...
    nlsys.Ts);
% Compute operating point.
[x,u] = findop(nlsys,'snapshot',20,stepinput);
```

# linearize(idnlarx)

---

- 5 Linearize the model about the operating point corresponding to the model snapshot at  $T=20$ .

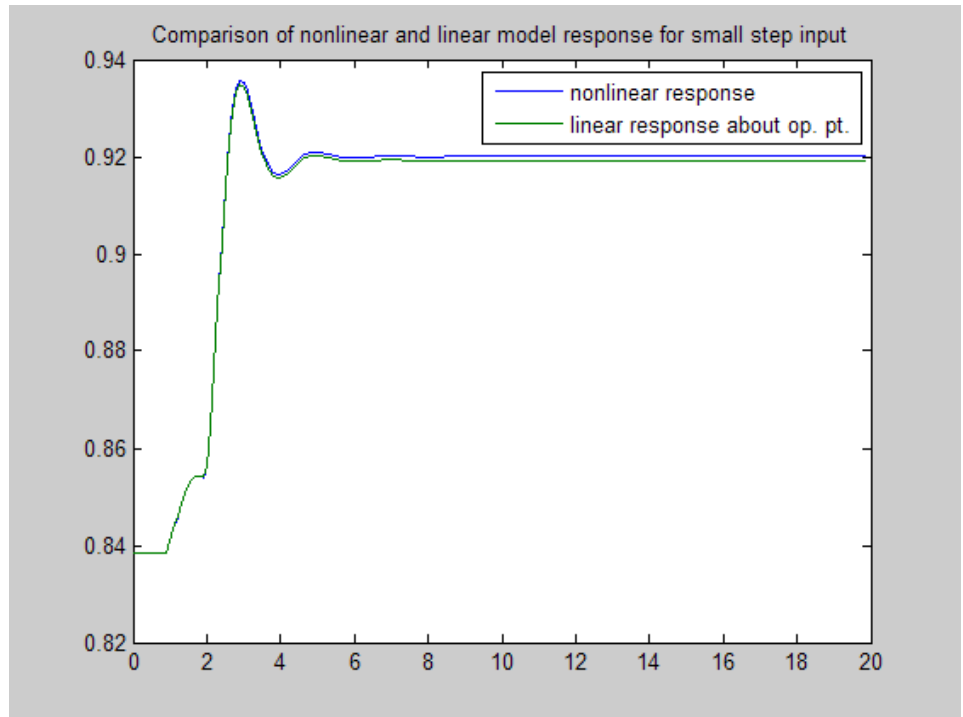
```
sys = linearize(nlsys,u,x)
```

- 6 To validate the linear model, apply a small perturbation `delta_u` to the steady-state input of the nonlinear model `nlsys`. If the linear approximation is accurate, the following should match:

- The response of the nonlinear model `y_nl` to an input that is the sum of the equilibrium level and the perturbation `delta_u`.
- The sum of the response of the linear model to a perturbation input `delta_u` and the output equilibrium level.

```
% Generate a 200-sample step signal with amplitude 0.1
% This is the perturbation signal.
delta_u = [zeros(10,1); 0.1*ones(190,1)];
%
% For a nonlinear system with a steady-state input of 1
% and a steady-state output of 0.8383,
% compute the steady-state response
% y_nl to the perturbed input u_nl. Use equilibrium state
% values x as initial conditions (see Step 4).
u_nl = 1 + delta_u;
y_nl = sim(nlsys,u_nl,x);
%
% Compute response of linear model to perturbation input
% and add it to the output equilibrium level:
y_lin = 0.8383 + lsim(sys,delta_u);
%
% Compare the response of nonlinear and linear models:
time = [0:0.1:19.9]';
plot(time,y_nl,time,y_lin)
legend('Nonlinear response',...
       'Linear response about op. pt.')
title(['Nonlinear and linear model response'...])
```

```
' for small step input'])
```



The linearized model response tracks the nonlinear model output.

## See Also

[findop\(idnlarx\)](#) | [getDelayInfo](#) | [idnlarx](#) | [linapp](#)

## How To

- “Linear Approximation of Nonlinear Black-Box Models”

# 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 around the 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` around 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 “`idnlhw` States” on page 2-389.

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 Arguments**

- `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.

---

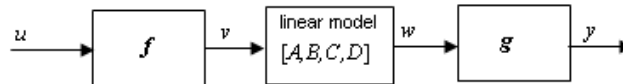
**Output Arguments**

- `SYS` is an `idss` model.

When the Control System Toolbox product is installed, `SYS` is an LTI object.

**Algorithms** 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-Wiener model around 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

- $\Delta X(t) = X(t) - X^*(t)$
- $\Delta u(t) = u(t) - u^*(t)$
- $\Delta y(t) = y(t) - y^*(t)$

# linearize(idnlhw)

---

- $f_u = \left. \frac{\partial}{\partial u} f(u) \right|_{u=u^*}$

- $g_w = \left. \frac{\partial}{\partial w} g(w) \right|_{w=w^*}$

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

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];
nlsys = 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 around the computed input and state values.

```
[x,u_s,report] = findop(nlsys,'steady',[10,5]);  
sys = linearize(nlsys,u_s,x);  
% sys is a state-space model
```

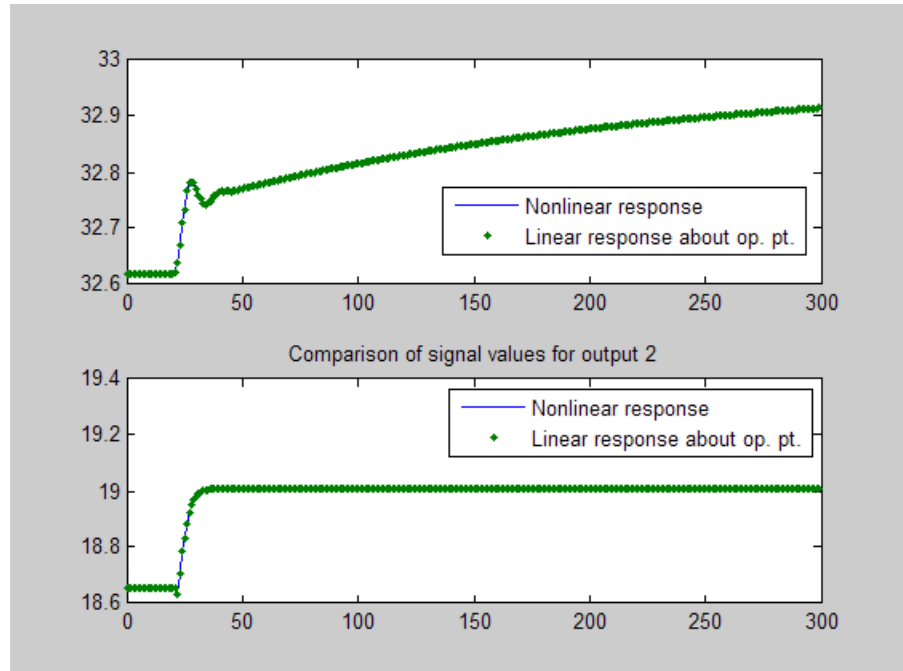
- 4** To validate the linear model, apply a small perturbation `delta_u` to the steady-state input of the nonlinear model `nlsys`. If the linear approximation is accurate, the following should match:

- The response of the nonlinear model `y_nl` to an input that is the sum of the equilibrium level and the perturbation `delta_u`.
- The sum of the response of the linear model to a perturbation input `delta_u` and the output equilibrium level.

```
% Generate a 300-sample step signal with amplitude 0.1  
% This is the perturbation input signal.  
delta_u = [zeros(20,2); 0.1*ones(280,2)];  
%  
% Compute the response of the linear model delta_y_lin  
% to the perturbed input signal delta_u:  
delta_y_lin = lsim(sys,delta_u);  
%  
% For the nonlinear system with a steady-state input u_s,  
% compute the steady-state output y_s from the  
% SignalLevels field of the findop report (see Step 3):  
y_s = report.SignalLevels.Output;  
%  
% Compute the perturbed input to the nonlinear system  
% as the sum of the steady-state input u_s and  
% the perturbation signal delta_u:  
u_nl = bsxfun(@plus,delta_u,u_s);  
  
% Compute the steady-state response of the  
% nonlinear system y_nl to the perturbed input u_nl.  
% Use equilibrium state values x as initial conditions.  
y_nl = sim(nlsys,u_nl,x);  
%  
% Compare the response of nonlinear and linear models:
```

# linearize(idnlhw)

```
time = (0:299)';  
subplot(211)  
plot(time,y_nl(:,1),time,delta_y_lin(:,1)+y_s(1),'.')  
legend('Nonlinear response',...  
       'Linear response about op. pt.')  
title('Comparison of signal values for output 1')  
  
subplot(212)  
plot(time,y_nl(:,2),time,delta_y_lin(:,2)+y_s(2),'.')  
legend('Nonlinear response',...  
       'Linear response about op. pt.')  
title('Comparison of signal values for output 2')
```



## See Also

[findop\(idnlhw\)](#) | [idnlhw](#) | [linapp](#)



## How To

- “Linear Approximation of Nonlinear Black-Box Models”

**Purpose** Simulate time response of dynamic system to arbitrary inputs

**Syntax**

```
lsim
lsim(sys,u,t)
lsim(sys,u,t,x0)
lsim(sys,u,t,x0,'zoh')
lsim(sys,u,t,x0,'foh')
lsim(sys)
```

**Description** `lsim` simulates the (time) response of continuous or discrete linear systems to arbitrary inputs. When invoked without left-hand arguments, `lsim` plots the response on the screen.

`lsim(sys,u,t)` produces a plot of the time response of the dynamic system model `sys` to the input time history `t,u`. The vector `t` specifies the time samples for the simulation (in system time units, specified in the `TimeUnit` property of `sys`), and consists of regularly spaced time samples.

```
t = 0:dt:Tfinal
```

The matrix `u` must have as many rows as time samples (`length(t)`) and as many columns as system inputs. Each row `u(i,:)` specifies the input value(s) at the time sample `t(i)`.

The LTI model `sys` can be continuous or discrete, SISO or MIMO. In discrete time, `u` must be sampled at the same rate as the system (`t` is then redundant and can be omitted or set to the empty matrix). In continuous time, the time sampling `dt=t(2)-t(1)` is used to discretize the continuous model. If `dt` is too large (undersampling), `lsim` issues a warning suggesting that you use a more appropriate sample time, but will use the specified sample time. See “Algorithms” on page 2-563 for a discussion of sample times.

`lsim(sys,u,t,x0)` further specifies an initial condition `x0` for the system states. This syntax applies only to state-space models.

`lsim(sys,u,t,x0,'zoh')` or `lsim(sys,u,t,x0,'foh')` explicitly specifies how the input values should be interpolated between samples

(zero-order hold or linear interpolation). By default, `lsim` selects the interpolation method automatically based on the smoothness of the signal `U`.

Finally,

```
lsim(sys1,sys2,...,sysN,u,t)
```

simulates the responses of several LTI models to the same input history `t,u` and plots these responses on a single figure. As with `bode` or `plot`, you can specify a particular color, linestyle, and/or marker for each system, for example,

```
lsim(sys1,'y:',sys2,'g--',u,t,x0)
```

The multisystem behavior is similar to that of `bode` or `step`.

When invoked with left-hand arguments,

```
[y,t] = lsim(sys,u,t)
[y,t,x] = lsim(sys,u,t)      % for state-space models only
[y,t,x] = lsim(sys,u,t,x0)  % with initial state
```

return the output response `y`, the time vector `t` used for simulation, and the state trajectories `x` (for state-space models only). No plot is drawn on the screen. The matrix `y` has as many rows as time samples (`length(t)`) and as many columns as system outputs. The same holds for `x` with "outputs" replaced by states.

`lsim(sys)` opens the Linear Simulation Tool GUI. For more information about working with this GUI, see [Working with the Linear Simulation Tool](#).

## Examples

### Example 1

Simulate and plot the response of the system

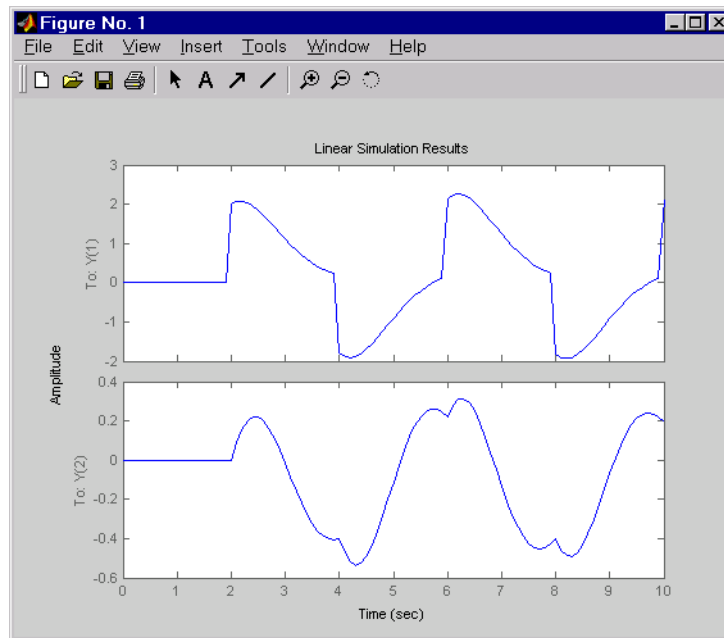
$$H(s) = \begin{bmatrix} \frac{2s^2 + 5s + 1}{s^2 + 2s + 3} \\ \frac{s - 1}{s^2 + s + 5} \end{bmatrix}$$

to a square wave with period of four seconds. First generate the square wave with gensig. Sample every 0.1 second during 10 seconds:

```
[u,t] = gensig('square',4,10,0.1);
```

Then simulate with lsim.

```
H = [tf([2 5 1],[1 2 3]) ; tf([1 -1],[1 1 5])]  
lsim(H,u,t)
```



**Example 2**

Simulate the response of an identified linear model using the same input signal as the one used for estimation and the initial states returned by the estimation command.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'dcmo
z = iddata(y, u, 0.1, 'Name', 'DC-motor');
```

```
[sys, x0] = n4sid(z, 4);
[y,t,x] = lsim(sys, z.InputData, [], x0);
```

Compare the simulated response `y` to measured response `z.OutputData`.

```
plot(t,z.OutputData,'k', t,y, 'r')
legend('Measured', 'Simulated')
```

**Algorithms**

Discrete-time systems are simulated with `ltitr` (state space) or `filter` (transfer function and zero-pole-gain).

Continuous-time systems are discretized with `c2d` using either the 'zoh' or 'foh' method ('foh' is used for smooth input signals and 'zoh' for discontinuous signals such as pulses or square waves). The sampling period is set to the spacing `dt` between the user-supplied time samples `t`.

The choice of sampling period can drastically affect simulation results. To illustrate why, consider the second-order model

$$H(s) = \frac{\omega^2}{s^2 + 2s + \omega^2}, \quad \omega = 62.83$$

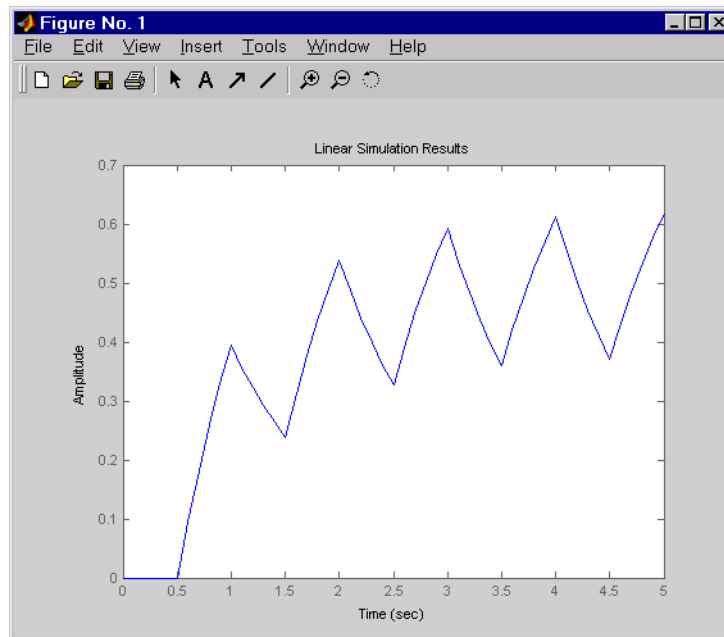
To simulate its response to a square wave with period 1 second, you can proceed as follows:

```
w2 = 62.83^2
h = tf(w2,[1 2 w2])
t = 0:0.1:5; % vector of time samples
u = (rem(t,1)>=0.5); % square wave values
```

```
lsim(h,u,t)
```

lsim evaluates the specified sample time, gives this warning

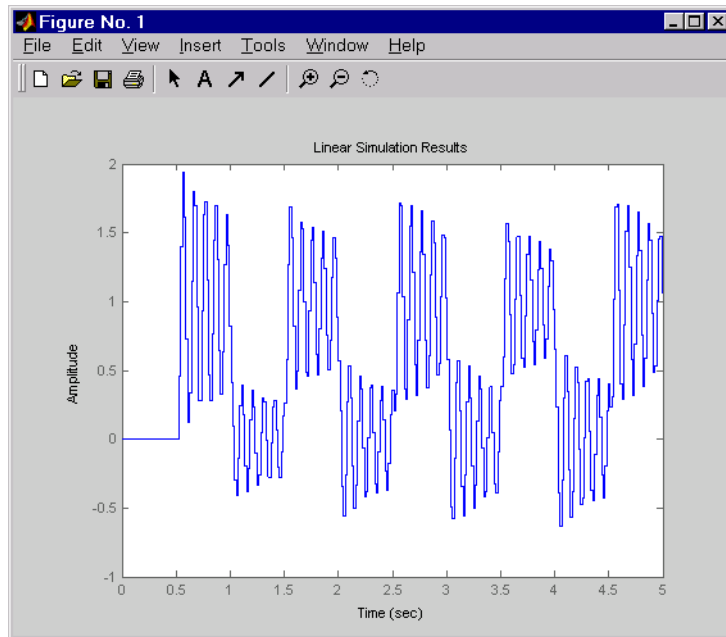
Warning: Input signal is undersampled. Sample every 0.016 sec or faster.



and produces this plot.

To improve on this response, discretize  $H(s)$  using the recommended sampling period:

```
dt=0.016;  
ts=0:dt:5;  
us = (rem(ts,1)>=0.5)  
hd = c2d(h,dt)  
lsim(hd,us,ts)
```



This response exhibits strong oscillatory behavior hidden from the undersampled version.

### See Also

`gensig` | `impulse` | `initial` | `ltiview` | `step` | `sim`

# lsiminfo

---

**Purpose** Compute linear response characteristics

**Syntax**

```
S = lsiminfo(y,t,yfinal)
S = lsiminfo(y,t)
S = lsiminfo(...,'SettlingTimeThreshold',ST)
```

**Description** `S = lsiminfo(y,t,yfinal)` takes the response data  $(t,y)$  and a steady-state value `yfinal` and returns a structure `S` containing the following performance indicators:

- `SettlingTime` — Settling time
- `Min` — Minimum value of `Y`
- `MinTime` — Time at which the min value is reached
- `Max` — Maximum value of `Y`
- `MaxTime` — Time at which the max value is reached

For SISO responses, `t` and `y` are vectors with the same length `NS`. For responses with `NY` outputs, you can specify `y` as an `NS`-by-`NY` array and `yfinal` as a `NY`-by-1 array. `lsiminfo` then returns an `NY`-by-1 structure array `S` of performance metrics for each output channel.

`S = lsiminfo(y,t)` uses the last sample value of `y` as steady-state value `yfinal`. `s = lsiminfo(y)` assumes `t = 1:NS`.

`S = lsiminfo(...,'SettlingTimeThreshold',ST)` lets you specify the threshold `ST` used in the settling time calculation. The response has settled when the error  $|y(t) - y_{\text{final}}|$  becomes smaller than a fraction `ST` of its peak value. The default value is `ST=0.02` (2%).

**Examples** Create a fourth order transfer function and ascertain the response characteristics.

```
sys = tf([1 -1],[1 2 3 4]);
[y,t] = impulse(sys);
s = lsiminfo(y,t,0) % final value is 0
s =
```



```
SettlingTime: 22.8626
  Min: -0.4270
  MinTime: 2.0309
  Max: 0.2845
  MaxTime: 4.0619
```

## See Also

`lsim` | `impulse` | `initial` | `stepinfo`

# lsimplot

---

**Purpose** Simulate response of dynamic system to arbitrary inputs and return plot handle

**Syntax**

```
h = lsimplot(sys)
lsimplot(sys1,sys2,...)
lsimplot(sys,u,t)
lsimplot(sys,u,t,x0)
lsimplot(sys1,sys2,...,u,t,x0)
lsimplot(AX,...)
lsimplot(..., plotoptions)
lsimplot(sys,u,t,x0,'zoh')
lsimplot(sys,u,t,x0,'foh')
```

**Description** `h = lsimplot(sys)` opens the Linear Simulation Tool for the dynamic system model `sys`, which enables interactive specification of driving input(s), the time vector, and initial state. It also returns the plot handle `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands. Type

```
help timeoptions
```

for a list of available plot options.

`lsimplot(sys1,sys2,...)` opens the Linear Simulation Tool for multiple models `sys1,sys2,...`. Driving inputs are common to all specified systems but initial conditions can be specified separately for each.

`lsimplot(sys,u,t)` plots the time response of the model `sys` to the input signal described by `u` and `t`. The time vector `t` consists of regularly spaced time samples (in system time units, specified in the `TimeUnit` property of `sys`). For MIMO systems, `u` is a matrix with as many columns as inputs and whose `i`th row specifies the input value at time `t(i)`. For SISO systems `u` can be specified either as a row or column vector. For example,

```
t = 0:0.01:5;
u = sin(t);
```

`lsimplot(sys,u,t)`

simulates the response of a single-input model `sys` to the input `u(t)=sin(t)` during 5 seconds.

For discrete-time models, `u` should be sampled at the same rate as `sys` (`t` is then redundant and can be omitted or set to the empty matrix).

For continuous-time models, choose the sampling period `t(2)-t(1)` small enough to accurately describe the input `u`. `lsim` issues a warning when `u` is undersampled, and hidden oscillations can occur.

`lsimplot(sys,u,t,x0)` specifies the initial state vector `x0` at time `t(1)` (for state-space models only). `x0` is set to zero when omitted.

`lsimplot(sys1,sys2,...,u,t,x0)` simulates the responses of multiple LTI models `sys1,sys2,...` on a single plot. The initial condition `x0` is optional. You can also specify a color, line style, and marker for each system, as in

`lsimplot(sys1,'r',sys2,'y--',sys3,'gx',u,t)`

`lsimplot(AX,...)` plots into the axes with handle `AX`.

`lsimplot(..., plotoptions)` plots the initial condition response with the options specified in `plotoptions`. Type

`help timeoptions`

for more detail.

For continuous-time models, `lsimplot(sys,u,t,x0,'zoh')` or `lsimplot(sys,u,t,x0,'foh')` explicitly specifies how the input values should be interpolated between samples (zero-order hold or linear interpolation). By default, `lsimplot` selects the interpolation method automatically based on the smoothness of the signal `u`.

## See Also

`getoptions` | `lsim` | `setoptions`

# mag2db

---

**Purpose** Convert magnitude to decibels (dB)

**Syntax** `ydb = mag2db(y)`

**Description** `ydb = mag2db(y)` returns the corresponding decibel (dB) value *ydb* for a given magnitude *y*. The relationship between magnitude and decibels is  $ydb = 20 \log_{10}(y)$ .

**See Also** `db2mag`

**Purpose** Merge data sets into 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

## merge (iddata)

---

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(731:1000))  
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 #8`.

### See Also

`iddata` | `getexp` | `merge`

**Purpose**

Merge estimated models

**Syntax**

```
m = merge(m1,m2,...,mN)
[m,tv] = merge(m1,m2)
```

**Description**

The models  $m_1, m_2, \dots, m_N$  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  $m_k$ .

When two models are merged,

```
[m, tv] = merge(m1,m2)
```

returns a test variable  $tv$ . It is  $\chi^2$  distributed with  $n$  degrees of freedom, if the parameters of  $m_1$  and  $m_2$  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.

For `idfrd` models, `merge` is a statistical average of two responses in the individual models, weighted using inverse variances. You can only merge two `idfrd` models with responses at the same frequencies and nonzero covariances.

Merging models is an alternative to merging data sets and estimating a model for the merged data.

```
load iddata1 z1;
load iddata2 z2;
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]);
```

## merge

---

result in models  $m_a$  and  $m_b$  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 folder 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 folder as the user's `startup.m` file.

`midprefs` is used to select or change the folder where you store `idprefs.mat`. Either type `midprefs` and follow the instructions, or give the folder name as the argument. Include all folder 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 NaNs.

`Datae` is an `iddata` object where the missing data has been replaced by reasonable estimates.

`Model` is any linear identified model (`idtf`, `idproc`, `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.

**Algorithms** 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.

**See Also** `arx` | `advice` | `pexcit` | `tfest`

**Purpose**

Estimate state-space model using a subspace method.

**Syntax**

```
sys = n4sid(data,nx)
sys = n4sid(data,nx,Name,Value)
sys = n4sid(data,nx, __ ,opt)
[sys,x0] = n4sid(data,nx, __ ,)
```

**Description**

`sys = n4sid(data,nx)` estimates an `nx` order state-space model, `sys`, using measured input-output data, `data`.

`sys` is an `idss` model representing the system:

$$\begin{aligned}\dot{x}(t) &= Ax(t) + Bu(t) + Ke(t) \\ y(t) &= Cx(t) + Du(t) + e(t)\end{aligned}$$

Here,  $A, B, C$ , and  $D$  are state-space matrices.  $K$  is the disturbance matrix.  $u(t)$  is the input,  $y(t)$  is the output,  $x(t)$  is the vector of  $nx$  states and  $e(t)$  is the disturbance.

All the entries of the  $A, B, C$  and  $K$  matrices are considered free estimation parameters.  $D$  is fixed to zero, meaning that there is no feedthrough, except for static systems ( $nx=0$ ).

`sys = n4sid(data,nx,Name,Value)` specifies additional attributes of the state-space structure using one or more `Name, Value` pair arguments.

`sys = n4sid(data,nx, __ ,opt)` specifies estimation options, `opt`, that configure the initial states, estimation objective, and sub-space algorithm related choices to be used for estimation.

`[sys,x0] = n4sid(data,nx, __ ,)` also returns the estimated initial state.

**Input Arguments****data**

Estimation data.

For time domain estimation, `data` is an `iddata` object containing the input and output signal values.

For frequency domain estimation, **data** can be one of the following:

- recorded frequency response data (`frd` or `idfrd`)
- `iddata` object with its properties specified as follows:
  - `InputData` — Fourier transform of the input signal
  - `OutputData` — Fourier transform of the output signal
  - `Domain` — 'Frequency'

For multi-experiment data, the sample times and inter-sample behavior of all the experiments must match.

You can only estimate continuous-time models using continuous-time frequency domain data. You can estimate both continuous-time and discrete-time models (of sample time matching that of **data**) using time-domain data and discrete-time frequency domain data.

## **nx**

Order of estimated model.

Specify **nx** as a positive integer. **nx** may be a scalar or a vector. If **nx** is a vector, then `n4sid` creates a plot which you can use to choose a suitable model order. The plot shows the Hankel singular values for models of different orders. States with relatively small Hankel singular values can be safely discarded. A default choice is suggested in the plot.

You can also specify **nx** as 'best', in which case the optimal order is automatically chosen from  $nx = 1, \dots, 10$ .

## **opt**

Estimation options.

**opt** is an options set that specifies estimation options including:

- estimation objective
- handling of initial conditions
- subspace algorithm related choices

Use `n4sidOptions` to create the options set.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

#### **Ts**

Sampling time. For continuous-time models, use `Ts = 0`. For discrete-time models, specify `Ts` as a positive scalar whose value is equal to that of the data sampling time.

**Default:** `data.Ts`

#### **Form**

Type of canonical form of `sys`.

`Form` is a string that takes one of the following values:

- 'modal' — obtain `sys` in modal form.
- 'companion' — obtain `sys` in companion form.
- 'free' — all entries of the  $A$ ,  $B$  and  $C$  matrices are treated as free.
- 'canonical' — obtain `sys` in observable canonical form [1].

**Default:** 'free'

#### **Feedthrough**

Logical specifying direct feedthrough from input to output.

`Feedthrough` is a logical vector of length of length  $Nu$ , where  $Nu$  is the number of inputs.

If `Feedthrough` is specified as a logical scalar, it is applied to all the inputs.

**Default:** `false(1,Nu)` ( $Nu$  is the number of inputs). If the model has no states, then `Feedthrough` is `true(1,Nu)`.

## **DisturbanceModel**

Specifies if the noise component, the  $K$  matrix, is to be estimated.

`DisturbanceModel` takes one of the following values:

- 'none' — Noise component is not estimated. The value of the  $K$  matrix, is fixed to zero value.
- 'estimate' — The  $K$  matrix is treated as a free parameter.

`DisturbanceModel` must be 'none' when using frequency domain data.

**Default:** 'estimate' (For time domain data)

## **InputDelay**

Input delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. For continuous-time systems, specify input delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify input delays in integer multiples of the sampling period  $T_s$ . For example, `InputDelay = 3` means a delay of three sampling periods.

For a system with  $Nu$  inputs, set `InputDelay` to an  $Nu$ -by-1 vector, where each entry is a numerical value representing the input delay for the corresponding input channel. You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0 for all input channels

## **Output Arguments**

### **sys**

Identified state space model.

`sys` is an `idss` model, which encapsulates the identified state space model.

**x0**

Initial states computed during the estimator of `sys`.

If `data` contains multiple experiments, then `x0` is an array with each column corresponding to an experiment.

**Definitions****Modal Form**

In modal form,  $A$  is a block-diagonal matrix. The block size is typically 1-by-1 for real eigenvalues and 2-by-2 for complex eigenvalues. However, if there are repeated eigenvalues or clusters of nearby eigenvalues, the block size can be larger.

For example, for a system with eigenvalues  $(\lambda_1, \sigma \pm j\omega, \lambda_2)$ , the modal  $A$  matrix is of the form

$$\begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \sigma & \omega & 0 \\ 0 & -\omega & \sigma & 0 \\ 0 & 0 & 0 & \lambda_2 \end{bmatrix}$$

**Companion Form**

In the companion realization, the characteristic polynomial of the system appears explicitly in the rightmost column of the  $A$  matrix. For a system with characteristic polynomial

$$p(s) = s^n + \alpha_1 s^{n-1} + \dots + \alpha_{n-1} s + \alpha_n$$

the corresponding companion  $A$  matrix is

$$A = \begin{bmatrix} 0 & 0 & \dots & \dots & 0 & -\alpha_n \\ 1 & 0 & 0 & \dots & 0 & -\alpha_n - 1 \\ 0 & 1 & 0 & \dots & \dots & \vdots \\ \vdots & 0 & \dots & \dots & \vdots & \vdots \\ 0 & \dots & \dots & 1 & 0 & -\alpha_2 \\ 0 & \dots & \dots & 0 & 1 & -\alpha_1 \end{bmatrix}$$

The companion transformation requires that the system be controllable from the first input. The companion form is poorly conditioned for most state-space computations; avoid using it when possible.

## Examples

Build a fifth-order model from data with three inputs and two outputs. Try several choices of N4Horizon. Look at the frequency response of the model.

```
z = iddata([y1 y2],[ u1 u2 u3]);
opt = n4sidOptions('N4Horizon',[7:15'],'Display','on');
m = n4sid(z, 5, opt);
h = bodeplot(m)
showConfidence(h)
```

---

Estimate a continuous-time model, in a canonical form parameterization, focusing on the simulation behavior. Determine the order yourself, in the 1:10 range, after seeing the plot of singular values.

```
load iddata1 z1
opt = n4sidOptions('Focus','simulation');
m = n4sid(z1, 1:10,'Ts',0,'form','canonical',opt)
bode(m)
```

## References

[1] Ljung, L. *System Identification — Theory For the User*, Appendix 4A, pp 132-134, 2nd ed, PTR Prentice Hall, Upper Saddle River, N.J., 1999.



[2] van Overschee, P., and B. De Moor, *Subspace Identification of Linear Systems: Theory, Implementation, Applications*, Kluwer Academic Publishers, 1996.

[3] Verhaegen, M., *Identification of the deterministic part of MIMO state space models*, Automatica, Vol. 30, pp. 61-74, 1994.

[4] Larimore, W.E., *Canonical variate analysis in identification, filtering and adaptive control*, Proc. 29th IEEE Conference on Decision and Control, pp.596-604, Honolulu, 1990.

### See Also

n4sidOptions | idss | ssest | tfest | procest | polyest |  
iddata | idfrd | idgrey | canon | pem

# n4sidOptions

---

**Purpose** Option set for n4sid

**Syntax**  
opt = n4sidOptions  
opt = n4sidOptions(Name,Value)

**Description** opt = n4sidOptions creates the default options set for n4sid.  
opt = n4sidOptions(Name,Value) creates an option set with the options specified by one or more Name,Value pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments, where Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

### InitialState

Specify handling of initial states during estimation.

InitialState requires one of the following values:

- 'zero' — The initial state is set to zero.
- 'estimate' — The initial state is treated as an independent estimation parameter.

**Default:** 'estimate'

### N4Weight

Weighting scheme used for singular-value decomposition by the N4SID algorithm.

'N4Weight' requires one of the following values:

- 'MOESP' — Uses the MOESP algorithm by Verhaegen [2].
- 'CVA' — Uses the Canonical Variable Algorithm by Larimore [1].

- 'auto' — The estimating function chooses between the MOESP and CVA algorithms.

**Default:** 'auto'

## **N4Horizon**

Forward- and backward-prediction horizons used by the N4SID algorithm.

'N4Horizon' requires one of the following values:

- A row vector with three elements —  $[r \ sy \ su]$ , where  $r$  is the maximum forward prediction horizon, using 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 [3] 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.  $k$  is the number of guesses of  $[r \ sy \ su]$  combinations. If you specify N4Horizon as a single column,  $r = sy = su$  is used.
- 'auto' — The software uses an Akaike Information Criterion (AIC) for the selection of  $sy$  and  $su$ .

**Default:** auto

## **Focus**

Defines how the errors  $e$  between the measured and the modeled outputs are weighed at specific frequencies during the minimization of the prediction error.

Higher weighting at specific frequencies emphasizes the requirement for a good fit at these frequencies.

**Focus** can take the following values:

- '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.

This method provides a stable model.

- 'prediction' — Automatically calculates the weighting function as a product of the input spectrum and the inverse of the noise model. The weighting function minimizes the one-step-ahead prediction, which typically favors fitting small time intervals (higher frequency range). From a statistical-variance point of view, this weighting function is optimal. However, this method neglects the approximation aspects (bias) of the fit. Thus, the method may not result in a stable model. Specify `FOCUS` as 'stability' when you want to ensure a stable model.
- 'stability' — Same as 'prediction', but with model stability enforced.
- Passbands — Row vector or matrix containing frequency values that define desired passbands. For example:

```
[w1,wh]  
[w11,w1h;w21,w2h;w31,w3h;...]
```

where `w1` 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.

- SISO filter — Enter any SISO linear filter in any of the following ways:
  - A single-input-single-output (SISO) linear system.
  - The {A,B,C,D} format, which specifies the state-space matrices of the filter.
  - The {numerator, denominator} format, which specifies the numerator and denominator of the filter transfer function

This format calculates the weighting function as a product of the filter and the input spectrum to estimate the transfer function. To obtain a good model fit for a specific frequency range, you must choose the filter with a passband in this range. The estimation result is the same if you first prefilter the data using `idfilt`.

- **Weighting vector** — 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.

**Default:** 'prediction'

### **EstCovar**

Controls whether parameter covariance data is generated or not.

If `EstCovar` is `true`, then use `getcov` to fetch the covariance matrix from the estimated model.

**Default:** `true`

### **Display**

Specifies whether estimation progress should be displayed.

`Display` requires one of the following strings:

- 'on' — Information on model structure and estimation results are displayed in a progress viewer window
- 'off' — No progress or results information is displayed

**Default:** 'off'

### **InputOffset**

Removes offset from time domain input data during estimation.

Specify as a column vector of length  $Nu$ , where  $Nu$  is the number of inputs.

Use `[]` to indicate no offset.

For multiexperiment data, specify `InputOffset` as a  $Nu$ -by- $Ne$  matrix.  $Nu$  is the number of inputs, and  $Ne$  is the number of experiments.

Each entry specified by `InputOffset` is subtracted from the corresponding input data.

**Default:** `[]`

## **OutputOffset**

Removes offset from time domain output data during estimation.

Specify as a column vector of length  $Ny$ , where  $Ny$  is the number of outputs.

Use `[]` to indicate no offset.

For multiexperiment data, specify `OutputOffset` as a  $Ny$ -by- $Ne$  matrix.  $Ny$  is the number of outputs, and  $Ne$  is the number of experiments. Each entry specified by `OutputOffset` is subtracted from the corresponding output data.

**Default:** `[]`

## **OutputWeight**

Specifies criterion used during minimization.

`OutputWeight` can have the following values:

- 'noise' — Minimize  $\det(E'^* E)$ , where  $E$  represents the prediction error. This choice is optimal in a statistical sense and leads to the maximum likelihood estimates in case no data is available about the variance of the noise. This option uses the inverse of the estimated noise variance as the weighting function.

- positive semidefinite symmetric matrix ( $W$ ) — Minimize the trace of the weighted prediction error matrix  $\text{trace}(E' * E * W)$ .  $E$  is the matrix of prediction errors, with one column for each output.  $W$  is the positive semidefinite symmetric matrix of size equal to the number of outputs. Use  $W$  to specify the relative importance of outputs in multiple-input, multiple-output models, or the reliability of corresponding data.

This option is relevant only for multi-input, multi-output models.

- [ ] — The software chooses between the 'noise' or using the identity matrix for  $W$ .

**Default:** [ ]

## Advanced

Advanced is a structure with the field MaxSize. MaxSize specifies the maximum number of elements in a segment when input-output data is split into segments.

MaxSize must be a positive integer.

**Default:** 250000

## Output Arguments

### opt

Option set containing the specified options for n4sid.

## Examples

### Create Default Options Set for State-Space Estimation Using Subspace Method

```
opt = n4sidOptions;
```

### Specify Options for State-Space Estimation Using Subspace Method

Create an options set for n4sid using the 'zero' option to initialize the state. Set the Display to 'on'.

```
opt = n4sidOptions('InitialState','zero','Display','on');
```

# n4sidOptions

---

Alternatively, use dot notation to set the values of `opt`.

```
opt = n4sidOptions;  
opt.InitialState = 'zero';  
opt.Display = 'on';
```

## References

- [1] Larimore, W.E. “Canonical variate analysis in identification, filtering and adaptive control.” *Proceedings of the 29th IEEE Conference on Decision and Control*, pp. 596–604, 1990.
- [2] Verhaegen, M. “Identification of the deterministic part of MIMO state space models.” *Automatica*, Vol. 30, 1994, pp. 61–74.
- [3] Ljung, L. *System Identification: Theory for the User*. Upper Saddle River, NJ: Prentice-Hall PTR, 1999.

## See Also

`n4sid` | `idpar` | `idfilt`



**Purpose** Query number of dimensions of dynamic system model or model array

**Syntax** `n = ndims(sys)`

**Description** `n = ndims(sys)` is the number of dimensions of a dynamic system model or a model array `sys`. A single model has two dimensions (one for outputs, and one for inputs). A model array has  $2 + p$  dimensions, where  $p \geq 2$  is the number of array dimensions. For example, a 2-by-3-by-4 array of models has  $2 + 3 = 5$  dimensions.

```
ndims(sys) = length(size(sys))
```

**Examples**

```
sys = rss(3,1,1,3);  
ndims(sys)  
ans =  
     4
```

`ndims` returns 4 for this 3-by-1 array of SISO models.

**See Also** `size`

# neuralnet

---

**Purpose** Class representing neural network nonlinearity estimator for nonlinear ARX models

**Syntax** `net_estimator = neuralnet(Network)`

**Description** `neuralnet` is the class that encapsulates the neural network nonlinearity estimator. A `neuralnet` object lets you use networks, created using Neural Network Toolbox™ software, in nonlinear ARX models.

The neural network nonlinearity estimator defines a nonlinear function  $y = F(x)$ , where  $F$  is a multilayer feed-forward (static) neural network, as defined in the Neural Network Toolbox software.  $y$  is a scalar and  $x$  is an  $m$ -dimensional row vector.

You create multi-layer feed-forward neural networks using Neural Network Toolbox commands such as `feedforwardnet`, `cascadeforwardnet` and `linearlayer`. When you create the network:

- Designate the input and output sizes to be unknown by leaving them at the default value of zero (recommended method). When estimating a nonlinear ARX model using the `nlarx` command, the software automatically determines the input-output sizes of the network.
- Initialize the sizes manually by setting input and output ranges to  $m$ -by-2 and 1-by-2 matrices, respectively, where  $m$  is the number of nonlinear ARX model regressors and the range values are minimum and maximum values of regressors and output data, respectively.

See “Examples” on page 2-594 for more information.

Use `evaluate(net_estimator, x)` to compute the value of the function defined by the `neuralnet` object `net_estimator` at input value  $x$ . When used for nonlinear ARX model estimation,  $x$  represents the model regressors for the output for which the `neuralnet` object is assigned as the nonlinearity estimator.

You cannot use `neuralnet` when Focus property of the `idnlarx` model is 'Simulation' because this nonlinearity estimator is considered to

be nondifferentiable for estimation. Minimization of simulation error requires differentiable nonlinear functions.

## Construction

`net_estimator = neuralnet(Network)` creates a neural network nonlinearity estimator based on the feed-forward (static) network object `Network` created using Neural Network Toolbox commands `feedforwardnet`, `cascadeforwardnet`, and `linearlayer`. `Network` must represent a static mapping between the inputs and output without I/O delays or feedback. The number of outputs of the network, if assigned, must be one. For a multiple-output nonlinear ARX models, create a separate `neuralnet` object for each output—that is, each estimator must represent a single-output network object.

## Properties

Network

Neural network object, typically created using the Neural Network Toolbox commands `feedforwardnet`, `cascadeforwardnet`, and `linearlayer`.

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.

## Examples

Create a neural network nonlinearity estimator using a feed-forward neural network with three hidden layers, transfer functions of types `logsig`, `radbas`, and `purelin` and unknown input and output sizes:

```
% Create a neural network.
net = feedforwardnet([4 6 1]);
net.layers{1}.transferFcn = 'logsig';
net.layers{2}.transferFcn = 'radbas';
net.layers{3}.transferFcn = 'purelin';
% View the network diagram.
view(net)
% Create a neuralnet estimator.
net_estimator = neuralnet(net);
```

---

Create a single-layer, cascade-forward network with unknown input and output sizes and use this network for nonlinear ARX model estimation:

- 1 Create a cascade-forward neural network with 20 neurons and unknown input-output sizes.

```
net = cascadeforwardnet(20);
```

- 2 Create a neural network nonlinearity estimator.

```
net_estimator = neuralnet(net);
```

- 3 Estimate nonlinear ARX model.

```
% Create estimation data.
load twotankdata
Data = iddata(y, u, 0.2);
% Estimate model.
Model = nlarx(Data, [2 2 1], net_estimator);
% Compare model response to measured output signal.
compare(Data, Model)
```

---

Initialize the input-output sizes of a two-layer feed-forward neural network based on estimation data and use this network for nonlinear ARX estimation:

**1** Create estimation data.

```
% Load estimation data.
load iddata7 z7
% Use only first 200 samples for estimation.
z7 = z7(1:200);
```

**2** Create a template Nonlinear ARX model with no nonlinearity.

```
model = idnlarx([4 4 4 1 1], []);
```

This model has six regressors and is simply used to define the regressors. The range of regressor values for input-output data in *z7* is then used to set the input ranges in the neural network object, as shown in the next steps.

**3** Obtain the model regressor values.

```
R = getreg(model, 'all', z7);
```

*R* is a matrix of regressor values for *z7*.

**4** Create a two-layer, feed-forward neural network and initialize the network input and output dimensions to 2 and 1, respectively.

```
% Use 5 neurons for first layer and 7 for second layer.
net = feedforwardnet([5 7]);
% Determine input range.
InputRange = [min(R); max(R)].';
% Initialize input dimensions of estimator.
net.inputs{1}.range = InputRange;
% Determine output range.
OutputRange = [min(z7.OutputData), max(z7.OutputData)];
% Initialize output dimensions of estimator.
```

```
net.outputs{net.outputConnect}.range = OutputRange;  
% Create neuralnet estimator.  
net_estimator = neuralnet(net);
```

**5** Specify the nonlinearity estimator in the model.

```
model.Nonlinearity = net_estimator;
```

**6** Estimate the parameters of the network to minimize the prediction error between data and model.

```
% Estimate model.  
model = nlarx(z7, model);  
% Compare model's predicted response to measured output signal.  
compare(z7(1:100), model,1)
```

## Algorithms

The `nlarx` command uses the `train` method of the network object, defined in the Neural Network Toolbox software, to compute the network parameter values.

## See Also

`nlarx` | `sigmoidnet` | `wavenet` | `treepartition` | `customnet` | `feedforwardnet` | `cascadeforwardnet` | `linearlayer`

## Tutorials

- “Identifying Nonlinear ARX Models”

---

<b>Purpose</b>	Shift data sequences
<b>Syntax</b>	<code>Datas = nkshift(Data,nk)</code>
<b>Description</b>	<p>Data contains input-output data in the <code>iddata</code> format.</p> <p><code>nk</code> is a row vector with the same length as the number of input channels in <code>Data</code>.</p> <p><code>Datas</code> is an <code>iddata</code> object where the input channels in <code>Data</code> have been shifted according to <code>nk</code>. A positive value of <code>nk(ku)</code> means that input channel number <code>ku</code> is delayed <code>nk(ku)</code> samples.</p> <p><code>nkshift</code> supports both frequency- and time-domain data. For frequency-domain data it multiplies with <math>e^{ink\omega T}</math> to obtain the same effect as shifting in the time domain. For continuous-time frequency-domain data (<math>T_s = 0</math>), <code>nk</code> should be interpreted as the shift in seconds.</p> <p><code>nkshift</code> lives in symbiosis with the <code>InputDelay</code> property of linear identified models:</p> <pre>m1 = ssest(dat,4,'InputDelay',nk)</pre> <p>is related to</p> <pre>m2 = ssest(nkshift(dat,nk),4);</pre> <p>such that <code>m1</code> and <code>m2</code> are the same models, but <code>m1</code> stores the delay information and uses this information when computing the frequency response, for example. When using <code>m2</code>, the delay value must be accounted for separately when computing time and frequency responses.</p>
<b>See Also</b>	<code>idpoly</code>   <code>absorbDelay</code>   <code>delayest</code>   <code>idss</code>

## Purpose

Estimate nonlinear ARX model

## Syntax

```
m = nlarx(data, [na nb nk])  
m = nlarx(data, [na nb nk], Nonlinearity)  
m = nlarx(data, [na nb nk], 'PropertyName', PropertyValue)  
m = nlarx(data, LinModel)  
m = nlarx(data, LinModel, Nonlinearity)  
m = nlarx(data, LinModel, Nonlinearity, 'PropertyName',  
    PropertyValue)
```

## Description

*m* = nlarx(*data*, [*na nb nk*]) creates and estimates a nonlinear ARX model using a default wavelet network as its nonlinearity estimator. *data* is an iddata object. *na*, *nb*, and *nk* are positive integers that specify the model orders and delays.

*m* = nlarx(*data*, [*na nb nk*], *Nonlinearity*) specifies a nonlinearity estimator *Nonlinearity*, as a nonlinearity estimator object or string representing the nonlinearity estimator type.

*m* = nlarx(*data*, [*na nb nk*], '*PropertyName*', *PropertyValue*) constructs and estimates the model using options specified as idnlrx property name and value pairs. Specify *PropertyName* inside single quotes.

*m* = nlarx(*data*, *LinModel*) creates and estimates a nonlinear ARX model using a linear model (in place of [*na nb nk*]), and a wavelet network as its nonlinearity estimator. *LinModel* is a discrete time input-output polynomial model of ARX structure (idpoly) for single-output systems, and idarx object, for multi-output systems. *LinModel* sets the model orders, input delay, input-output channel names and units, sample time, and time unit of *m*, and the polynomials initialize the linear function of the nonlinearity estimator.

*m* = nlarx(*data*, *LinModel*, *Nonlinearity*) specifies a nonlinearity estimator *Nonlinearity*.

```
m =  
nlarx(data, LinModel, Nonlinearity, 'PropertyName', PropertyValue),
```



constructs and estimates the model using options specified as idnlarx property name and value pairs.

## Input Arguments

### **data**

Time-domain iddata object.

### **na nb nk**

Positive integers that specify the model orders and delays.

For  $n_y$  output channels and  $n_u$  input channels,  $na$  is an  $n_y$ -by- $n_y$  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  $n_y$ -by- $n_u$  matrices, where each row defines the orders for the corresponding output.

### **Nonlinearity**

Nonlinearity estimator, specified as a nonlinearity estimator object or string representing the nonlinearity estimator type.

'wavenet' or wavenet object (default)	Wavelet network
'sigmoidnet' or sigmoidnet object	Sigmoid network
'treepartition' or treepartition object	Binary-tree
'linear' or [] or linear object	Linear function
neuralnet object	Neural network
customnet object	Custom network

Specifying a string creates a nonlinearity estimator object with default settings. Use object representation to configure the properties of a nonlinearity estimator.

For  $n_y$  output channels, you can specify nonlinear estimators individually for each output channel by setting *Nonlinearity* to an  $n_y$ -by-1 cell array or object array of nonlinearity estimators. To specify

the same nonlinearity for all outputs, specify *Nonlinearity* as a single nonlinearity estimator.

## LinModel

Discrete time input-output polynomial model of ARX structure, typically estimated using the `arx` command:

- `idpoly` object for single-output systems
- `idarx` object for multi-output systems

## Examples

Estimate nonlinear ARX model with default settings:

```
load twotankdata
Ts = 0.2; % Sampling interval is 0.2 min
z = iddata(y,u,Ts); % constructs iddata object
m = nlarx(z,[4 4 1]) % na=nb=4 and nk=1
```

---

Estimate nonlinear ARX model with a specific nonlinearity:

```
NL = wavenet('NumberOfUnits',5);
% Wavelet network has 5 units
m = nlarx(z,[4 4 1],NL)
```

---

Estimate nonlinear ARX model with a custom network nonlinearity:

```
% Define custom unit function and save it as gaussunit.m.
function [f, g, a] = GAUSSUNIT(x)
[f, g, a] = gaussunit(x)
f = exp(-x.*x);
if nargout>1
    g = - 2*x.*f;
    a = 0.2;
end
```

```
% Estimate nonlinear ARX model using the custom
% Gauss unit function.
H = @gaussunit;
CNetw = customnet(H);
m = nlarx(data,[na nb nk],CNetw)
```

---

Estimate nonlinear ARX model with specific algorithm settings:

```
m = nlarx(z,[4 4 1],'sigmoidnet','MaxIter',50,...
           'Focus','Simulation')
% Maximum number of estimation iterations is 50.
% Estimation focus 'simulation' optimizes model for
% simulation applications.
```

---

Estimate nonlinear ARX model from time series data:

```
t = 0:0.01:10;
y = 10*sin(2*pi*10*t)+rand(size(t));
z = iddata(y',[ ],0.01);
m = nlarx(z,2,'sigmoid')
compare(z,m,1) % compare 1-step-ahead
               % prediction pf response
```

---

Estimate nonlinear ARX model and avoid local minima:

```
% Estimate initial model.
load iddata1
m1=nlarx(z1,[4 2 1],'wave','nlr',[1:3])
% Perturb parameters slightly to avoid local minima:
m2=init(m1)
% Estimate model with perturbed initial parameter values:
m2=nlarx(z1,m2)
```

---

Estimate nonlinear ARX model with custom regressors:

```
% Load sample data z1 (iddata object).
load iddata1
% Estimate the model parameters:
m = nlrx(z1,[0 0 0],'linear','CustomReg',...
        {'y1(t-1)^2',...
         'y1(t-2)*u1(t-3)'}))
% na=nb=nk=0 means there are no standard regressors.
% 'linear' means that the nonlinear estimator has only
% the linear function.
```

---

Estimate nonlinear ARX model with custom regressor object:

```
% Load sample data z1 (iddata object):
load iddata1
% Define custom regressors as customreg objects:
C1 = customreg(@(x)x^2,{'y1'}, [1]); % y1(t-1)^2
C2 = customreg(@(x,y)x*y,{'y1', 'u1'},...
               [2 3]); % y1(t-2)*u1(t-3)
C = [C1, C2]; % object array of custom regressors
% Estimate model with custom regressors:
m = nlrx(z1,[0 0 0],'linear','CustomReg',C);
% List all model regressors:
getreg(m)
```

---

Estimate nonlinear ARX model and search for optimum regressors for input to the nonlinear function:

```
load iddata1
m = nlrx(z1,[4 4 1],'sigmoidnet',...
        'NonlinearRegressors','search');
m.NonlinearRegressors
```

---

```
% regressors indices in nonlinear function
```

---

Estimate nonlinear ARX model with selected regressors as inputs to the nonlinear function:

```
load iddata1
m = nlarx(z1,[4 4 1],'sigmoidnet',...
          'NonlinearReg','input');
% Only input regressors enter the nonlinear function.
% m is linear in past outputs.
```

---

Estimate nonlinear ARX model with no linear term in the nonlinearity estimator:

```
load iddata1
SNL = sigmoidnet('LinearTerm','off')
m = nlarx(z1,[2 2 1],SNL);
```

---

Estimate MIMO nonlinear ARX model that has the same nonlinearity estimator for all output channels:

```
m = nlarx(data,[[2 1;0 1] [2;1] [1;1]],...
            sigmoidnet('num',7))
% m uses a sigmoid network with 7 units
% for all output channels.
```

---

Estimate MIMO nonlinear ARX model with different nonlinearity estimator for each output channel:

```
m = nlarx(data,[[2 1;0 1] [2;1] [1;1]],...
            ['wavenet'; sigmoidnet('num',7)])
% first output channel uses a wavelet network
% second output channel uses a sigmoid network with 7 units
```

Estimate a nonlinear ARX model using an ARX model:

```
% Estimate linear ARX model.
load throttledata.mat
Tr = getTrend(ThrottleData);
Tr.OutputOffset = 15;
DetrendedData = detrend(ThrottleData, Tr);
LinearModel = arx(DetrendedData, [2 1 1], 'Focus', 'Simulation');

% Estimate nonlinear ARX model using linear model to model
% output saturation in data.
NonlinearModel = nlarx(ThrottleData, LinearModel, 'sigmoidnet',...
    'Focus', 'Simulation')
```

## See Also

[addreg](#) | [customreg](#) | [getreg](#) | [idnlarx](#) | [init](#) | [polyreg](#)

## Tutorials

- “Example – Using nlarx to Estimate Nonlinear ARX Models”
- “Estimate Nonlinear ARX Models Using Linear ARX Models”

## How To

- “Identifying Nonlinear ARX Models”
- “Using Linear Model for Nonlinear ARX Estimation”

**Purpose**

Estimate Hammerstein-Wiener model

**Syntax**

```

m = nlhw(data,[nb nf nk])
m = nlhw(data,[nb nf nk],InputNL,OutputNL)
m = nlhw(data,[nb nf nk],InputNL,OutputNL,'PropertyName',
    PropertyValue)
m = nlhw(data,LinModel)
m = nlhw(data,LinModel,InputNL,OutputNL)
m = nlhw(data,LinModel,InputNL,OutputNL,'PropertyName',
    PropertyValue)

```

**Description**

`m = nlhw(data,[nb nf nk])` creates and estimates a Hammerstein-Wiener model using piecewise linear functions as its input and output nonlinearity estimators. `data` is a time-domain `iddata` object. `nb`, `nf`, and `nk` are positive integers that specify the model orders and delay. `nb` is the number of zeros plus 1, `nf` is the number of poles, and `nk` is the input delay.

`m = nlhw(data,[nb nf nk],InputNL,OutputNL)` specifies input nonlinearity `InputNL` and output nonlinearity `OutputNL`, as a nonlinearity estimator object or string representing the nonlinearity estimator type.

`m = nlhw(data,[nb nf nk],InputNL,OutputNL,'PropertyName',PropertyValue)` creates and estimates the model using options specified as `idnlhw` property name and value pairs. Specify `PropertyName` inside single quotes.

`m = nlhw(data,LinModel)` creates and estimates a Hammerstein-Wiener model using a linear model (in place of `[nb nf nk]`), and default piecewise linear functions for the input and output nonlinearity estimators. `LinModel` is a discrete-time input-output polynomial model of Output-Error (OE) structure (`idpoly`) or state-space model with no disturbance component (`idss` with  $K = 0$ ) for single-output systems, and `idss` model with  $K = 0$  for multi-output systems. `LinModel` sets the model orders, input delay,  $B$  and  $F$  polynomial values, input-output names and units, sampling time and time units of `m`.

$m = \text{nlhw}(\text{data}, \text{LinModel}, \text{InputNL}, \text{OutputNL})$  specifies input nonlinearity *InputNL* and output nonlinearity *OutputNL*.

$m = \text{nlhw}(\text{data}, \text{LinModel}, \text{InputNL}, \text{OutputNL}, \text{'PropertyName'}, \text{PropertyValue})$  creates and estimates the model using options specified as *idnlhw* property name and value pairs.

## Input Arguments

### **data**

Time-domain *iddata* object.

### **nb, nf nk**

Order of the linear transfer function, where *nb* is the number of zeros plus 1, *nf* is the number of poles, and *nk* is the input delay.

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, OutputNL**

Input and output nonlinearity estimators, respectively, specified as a nonlinearity estimator object or string representing the nonlinearity estimator type.

'pwnlinear' or *pwnlinear* object  
(default)

Piecewise linear function

'sigmoidnet' or *sigmoidnet* object

Sigmoid network

'wavenet' or *wavenet* object

Wavelet network

'saturation' or *saturation* object

Saturation

'deadzone' or *deadzone* object

Dead zone

'poly1d' or *poly1d* object

One-dimensional polynomial



'unitgain' or unitgain object

Unit gain

customnet object

Custom network

Specifying a string creates a nonlinearity estimator object with default settings. Use object representation to configure the properties of a nonlinearity estimator.

For  $n_y$  output channels, you can specify nonlinear estimators individually for each output channel by setting *InputNL* or *OutputNL* to an  $n_y$ -by-1 cell array or object array of nonlinearity estimators. To specify the same nonlinearity for all outputs, specify a single input and output nonlinearity estimator.

### LinModel

Discrete time linear model, typically estimated using the `oe` or `n4sid` command:

- Input-output polynomial model of Output-Error (OE) structure (`idpoly`) or state-space model with no disturbance component (`idss` with  $K = 0$ )

## Examples

Estimate a Hammerstein-Wiener model:

```
load iddata3
m1=nlhw(z3,[4 2 1],'sigmoidnet','deadzone')
```

---

Estimate a Hammerstein model with saturation:

```
load iddata1
% Create a saturation object with lower limit of 0
% and upper limit of 5:
InputNL = saturation('LinearInterval', [0 5]);
% Estimate model with no output nonlinearity.
m = nlhw(z1,[2 3 0],InputNL,[]);
```

---

Estimate a Wiener model with a nonlinearity containing 5 sigmoid units:

```
load iddata1
m2 = nlhw(z1,[2 3 0],[],sigmoidnet('num', 5))
```

---

Estimate a Hammerstein-Wiener model with a custom network nonlinearity:

```
% Define custom unit function and save it as gaussunit.m.
function [f, g, a] = GAUSSUNIT(x)
[f, g, a] = gaussunit(x)
f = exp(-x.*x);
if nargout>1
    g = - 2*x.*f;
    a = 0.2;
end

% Estimate Hammerstein-Wiener model using the custom
% Gauss unit function.
H = @gaussunit;
CNetw = cutomnet(H);
m = nlhw(data,[na nb nk],CNetw)
```

---

Estimate a MISO Hammerstein model with a different nonlinearity for each input:

```
m = nlhw(data,[nb,nf,nk],...
          [sigmoidnet;pwlinear],...
          [])
```

---

Refine a Hammerstein-Wiener model using successive calls of nlhw:

```
load iddata3
```

```
m3 = nlhw(z3,[4 2 1], 'sigmoidnet', 'deadzone')
m3 = nlhw(z3,m3)
LinearBlock = m3.LinearModel % retrieves the linear block
```

---

Estimate a Hammerstein-Wiener model and avoid local minima:

```
load iddata3
M1 = nlhw(z3, [2 2 1], 'sigm', 'wave'); % original model
M1p = init(M1); % randomly perturbs parameters about nominal values
M2 = pem(z3, M1p); % estimates parameters of perturbed model
```

---

Estimate default Hammerstein-Wiener model using an input-output polynomial model of Output-Error (OE) structure:

```
% Estimate linear OE model.
load throttledata.mat
Tr = getTrend(ThrottleData);
Tr.OutputOffset = 15;
DetrendedData = detrend(ThrottleData, Tr);
opt = oeOptions('Focus', 'simulation');
LinearModel = oe(DetrendedData,[1 2 1],opt);

% Estimate Hammerstein-Wiener model using OE model as
% its linear component and saturation as its output nonlinearity.
NonlinearModel = nlhw(ThrottleData, LinearModel, [], 'saturation')
```

## See Also

customnet | deadzone | findop(idnlhw) | linapp |  
 linearize(idnlhw) | idnlhw | pem | poly1d | pwlinear | saturation  
 | sigmoidnet | unitgain | wavenet

## Tutorials

- “Example – Using nlhw to Estimate Hammerstein-Wiener Models”
- “Estimate Hammerstein-Wiener Models Using Linear OE Models”

## How To

- “Identifying Hammerstein-Wiener Models”

- “Using Linear Model for Hammerstein-Wiener Estimation”

---

<b>Purpose</b>	Noise component of model
<b>Syntax</b>	<pre>noise_model = noise2meas(sys) noise_model = noise2meas(sys,noise)</pre>
<b>Description</b>	<p><code>noise_model = noise2meas(sys)</code> returns the noise component, <code>noise_model</code>, of a linear identified model, <code>sys</code>. Use <code>noise2meas</code> to convert a time-series model (no inputs), to an input/output model that may be used for linear analysis, including viewing pole/zero maps, and plotting the step response.</p> <p><code>noise_model = noise2meas(sys,noise)</code> specifies the noise variance normalization method.</p>
<b>Input Arguments</b>	<p><b>sys</b> Identified linear model.</p> <p><b>noise</b> Noise variance normalization method.</p> <p><code>noise</code> is a string that takes one of the following values:</p> <ul style="list-style-type: none"><li>• 'innovations' — Noise sources are not normalized and remain as the innovations process.</li><li>• 'normalize' — Noise sources are normalized to be independent and of unit variance.</li></ul> <p style="text-align: center;"><b>Default:</b> 'innovations'</p>
<b>Output Arguments</b>	<p><b>noise_model</b> Noise component of <code>sys</code>.</p> <p><code>sys</code> represents the system</p> $y(t) = Gu(t) + He(t)$

Here  $G$  is the transfer function between the measured input,  $u(t)$ , and the output,  $y(t)$ .  $H$  is the noise model and describes the effect of the disturbance,  $e(t)$ , on the model's response.

An equivalent state-space representation of `sys` is

$$\begin{aligned}\dot{x}(t) &= Ax(t) + Bu(t) + Ke(t) \\ y(t) &= Cx(t) + Du(t) + e(t) \\ e(t) &= Lv(t)\end{aligned}$$

Here,  $v(t)$  is white noise with independent channels and unit variances. The white noise signal  $e(t)$  represents the model's innovations and has variance  $LL^T$ . The noise variance data is stored using the `NoiseVariance` property of `sys`.

- If `noise` is 'innovations', `noise2meas` returns  $H$  and `noise_model` represents the system

$$y(t) = He(t)$$

An equivalent state-space representation of `noise_model` is

$$\begin{aligned}\dot{x}(t) &= Ax(t) + Ke(t) \\ y(t) &= Cx(t) + e(t)\end{aligned}$$

`noise2meas` returns the noise channels of `sys` as the input channels of `noise_model`. The input channels are named using the format 'e@yk', where `yk` corresponds to the `OutputName` property of an output. The measured input channels of `sys` are discarded and the noise variance is set to zero.

- If `noise` is 'normalize', `noise2meas` first normalizes

$$e(t) = Lv(t)$$

`noise_model` represents the system

$$y(t) = HLv(t)$$

or, equivalently, in state-space representation

$$\begin{aligned}\dot{x}(t) &= Ax(t) + KLv(t) \\ y(t) &= Cx(t) + Lv(t)\end{aligned}$$

The input channels are named using the format 'v@yk', where yk corresponds to the `OutputName` property of an output.

The model type of `noise_model` depends on the model type of `sys`.

- `noise_model` is an `idtf` model if `sys` is an `idproc` model.
- `noise_model` is an `idss` model if `sys` is an `idgrey` model.
- `noise_model` is the same type of model as `sys` for all other model types.

To obtain the model coefficients of `noise_model` in state-space form, use `ssdata`. Similarly, to obtain the model coefficients in transfer-function form, use `tfddata`.

## Examples

### Convert Noise Component of Linear Identified Model into Input/Output Model

Convert a time series model (no inputs) to an input/output model that may be used by linear analysis tools.

Identify a time series model

```
load iddata9 z9;
sys = ar(z9,4,'ls');
```

`sys` is an `idpoly` model with no inputs.

Convert `sys` to a measured model.

```
noise_model = noise2meas(sys);
```

`noise_model` is an `idpoly` model with one input.

You can use `noise_model` for linear analysis functions such as `step`, `iopzmap`, etc.

## Normalizing Noise Variance

Convert an identified linear model's to an input/output model and normalize its noise variance.

Identify a linear model using data.

```
load twotankdata;
data = iddata(y,u,0.2);
sys = ssest(z,4);
```

`sys` is an `idss` model, with a noise variance of  $6.6211e-06$ . The value of  $L$  is `sqrt(sys.NoiseVariance)`, which is  $0.0026$ . The disturbance matrix is

```
sys.K
```

```
    0.2719
    1.6570
   -0.6318
   -0.2877
```

Obtain a model that absorbs the noise variance of `sys`.

```
noise_model_normalize = noise2meas(sys,'normalize');
```

`noise_model_normalize` is an `idpoly` model. The  $B$  matrix for `noise_model_normalize` is

```
noise_model_normalize.B
```

```
    0.0007
    0.0043
   -0.0016
   -0.0007
```

As expected, `noise_model_normalize.B` is equal to  $L * sys.K$ .



Compare the Bode response with a model that ignores the noise variance of `sys`.

```
noise_model_innovation = noise2meas(sys,'innovations');  
bodemag(noise_model_normalize,noise_model_innovation);  
legend('Normalized noise variance','Ignored noise variance');
```

The difference between the bode magnitudes of the `noise_model_innovation` and `noise_model_normalized` is approximately 51dB. As expected, the magnitude difference is approximately equal to  $20 \cdot \log_{10}(L)$ .

**See Also**

[noisecnv](#) | [tfdata](#) | [zpkdata](#) | [idssdata](#) | [spectrum](#)

**Purpose** Transform `idmodel` object with noise channels to model with measured channels only

**Syntax**

```
mod1 = noisecnv(mod)
mod2 = noisecnv(mod, 'normalize')
```

**Description** `mod` is any linear identified model, `idproc`, `idtf`, `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  $\Lambda$ :

$$y = Gu + He$$
$$\text{cov}(e) = \Lambda = LL'$$

where  $L$  is a lower triangular matrix. Note that `mod.NoiseVariance` =  $\Lambda$ . The model can also be described with unit variance, using a normalized noise source  $v$ :

$$y = Gu + HLv$$
$$\text{cov}(v) = I$$

- `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(:,[])` 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(:,[]))` or `noise2meas(m)` describe just the transfer function  $H$ . To obtain a description of the normalized transfer function  $HL$ , use `noisecnv(m(:,[]),'normalize')` or `noise2meas('normalize')`.

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.

## Examples

Identify a model with a measured component ( $G$ ) and a non-trivial noise component ( $H$ ). Compare the amplitude of the measured component's frequency response to the noise component's spectrum amplitude. You must convert the noise component into a measured one by using `noisecnv` if you want to compare its behavior against a truly measured component.

```
load iddata2 z2
sys1 = armax(z2,[2 2 2 1]); % model with noise component
sys2 = tfest(z2,3); % model with a trivial noise component

sys1 = noisecnv(sys1);
sys2 = noisecnv(sys2);
bodemag(sys1,sys2)
```

## See Also

`noise2meas` | `tfddata` | `zpkdata` | `idssdata`

# norm

---

## Purpose

Norm of linear model

## Syntax

```
n = norm(sys)
n = norm(sys,2)
n = norm(sys,inf)
[n,fpeak] = norm(sys,inf)
[...] = norm(sys,inf,tol)
```

## Description

`n = norm(sys)` or `n = norm(sys,2)` return the  $H_2$  norm of the linear dynamic system model `sys`.

`n = norm(sys,inf)` returns the  $H_\infty$  norm of `sys`.

`[n,fpeak] = norm(sys,inf)` also returns the frequency `fpeak` at which the gain reaches its peak value.

`[...] = norm(sys,inf,tol)` sets the relative accuracy of the  $H_\infty$  norm to `tol`.

## Input Arguments

### **sys**

Continuous- or discrete-time linear dynamic system model. `sys` can also be an array of linear models.

### **tol**

Positive real value setting the relative accuracy of the  $H_\infty$  norm.

**Default:** 0.01

## Output Arguments

### **n**

$H_2$  norm or  $H_\infty$  norm of the linear model `sys`.

If `sys` is an array of linear models, `n` is an array of the same size as `sys`. In that case each entry of `n` is the norm of each entry of `sys`.

### **fpeak**

Frequency at which the peak gain of `sys` occurs.

**Definitions****H2 norm**

The  $H_2$  norm of a stable continuous-time system with transfer function  $H(s)$ , is given by:

$$\|H\|_2 = \sqrt{\frac{1}{2\pi} \int_{-\infty}^{\infty} \text{Trace} [H(j\omega)^H H(j\omega)] d\omega}.$$

For a discrete-time system with transfer function  $H(z)$ , the  $H_2$  norm is given by:

$$\|H\|_2 = \sqrt{\frac{1}{2\pi} \int_{-\pi}^{\pi} \text{Trace} [H(e^{j\omega})^H H(e^{j\omega})] d\omega}.$$

The  $H_2$  norm is equal to the root-mean-square of the impulse response of the system. The  $H_2$  norm measures the steady-state covariance (or power) of the output response  $y = Hw$  to unit white noise inputs  $w$ :

$$\|H\|_2^2 = \lim_{t \rightarrow \infty} E \{y(t)^T y(t)\}, \quad E(w(t)w(\tau)^T) = \delta(t - \tau)I.$$

The  $H_2$  norm is infinite in the following cases:

- sys is unstable.
- sys is continuous and has a nonzero feedthrough (that is, nonzero gain at the frequency  $\omega = \infty$ ).

`norm(sys)` produces the same result as

`sqrt(trace(covar(sys,1)))`

**H-infinity norm**

The  $H_\infty$  norm (also called the  $L_\infty$  norm) of a SISO linear system is the peak gain of the frequency response. For a MIMO system, the  $H_\infty$  norm is the peak gain across all input/output channels. Thus, for a continuous-time system  $H(s)$ , the  $H_\infty$  norm is given by:

$$\|H(s)\|_{\infty} = \max_{\omega} |H(j\omega)| \quad (\text{SISO})$$

$$\|H(s)\|_{\infty} = \max_{\omega} \sigma_{\max}(H(j\omega)) \quad (\text{MIMO})$$

where  $\sigma_{\max}(\cdot)$  denotes the largest singular value of a matrix.

For a discrete-time system  $H(z)$ :

$$\|H(z)\|_{\infty} = \max_{\theta \in [0, \pi]} |H(e^{j\theta})| \quad (\text{SISO})$$

$$\|H(z)\|_{\infty} = \max_{\theta \in [0, \pi]} \sigma_{\max}(H(e^{j\theta})) \quad (\text{MIMO})$$

The  $H_{\infty}$  norm is infinite if sys has poles on the imaginary axis (in continuous time), or on the unit circle (in discrete time).

## Examples

This example uses norm to compute the  $H_2$  and  $H_{\infty}$  norms of a discrete-time linear system.

Consider the discrete-time transfer function

$$H(z) = \frac{z^3 - 2.841z^2 + 2.875z - 1.004}{z^3 - 2.417z^2 + 2.003z - 0.5488}$$

with sample time 0.1 second.

To compute the  $H_2$  norm of this transfer function, enter:

```
H = tf([1 -2.841 2.875 -1.004],[1 -2.417 2.003 -0.5488],0.1)
norm(H)
```

These commands return the result:

```
ans =
    1.2438
```

To compute the  $H_{\infty}$  infinity norm, enter:

```
[ninf,fpeak] = norm(H,inf)
```

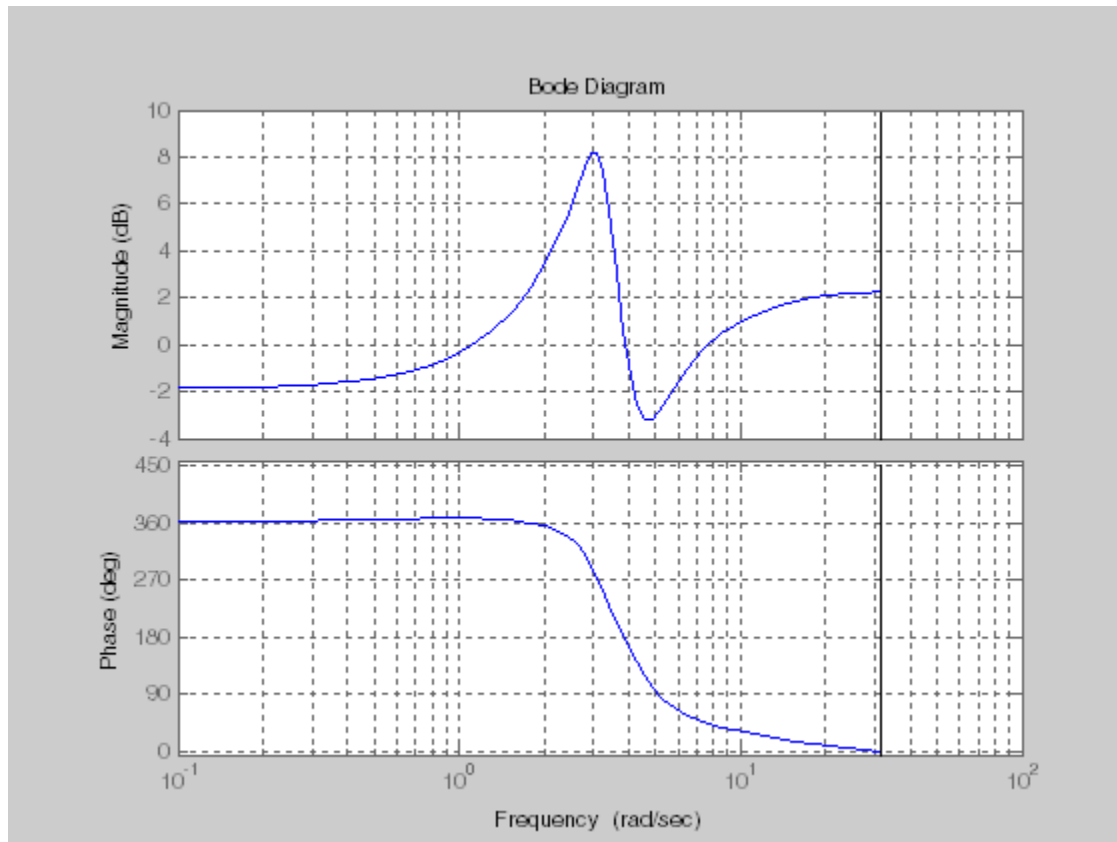
This command returns the result:

```
ninf =  
    2.5488
```

```
fpeak =  
    3.0844
```

You can use a Bode plot of  $H(z)$  to confirm these values.

```
bode(H)  
grid on;
```



The gain indeed peaks at approximately 3 rad/sec. To find the peak gain in dB, enter:

```
20*log10(ninf)
```

This command produces the following result:

```
ans =  
    8.1268
```



**Algorithms**

`norm` first converts `sys` to a state space model.

`norm` uses the same algorithm as `covar` for the  $H_2$  norm. For the  $H_\infty$  norm, `norm` uses the algorithm of [1]. `norm` computes the  $H_\infty$  norm (peak gain) using the SLICOT library. For more information about the SLICOT library, see <http://slicot.org>.

**References**

[1] Bruisma, N.A. and M. Steinbuch, "A Fast Algorithm to Compute the  $H_\infty$ -Norm of a Transfer Function Matrix," *System Control Letters*, 14 (1990), pp. 287-293.

**See Also**

`freqresp` | `sigma`

# nparams

---

**Purpose** Number of model parameters

**Syntax**  
`np = nparams(sys)`  
`np = nparams(sys, 'free')`

**Description** `np = nparams(sys)` returns the number of parameters in the identified model `sys`.

`np = nparams(sys, 'free')` returns the number free estimation parameters in the identified model `sys`.

---

**Note** Not all model coefficients are parameters, such as the leading entry of the denominator polynomials in `idpoly` and `idtf` models.

---

**Input Arguments** **sys**  
Identified linear model.

**Output Arguments** **np**  
Number of parameters of `sys`.  
For the syntax `np = nparams(sys, 'free')`, `np` is the number of free estimation parameters of `sys`.  
`idgrey` models can contain non-scalar parameters. `nparams` accounts for each individual entry of the non-scalar parameters in the total parameter count.

**Examples** Obtain the number of parameters of a transfer function model.

```
sys = idtf(1,[1 2]);  
np = nparams(sys);
```

---

Obtain the number of free estimation parameters of a transfer function model.

```
sys0 = idtf([1 0],[1 2 0]);  
sys0.Structure.den.Free(3) = false;  
np = nparams(sys,'free');
```

## See Also

[size](#) | [idpoly](#) | [idss](#) | [idtf](#) | [idproc](#) | [idgrey](#) | [idfnd](#)

# nuderst

---

**Purpose** Set step size for numerical differentiation

**Syntax** `nds = nuderst(pars)`

**Description** Many estimation functions use numerical differentiation with respect to the model parameters to compute their values.

The step size used in these numerical derivatives is determined by the `nuderst` command. 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**

Nyquist plot of frequency response

**Syntax**

```
nyquist(sys)
nyquist(sys,w)
nyquist(sys1,sys2,...,sysN)
nyquist(sys1,sys2,...,sysN,w)
nyquist(sys1,'PlotStyle1',...,sysN,'PlotStyleN')
[re,im,w] = nyquist(sys)
[re,im] = nyquist(sys,w)
[re,im,w,sdre,sdim] = nyquist(sys)
```

**Description**

`nyquist` creates a Nyquist plot of the frequency response of a dynamic system model. When invoked without left-hand arguments, `nyquist` produces a Nyquist plot on the screen. Nyquist plots are used to analyze system properties including gain margin, phase margin, and stability.

`nyquist(sys)` creates a Nyquist plot of a dynamic system `sys`. This model can be continuous or discrete, and SISO or MIMO. In the MIMO case, `nyquist` produces an array of Nyquist plots, each plot showing the response of one particular I/O channel. The frequency points are chosen automatically based on the system poles and zeros.

`nyquist(sys,w)` explicitly specifies the frequency range or frequency points to be used for the plot. To focus on a particular frequency interval, 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. Frequencies must be in `rad/TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`.

`nyquist(sys1,sys2,...,sysN)` or `nyquist(sys1,sys2,...,sysN,w)` superimposes the Nyquist plots of several LTI models on a single figure. All systems must have the same number of inputs and outputs, but may otherwise be a mix of continuous- and discrete-time systems. You can also specify a distinctive color, linestyle, and/or marker for each system plot with the syntax

```
nyquist(sys1, 'PlotStyle1', ..., sysN, 'PlotStyleN')
```

See `bode` for an example.

When invoked with left-hand arguments

```
[re,im,w] = nyquist(sys)
[re,im] = nyquist(sys,w)
```

return the real and imaginary parts of the frequency response at the frequencies `w` (in rad/TimeUnit). `re` and `im` are 3-D arrays (see "Arguments" below for details).

`[re,im,w,sdre,sdim] = nyquist(sys)` also returns the standard deviations of `re` and `im` for the identified system `sys`.

## Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see "Ways to Customize Plots".

## Arguments

The output arguments `re` and `im` are 3-D arrays with dimensions

$$(\text{number of outputs}) \times (\text{number of inputs}) \times (\text{length of } w)$$

For SISO systems, the scalars `re(1,1,k)` and `im(1,1,k)` are the real and imaginary parts of the response at the frequency  $\omega_k = w(k)$ .

$$\text{re}(1,1,k) = \text{Re}(h(j\omega_k))$$

$$\text{im}(1,1,k) = \text{Im}(h(j\omega_k))$$

For MIMO systems with transfer function  $H(s)$ , `re(:, :, k)` and `im(:, :, k)` give the real and imaginary parts of  $H(j\omega_k)$  (both arrays with as many rows as outputs and as many columns as inputs). Thus,

$$\text{re}(i, j, k) = \text{Re}(h_{ij}(j\omega_k))$$

$$\text{im}(i, j, k) = \text{Im}(h_{ij}(j\omega_k))$$

where  $h_{ij}$  is the transfer function from input  $j$  to output  $i$ .

## Examples

### Example 1

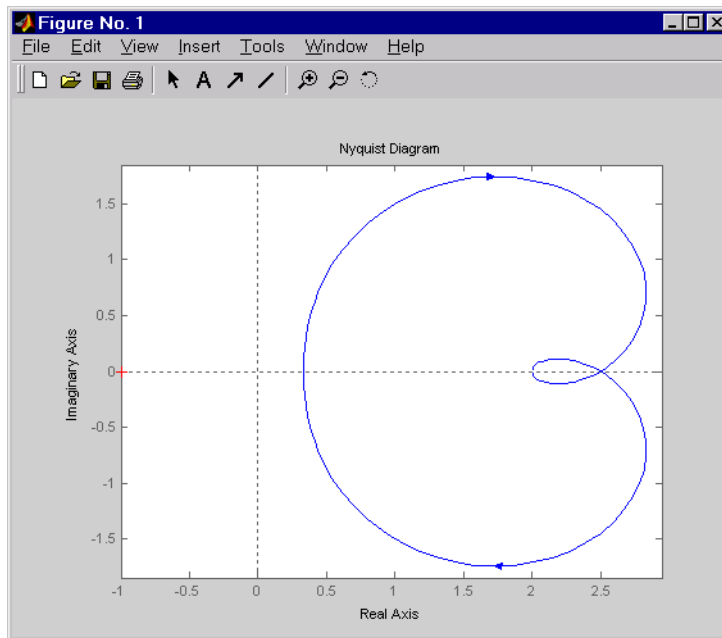
#### Nyquist Plot of Dynamic System

Plot the Nyquist response of the system

$$H(s) = \frac{2s^2 + 5s + 1}{s^2 + 2s + 3}$$

```
H = tf([2 5 1],[1 2 3])  
nyquist(H)
```

# nyquist



The nyquist function has support for M-circles, which are the contours of the constant closed-loop magnitude. M-circles are defined as the locus of complex numbers where

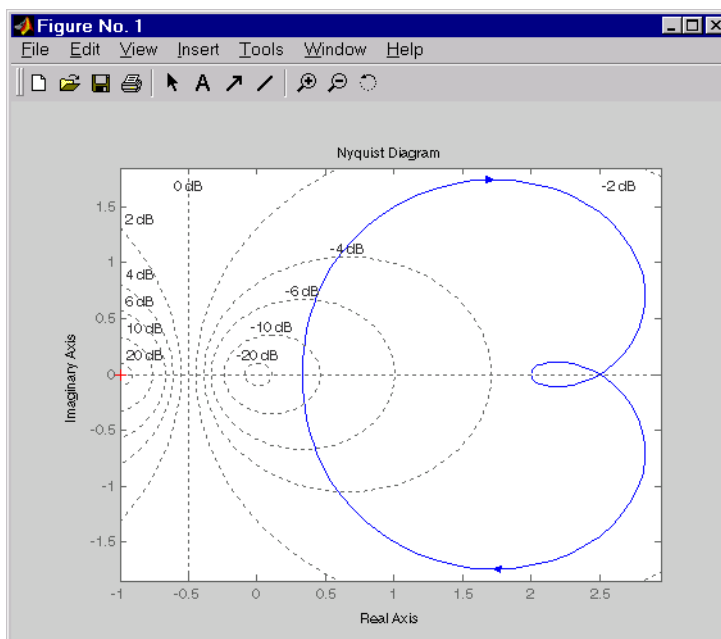
$$T(j\omega) = \left| \frac{G(j\omega)}{1 + G(j\omega)} \right|$$

is a constant value. In this equation,  $\omega$  is the frequency in radians/TimeUnit, where TimeUnit is the system time units, and  $G$  is the collection of complex numbers that satisfy the constant magnitude requirement.

To activate the grid, select **Grid** from the right-click menu or type  
grid



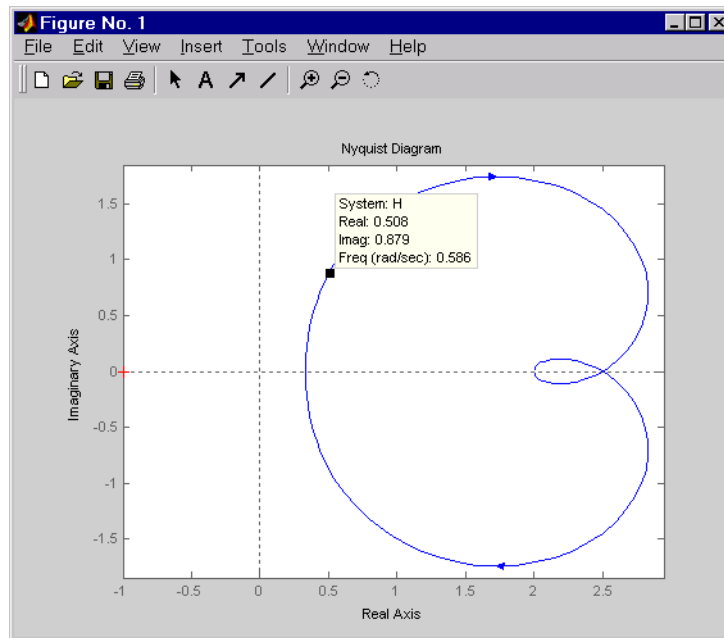
at the MATLAB prompt. This figure shows the M circles for transfer function  $H$ .



You have two zoom options available from the right-click menu that apply specifically to Nyquist plots:

- **Tight** — Clips unbounded branches of the Nyquist plot, but still includes the critical point  $(-1, 0)$
- **On  $(-1,0)$**  — Zooms around the critical point  $(-1,0)$

Also, click anywhere on the curve to activate data markers that display the real and imaginary values at a given frequency. This figure shows the nyquist plot with a data marker.



## Example 2

Compute the standard deviation of the real and imaginary parts of frequency response of an identified model. Use this data to create a 3 $\sigma$  plot of the response uncertainty.

Identify a transfer function model based on data. Obtain the standard deviation data for the real and imaginary parts of the frequency response.

```
load iddata2 z2;  
sys_p = tfest(z2,2);  
w = linspace(-10*pi,10*pi,512);  
[re, im, ~, sdre, sdim] = nyquist(sys_p,w);
```

sys\_p is an identified transfer function model. sdre and sdim contain 1-std standard deviation uncertainty values in re and im respectively.

Create a Nyquist plot showing the response and its  $3\sigma$  uncertainty:

```
re = squeeze(re);  
im = squeeze(im);  
sdre = squeeze(sdre);  
sdim = squeeze(sdim);  
plot(re,im,'b', re+3*sdre, im+3*sdim, 'k:', re-3*sdre, im-3*sdim, 'k:');
```

## Algorithms

See bode.

## See Also

bode | evalfr | freqresp | ltiview | nichols | sigma

# nyquistoptions

---

**Purpose** List of Nyquist plot options

**Syntax** P = nyquistoptions  
P = nyquistoptions('cstprefs')

**Description** P = nyquistoptions returns the default options for Nyquist plots. You can use these options to customize the Nyquist plot appearance using the command line.

P = nyquistoptions('cstprefs') initializes the plot options with the options you selected in the Control System Toolbox Preferences Editor. For more information about the editor, see “Toolbox Preferences Editor” in the User’s Guide documentation.

The following table summarizes the Nyquist plot options.

Option	Description
Title, XLabel, YLabel	Label text and style
TickLabel	Tick label style
Grid	Show or hide the grid Specified as one of the following strings: 'off'   'on' <b>Default:</b> 'off'
XlimMode, YlimMode	Limit modes
Xlim, Ylim	Axes limits
IOWGrouping	Grouping of input-output pairs Specified as one of the following strings: 'none'   'inputs'   'output'   'all' <b>Default:</b> 'none'
InputLabels, OutputLabels	Input and output label styles
InputVisible, OutputVisible	Visibility of input and output channels

Option	Description
FreqUnits	<p data-bbox="516 317 1221 348">Frequency units, specified as one of the following strings:</p> <ul data-bbox="516 383 836 1420" style="list-style-type: none"><li data-bbox="516 383 599 414">• 'Hz'</li><li data-bbox="516 432 718 463">• 'rad/second'</li><li data-bbox="516 480 614 512">• 'rpm'</li><li data-bbox="516 529 614 560">• 'kHz'</li><li data-bbox="516 578 614 609">• 'MHz'</li><li data-bbox="516 626 614 657">• 'GHz'</li><li data-bbox="516 675 777 706">• 'rad/nanosecond'</li><li data-bbox="516 723 792 755">• 'rad/microsecond'</li><li data-bbox="516 772 792 803">• 'rad/millisecond'</li><li data-bbox="516 821 718 852">• 'rad/minute'</li><li data-bbox="516 869 688 900">• 'rad/hour'</li><li data-bbox="516 918 673 949">• 'rad/day'</li><li data-bbox="516 966 688 998">• 'rad/week'</li><li data-bbox="516 1015 703 1046">• 'rad/month'</li><li data-bbox="516 1064 688 1095">• 'rad/year'</li><li data-bbox="516 1112 822 1144">• 'cycles/nanosecond'</li><li data-bbox="516 1161 836 1192">• 'cycles/microsecond'</li><li data-bbox="516 1209 836 1241">• 'cycles/millisecond'</li><li data-bbox="516 1258 733 1289">• 'cycles/hour'</li><li data-bbox="516 1307 718 1338">• 'cycles/day'</li><li data-bbox="516 1355 733 1387">• 'cycles/week'</li><li data-bbox="516 1404 747 1435">• 'cycles/month'</li></ul>

# nyquistoptions

Option	Description
	<ul style="list-style-type: none"><li>'cycles/year'</li></ul> <p><b>Default:</b> 'rad/s'</p> <p>You can also specify 'auto' which uses frequency units rad/TimeUnit relative to system time units specified in the TimeUnit property. For multiple systems with different time units, the units of the first system are used.</p>
MagUnits	Magnitude units Specified as one of the following strings: 'dB'   'abs' <b>Default:</b> 'dB'
PhaseUnits	Phase units Specified as one of the following strings: 'deg'   'rad' <b>Default:</b> 'deg'
ShowFullContour	Show response for negative frequencies Specified as one of the following strings: 'on'   'off' <b>Default:</b> 'on'
ConfidenceRegionNumberSD	Number of standard deviations to use to plotting the response confidence region (identified models only). <b>Default:</b> 1.
ConfidenceRegionDisplaySpacing	Frequency spacing of confidence ellipses. For identified models only. <b>Default:</b> 5, which means the confidence ellipses are shown at every 5th frequency sample.

## Examples

This example shows how to create a Nyquist plot displaying the full contour (the response for both positive and negative frequencies).

```
P = nyquistoptions;  
P.ShowFullContour = 'on';  
h = nyquistplot(tf(1,[1,.2,1]),P);
```

**See Also**

nyquist | nyquistplot | getoptions | setoptions | setoptions  
| showConfidence

# nyquistplot

---

## Purpose

Nyquist plot with additional plot customization options

## Syntax

```
h = nyquistplot(sys)
nyquistplot(sys, {wmin, wmax})
nyquistplot(sys, w)
nyquistplot(sys1, sys2, ..., w)
nyquistplot(AX, ...)
nyquistplot(..., plotoptions)
```

## Description

`h = nyquistplot(sys)` draws the Nyquist plot of the dynamic system model `sys`. It also returns the plot handle `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands.  
Type

`help nyquistoptions`

for a list of available plot options.

The frequency range and number of points are chosen automatically. See `bode` for details on the notion of frequency in discrete time.

`nyquistplot(sys, {wmin, wmax})` draws the Nyquist plot for frequencies between `wmin` and `wmax` (in `rad/TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`).

`nyquistplot(sys, w)` uses the user-supplied vector `w` of frequencies (in `rad/TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`) at which the Nyquist response is to be evaluated. See `logspace` to generate logarithmically spaced frequency vectors.

`nyquistplot(sys1, sys2, ..., w)` draws the Nyquist plots of multiple models `sys1, sys2, ...` on a single plot. The frequency vector `w` is optional. You can also specify a color, line style, and marker for each system, as in

```
nyquistplot(sys1, 'r', sys2, 'y--', sys3, 'gx')
```

`nyquistplot(AX, ...)` plots into the axes with handle `AX`.



`nyquistplot(..., plotoptions)` plots the Nyquist response with the options specified in `plotoptions`. Type

```
help nyquistoptions
```

for more details.

## Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

## Examples

### Example 1

#### Customize Nyquist Plot Frequency Units

Plot the Nyquist frequency response and change the units to rad/s.

```
sys = rss(5);
h = nyquistplot(sys);
% Change units to radians per second.
setoptions(h, 'FreqUnits', 'rad/s');
```

### Example 2

Compare the frequency responses of identified state-space models of order 2 and 6 along with their 1-std confidence regions rendered at every 50th frequency sample.

```
load iddata1
sys1 = n4sid(z1, 2) % discrete-time IDSS model of order 2
sys2 = n4sid(z1, 6) % discrete-time IDSS model of order 6
```

Both models produce about 76% fit to data. However, `sys2` shows higher uncertainty in its frequency response, especially close to Nyquist frequency as shown by the plot:

```
w = linspace(10, 10*pi, 256);
h = nyquistplot(sys1, sys2, w);
setoptions(h, 'ConfidenceRegionDisplaySpacing', 50, 'ShowFullContour');
```

# nyquistplot

---

Right-click to turn on the confidence region characteristic by using the **Characteristics-> Confidence Region**.

## **See Also**

`getoptions` | `nyquist` | `setoptions` | `showConfidence`

**Purpose**

Estimate Output-Error polynomial model using time or frequency domain data

**Syntax**

```
sys = oe(data,[nb nf nk])
sys = oe(data,[nb nf nk],Name,Value)
sys = oe(data,init_sys)
sys = oe(data, __ ,opt)
```

**Description**

`sys = oe(data,[nb nf nk])` estimates an Output Error model, `sys`, represented by:

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

Here,  $y(t)$  is the output,  $u(t)$  is the input and  $e(t)$  is the error.

`sys` is estimated for the time- or frequency-domain, measured input-output data, `data`. The orders, `[nb nf nk]`, parameterize the estimated polynomial.

`sys = oe(data,[nb nf nk],Name,Value)` specifies model structure attributes using additional options specified by one or more `Name,Value` pair arguments.

`sys = oe(data,init_sys)` uses the Output-Error structure polynomial model (`idpoly`) `init_sys` to configure the initial parameterization of `sys`.

`sys = oe(data, __ ,opt)` estimates a polynomial model using the option set, `opt`, to specify estimation behavior.

**Tips**

- If `data` represents continuous-time frequency response data and you want to estimate a continuous-time model, omit the `nk` input argument.

For example, use `sys = oe(data,[nb nf])`.

## Input Arguments

### **data**

Estimation data.

For time domain estimation, **data** is an `iddata` object containing the input and output signal values.

For frequency domain estimation, **data** can be one of the following:

- recorded frequency response data (`frd` or `idfrd`)
- `iddata` object with its properties specified as follows:
  - `InputData` — Fourier transform of the input signal
  - `OutputData` — Fourier transform of the output signal
  - `Domain` — 'Frequency'

For multi-experiment data, the sample times and inter-sample behavior of all the experiments must match.

### **[nb nf nk]**

Output error model orders.

For a system represented by:

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

Where,  $y(t)$  is the output,  $u(t)$  is the input and  $e(t)$  is the error.

- `nb` — Order of the  $B$  polynomial + 1. `nb` is an  $N_y$ -by- $N_u$  matrix, where  $N_y$  is the number of outputs and  $N_u$  is the number of inputs.
- `nf` — Order of the  $F$  polynomial. `nf` is an  $N_y$ -by- $N_u$  matrix, where  $N_y$  is the number of outputs and  $N_u$  is the number of inputs.
- `nk` — Input delay, expressed as the number of samples. `nk` is an  $N_y$ -by- $N_u$  matrix, where  $N_y$  is the number of outputs and  $N_u$  is the number of inputs. The delay appears as leading zeros of the  $B$  polynomial.

For estimation using continuous-time data, only specify [nb nf]. That is, omit nk.

### **init\_sys**

Polynomial model that configures the initial parameterization of `sys`.

Specify `init_sys` as an `idpoly` model having the Output-Error structure.

Use the `Structure` property of `init_sys` to configure initial guesses and constraints for  $B(q)$  and  $F(q)$ .

To specify an initial guess for, say, the  $F(q)$  term of `init_sys`, set `init_sys.Structure.f.Value` as the initial guess.

To specify constraints for, say, the  $B(q)$  term of `init_sys`:

- set `init_sys.Structure.b.Minimum` to the minimum  $B(q)$  coefficient values
- set `init_sys.Structure.b.Maximum` to the maximum  $B(q)$  coefficient values
- set `init_sys.Structure.b.Free` to indicate which  $B(q)$  coefficients are free for estimation

If `opt` is not specified, and `init_sys` was created by estimation, then the estimation options from `init_sys.Report.OptionsUsed` are used.

### **opt**

Estimation options.

`opt` is an options set that specifies estimation options. These options include the following:

- estimation objective
- handling of initial conditions
- numerical search method and the associated options

Use `oeOptions` to create the options set.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

## InputDelay

Input delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. For continuous-time systems, specify input delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify input delays in integer multiples of the sampling period `Ts`. For example, `InputDelay = 3` means a delay of three sampling periods.

For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector, where each entry is a numerical value representing the input delay for the corresponding input channel. You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0 for all input channels

## ioDelay

Transport delays. `ioDelay` is a numeric array specifying a separate transport delay for each input/output pair.

For continuous-time systems, specify transport delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify transport delays as integers denoting delay of a multiple of the sampling period `Ts`. You can specify `ioDelay` as an alternative to the `nk` value. Doing so simplifies the model structure by reducing the number of leading zeros the  $B$  polynomial. In particular, you can represent  $\max(nk-1, 0)$  leading zeros as input/output delays using `ioDelay` instead.

For a MIMO system with `Ny` outputs and `Nu` inputs, set `ioDelay` to a `Ny`-by-`Nu` array, where each entry is a numerical value representing the

transport delay for the corresponding input/output pair. You can also set `ioDelay` to a scalar value to apply the same delay to all input/output pairs.

**Default:** 0 for all input/output pairs

## Output Arguments

**sys**

Identified Output Error polynomial model.

`sys` is an `idpoly` model which encapsulates the identified Output Error model and the associated parameter covariance data.

## Definitions

### Output-Error (OE) Model

The general Output-Error model structure is:

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

The orders of the Output-Error model are:

$$\begin{aligned} nb: B(q) &= b_1 + b_2q^{-1} + \dots + b_{nb}q^{-nb+1} \\ nf: F(q) &= 1 + f_1q^{-1} + \dots + f_{nf}q^{-nf} \end{aligned}$$

### Continuous-Time Output-Error Model

If data is continuous-time (frequency-domain) data, `oe` estimates a continuous-time model with transfer function:

$$G(s) = \frac{B(s)}{F(s)} = \frac{b_{nb}s^{(nb-1)} + b_{nb-1}s^{(nb-2)} + \dots + b_1}{s^{nf} + f_{nf}s^{(nf-1)} + \dots + f_1}$$

The orders of the numerator and denominator are `nb` and `nf`, similar to the discrete-time case. However, the delay `nk` has no meaning and you should omit it when specifying model orders for estimation. That is, use `model = oe(data, [nb nf])` when you are using continuous-time

frequency domain data. Use the `ioDelay` model property to specify any input-output delays. For example, use `model = oe(data, [nb nf], 'ioDelay', iod)` instead.

## Examples

Estimating Output-Error (OE) model of the type

$$G(s) = \frac{b}{s^3 + f_1 s^2 + f_2 s + f_3} :$$

```
% Use fast sampled data (Ts = 0.001)
% from a plant with bandwidth of about 500 rad/s.
z = iddata(y,u,0.001);
zf = fft(z);
zf.ts = 0;
opt = oeOptions('Focus',[0 500]);
m = oe(zf,[1 3],opt);
```

---

Fit continuous-time transfer function to frequency response.

Generate data.

```
sys1 = tf([1 3],[1 2 1 1]); % TF requires Control System Toolbox
data = idfrd(sys1,logspace(-2,2,256)); % continuous-time FR data
```

Estimate an OE model to fit the data.

```
sys = oe(data, [2 3]); % use syntax OE(DATA, [nb, nf])
bode(data,sys) % compare data to model
```

## Algorithms

The estimation algorithm minimizes prediction errors.

## Alternatives

Output Error models are a special configuration of polynomial models, having only two active polynomials -  $B$  and  $F$ . For such models, it may be more convenient to use a transfer function (`idtf`) model and its estimation command `tfest`.



Also, `tfest` is the recommended command for estimating continuous-time models.

**See Also**

`oeOptions` | `tfest` | `arx` | `armax` | `iv4` | `n4sid` | `bj` |  
`polyest` | `idpoly` | `iddata` | `idfrd` | `sim` | `compare`

# oeOptions

---

**Purpose** Option set for oe

**Syntax**  
opt = oeOptions  
opt = oeOptions(Name,Value)

**Description** opt = oeOptions creates the default options set for oe.  
opt = oeOptions(Name,Value) creates an option set with the options specified by one or more Name,Value pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments, where Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

### InitialCondition

Specify how initial conditions are handled during estimation.

InitialCondition takes one of the following values:

- 'zero' — The initial conditions are set to zero.
- 'estimate' — The initial conditions are treated as independent estimation parameters.
- 'backcast' — The initial conditions are estimated using the best least squares fit.
- 'auto' — The software chooses the method to handle initial conditions based on the estimation data.

**Default:** 'auto'

### Focus

Defines how the errors  $e$  between the measured and the modeled outputs are weighed at specific frequencies during the minimization of the prediction error.

Higher weighting at specific frequencies emphasizes the requirement for a good fit at these frequencies.

**Focus** can take the following values:

- '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.

This method provides a stable model.

- prediction — Automatically calculates the weighting function as a product of the input spectrum and the inverse of the noise model. The weighting function minimizes the one-step-ahead prediction. This approach typically favors fitting small time intervals (higher frequency range). From a statistical-variance point of view, this weighting function is optimal. However, this method neglects the approximation aspects (bias) of the fit. Use 'stability' when you want to ensure a stable model.
- Passbands — Row vector or matrix containing frequency values that define desired passbands. For example:

```
[w1,wh]
[w11,w1h;w21,w2h;w31,w3h;...]
```

where  $w1$  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.

- SISO filter — Enter any SISO linear filter in any of the following ways:
  - A single-input-single-output (SISO) linear system

- The {A,B,C,D} format, which specifies the state-space matrices of the filter
- The {numerator, denominator} format, which specifies the numerator and denominator of the filter transfer function

This option calculates the weighting function as a product of the filter and the input spectrum to estimate the transfer function. To obtain a good model fit for a specific frequency range, you must choose the filter with a passband in this range. You receive an estimation result that is the same as if you had first prefiltered using `idfilt`.

- Weighting vector — 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.

**Default:** 'prediction'

## **EstCovar**

Controls whether parameter covariance data is generated or not.

If `EstCovar` is true, then use `getcov` to fetch the covariance matrix from the estimated model.

**Default:** true

## **Display**

Specifies whether estimation progress should be displayed.

`Display` requires one of the following strings:

- 'on' — Information on model structure and estimation results are displayed in a progress viewer window
- 'off' — No progress or results information is displayed

**Default:** 'off'

### **InputOffset**

Removes offset from time domain input data during estimation.

Specify as a column vector of length  $Nu$ , where  $Nu$  is the number of inputs.

Use [] to indicate no offset.

For multiexperiment data, specify **InputOffset** as a  $Nu$ -by- $Ne$  matrix.  $Nu$  is the number of inputs, and  $Ne$  is the number of experiments.

Each entry specified by **InputOffset** is subtracted from the corresponding input data.

**Default:** []

### **OutputOffset**

Removes offset from time domain output data during estimation.

Specify as a column vector of length  $Ny$ , where  $Ny$  is the number of outputs.

Use [] to indicate no offset.

For multiexperiment data, specify **OutputOffset** as a  $Ny$ -by- $Ne$  matrix.  $Ny$  is the number of outputs, and  $Ne$  is the number of experiments. Each entry specified by **OutputOffset** is subtracted from the corresponding output data.

**Default:** []

### **SearchMethod**

Search method used for iterative parameter estimation.

**SearchMethod** is a string that can take the following values:

- gn — The subspace Gauss-Newton direction. Singular values of the Jacobian matrix less than  $GnPinvConst * \epsilon * \max(\text{size}(J)) * \text{norm}(J)$

are discarded when computing the search direction.  $J$  is the Jacobian matrix. The Hessian matrix is approximated by  $J^T J$ . If there is no improvement in this direction, the function tries the gradient direction.

- **gna** — An adaptive version of subspace Gauss-Newton approach, suggested by Wills and Ninness [1]. Eigenvalues less than  $\text{gamma} \cdot \max(sv)$  of the Hessian are ignore, 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 `Advanced` for more information). This value is increased by the factor `LMStep` each time the search fails to find a lower value of the criterion in less than 5 bisections. This value is decreased by the factor  $2 \cdot \text{LMStep}$  each time a search is successful without any bisections.
- **lm** — Uses the Levenberg-Marquardt method. This means that the next parameter value is  $-\text{pinv}(H+d \cdot I) \cdot \text{grad}$  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.
- **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.
- **grad** — The steepest descent gradient search method.
- **auto** — A choice among the above is made in the algorithm. The descent direction is calculated using `gn`, `gna`, `lm` and `grad` successively, in that order, at each iteration until a sufficient reduction in error is achieved.

**Default:** 'auto'

## SearchOption

`SearchOption` is an options set for the search algorithm with the fields:

SearchMethod	SearchOption
gn,gna,lm,grad and auto	<ul style="list-style-type: none"> <li>• <b>Tolerance</b> — 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 <b>Tolerance</b>, the iterations are stopped. 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. <b>Default:</b> 0.01</li> <li>• <b>MaxIter</b> — Maximum number of iterations during loss-function minimization. The iterations stop when <b>MaxIter</b> is reached or another stopping criterion is satisfied, such as <b>Tolerance</b>. Setting <b>MaxIter</b> = 0 returns the result of the startup procedure. Use <code>sys.Report.Termination.Iterations</code> to get the actual number of iterations during an estimation, where <code>sys</code> is an <code>idtf</code> model. <b>Default:</b> 20</li> <li>• <b>Advanced</b> — Search settings: <ul style="list-style-type: none"> <li>▪ <b>GnPinvConst</b> — Singular values of the Jacobian that are smaller than <math>\text{GnPinvConst} * \max(\text{size}(J)) * \text{norm}(J) * \text{eps}</math> are discarded when computing the search direction and <b>SearchMethod</b> is 'gn'. <b>GnPinvConst</b> must be a positive real value. <b>Default:</b> 10000</li> <li>▪ <b>InitGnaTol</b> — Initial value of <i>gamma</i> when <b>SearchMethod</b> is 'gna'. <b>Default:</b> .0001</li> </ul> </li> </ul>

SearchMethod	SearchOption
	<ul style="list-style-type: none"><li data-bbox="583 361 1248 453">▪ <b>LMStartValue</b> — Starting value of search-direction length <math>d</math> in the Levenberg-Marquardt method. Applicable when <b>SearchMethod</b> is 'lm'. <b>Default:</b> .001</li><li data-bbox="583 522 1267 678">▪ <b>LMStep</b> — Size of the Levenberg-Marquardt step. The next value of the search-direction length <math>d</math> in the Levenberg-Marquardt method is <b>LMStep</b> times the previous one. Applicable when <b>SearchMethod</b> is 'lm'. <b>Default:</b> 2</li><li data-bbox="583 748 1233 852">▪ <b>MaxBisections</b> — Maximum number of bisections used by the line search along the search direction. <b>Default:</b> 25</li><li data-bbox="583 878 1267 1034">▪ <b>MaxFunEvals</b> — Iterations are stopped if the number of calls to the model file exceeds this value. MaxFunEvals must be a positive integer value. <b>Default:</b> Inf</li><li data-bbox="583 1052 1267 1208">▪ <b>MinParChange</b> — Smallest parameter update allowed per iteration MinParChange must be a positive, real value. <b>Default:</b> 0</li><li data-bbox="583 1225 1267 1407">▪ <b>RelImprovement</b> — Iterations are stopped if the relative improvement of the criterion function is less than <b>RelImprovement</b>. RelImprovement must be a positive integer value. <b>Default:</b> 0</li></ul>



SearchMethod	SearchOption
	<ul style="list-style-type: none"> <li>▪ <b>StepReduction</b> — Suggested parameter update is reduced by the factor <code>StepReduction</code> after each try until either <code>MaxBisections</code> tries are completed or a lower value of the criterion function is obtained.</li> </ul> <p><code>StepReduction</code> must be a positive, real value that is greater than 1.</p> <p><b>Default:</b> 2</p>
lsqnonlin	<ul style="list-style-type: none"> <li>• <b>TolFun</b> — Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values.</li> </ul> <p>The value of <code>TolFun</code> is the same as that of <code>sys.SearchOption.Advanced.TolFun</code>.</p> <p><b>Default:</b> 1e-5</p> <ul style="list-style-type: none"> <li>• <b>TolX</b> — Termination tolerance on the estimated parameter values.</li> </ul> <p>The value of <code>TolX</code> is the same as that of <code>sys.SearchOption.Advanced.TolX</code>.</p> <p><b>Default:</b> 1e-6</p> <ul style="list-style-type: none"> <li>• <b>MaxIter</b> — Maximum number of iterations during loss-function minimization. The iterations stop when <code>MaxIter</code> is reached or another stopping criterion is satisfied, such as <code>TolFun</code> etc..</li> </ul> <p>The value of <code>MaxIter</code> is the same as that of <code>sys.SearchOption.Advanced.MaxIter</code>.</p> <p><b>Default:</b> 20</p> <ul style="list-style-type: none"> <li>• <b>Advanced</b> — Options set for <code>lsqnonlin</code>.</li> </ul> <p>For more information, see “Optimization Options”.</p>

Search Method	Search Option
	Use <code>optimset('lsqnonlin')</code> to create an options set for <code>lsqnonlin</code> and then modify it to specify its various options.

## Advanced

Advanced is a structure with the following fields:

- **ErrorThreshold** — Specifies when to adjust the weight of large errors from quadratic to linear.

Errors larger than **ErrorThreshold** times the estimated standard deviation have a linear weight in the 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 the chapter “Computing the Estimate” of [2].)

**ErrorThreshold** = 0 disables robustification and leads to a purely quadratic criterion. When estimating with frequency-domain data, **ErrorThreshold** is set to zero. **ErrorThreshold** = 1.6 is a good choice for estimation when data contains outliers.

**Default:** 0

- **MaxSize** — Specifies the maximum number of elements in a segment when input-output data is split into segments.

**MaxSize** must be a positive integer.

**Default:** 250000

- **StabilityThreshold** — Specifies thresholds for stability tests.

**StabilityThreshold** is a structure with the following fields:

- **s** — Specifies the location of the right-most pole to test the stability of continuous-time models. A model is considered stable when its right-most pole is to the left of **s**.

**Default:** 0

- `z` — 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 `z` from the origin.

**Default:** `1+sqrt(eps)`

- `AutoInitThreshold` — Specifies when to automatically estimate the initial conditions.

When `InitialState = 'Auto'`, the initial condition is estimated when the ratio of the prediction-error norm with a zero initial condition to the norm with an estimated initial condition exceeds `AutoInitialState`.

**Default:** `1.05`

## Output Arguments

### **opt**

Option set containing the specified options for `oe`.

## Examples

### **Create Default Options Set for Output-Error Estimation**

Create a default options set for `oe`.

```
opt = oeOptions;
```

### **Specify Options for Output-Error Estimation**

Create an options set for `oe` using the `'backcast'` algorithm to initialize the state and set the `Display` to `'on'`.

```
opt = oeOptions('InitialState','backcast','Display','on');
```

Alternatively, use dot notation to set the values of `opt`.

```
opt = oeOptions;
opt.InitialState = 'backcast';
opt.Display = 'on';
```

## References

[1] Wills, Adrian , B. Ninness, and S. Gibson. "On Gradient-Based Search for Multivariable System Estimates" IFAC World Congress, Prague, 2005.

[2] Ljung, L. *System Identification: Theory for the User*. Upper Saddle River, NJ: Prentice-Hall PTR, 1999.

## See Also

oe | idfilt

<b>Purpose</b>	Construct operating point specification object for idnlarx model
<b>Syntax</b>	<code>SPEC = operspec(NLSYS)</code>
<b>Description</b>	<code>SPEC = operspec(NLSYS)</code> creates an operating point specification object for the <code>idnlarx</code> model <code>NLSYS</code> . The object encapsulates constraints on input and output signal values. These specifications are used to determine an operating point of the <code>idnlarx</code> model using <code>findop(idnlarx)</code> .
<b>Input Arguments</b>	<ul style="list-style-type: none"><li>• <code>NLSYS</code>: <code>idnlarx</code> model.</li></ul>
<b>Output Arguments</b>	<ul style="list-style-type: none"><li>• <code>SPEC</code>: Operating point specification object. <code>SPEC</code> contains the following properties:<ul style="list-style-type: none"><li>▪ <code>Input</code>: Structure with fields:<ul style="list-style-type: none"><li>• <code>Value</code>: 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.</li><li>• <code>Min</code>: Minimum value constraint on values of input signals for the model. Default: <code>-Inf</code> for all channels.</li><li>• <code>Max</code>: Maximum value constraint on values of input signals for the model. Default: <code>Inf</code> for all channels.</li><li>• <code>Known</code>: Specifies when <code>Value</code> is known (fixed) or is an initial guess. Use a logical vector to denote which signals are known (logical 1, or <code>true</code>) and which have to be estimated using <code>findop</code> (logical 0, or <code>false</code>). Default value: <code>true</code>.</li></ul></li><li>▪ <code>Output</code>: Structure with fields:<ul style="list-style-type: none"><li>• <code>Value</code>: Initial guess for the values of the output signals. Default value: Vector of zeros.</li><li>• <code>Min</code>: Minimum value constraint on values of output signals for the model. Default value: <code>-Inf</code>.</li></ul></li></ul></li></ul>

## operspec(idnlarx)

---

- **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 Arguments**

- NLSYS: idnlhw model.

**Output Arguments**

- 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.

# operspec(idnlhw)

---

---

## 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)`



<b>Purpose</b>	Query model order
<b>Syntax</b>	<code>NS = order(sys)</code>
<b>Description</b>	<p><code>NS = order(sys)</code> returns the model order <code>NS</code>. The order of a dynamic system model is the number of poles (for proper transfer functions) or the number of states (for state-space models). For improper transfer functions, the order is defined as the minimum number of states needed to build an equivalent state-space model (ignoring pole/zero cancellations).</p> <p><code>order(sys)</code> is an overloaded method that accepts SS, TF, and ZPK models. For LTI arrays, <code>NS</code> is an array of the same size listing the orders of each model in <code>sys</code>.</p>
<b>Caveat</b>	<p><code>order</code> does not attempt to find minimal realizations of MIMO systems. For example, consider this 2-by-2 MIMO system:</p> <pre>s=tf('s'); h = [1, 1/(s*(s+1)); 1/(s+2), 1/(s*(s+1)*(s+2))]; order(h) ans =          6</pre> <p>Although <code>h</code> has a 3rd order realization, <code>order</code> returns 6. Use <code>order(ss(h, 'min'))</code> to find the minimal realization order.</p>
<b>See Also</b>	<code>pole</code>   <code>balred</code>

**Purpose** Prediction error for an identified model

**Syntax**

```
err = pe(sys,data,K)
err = pe(sys,data,K, ___,opt)
[err,x0e,sys_pred] = pe(sys,data,K, ___,opt)
pe(sys,data,K, ___)
```

**Description** `err = pe(sys,data,K)` returns the K-step prediction error for the output of the identified model `sys`. The prediction error is determined by subtracting the K-step ahead predicted response from the measured output. The prediction error is calculated for the time span covered by `data`. For more information of computation of predicted response, see `predict`.

`err = pe(sys,data,K, ___,opt)` returns the prediction error using the option set `opt` to specify prediction error calculation behavior.

`[err,x0e,sys_pred] = pe(sys,data,K, ___,opt)` also returns the estimated initial state, `x0e`, and a predictor system, `sys_pred`.

`pe(sys,data,K, ___)` plots the prediction error.

## Input Arguments

**sys**  
Identified model.

**data**  
Measured input-output history.

If `sys` is a time-series model, which has no input signals, then specify `data` as an `iddata` object with no inputs. In this case, you may also specify `data` as a matrix of the past time-series values.

**K**  
Prediction horizon.  
Specify `K` as a positive integer that is a multiple of the data sample-time. Use `K = Inf` to compute the pure simulation error.

**Default:** 1

**opt**

Prediction options.

`opt` is an option set that configures the computation of the predicted response. Options that you can specify include:

- handling of initial conditions
- data offsets

Use `peOptions` to create the options set.

## Output Arguments

**err**

Prediction error.

`err` is an `iddata` object.

Outputs up to the time `t-K` and inputs up to the time instant `t` are used to calculate the prediction error at the time instant `t`.

When `K = Inf`, the predicted output is a pure simulation of the system.

For multi-experiment data, `err` contains the prediction error data for each experiment. The time span of the prediction error matches that of the observed data.

**x0e**

Estimated initial states.

`x0e` is returned only for state-space systems.

**sys\_pred**

Predictor system.

`sys_pred` is a dynamic system. When you simulate `sys_pred`, using `[data.OutputData data.InputData]` as the input, it yields an output, `yp`, such that `err.OutputData = data.OutputData - yp`. For

state-space models, `x0e` is used as the initial condition when simulating `sys_pred`.

For discrete-time data, `sys_pred` is always a discrete-time model.

For multi-experiment data, `sys_pred` is an array of models, with one entry for each experiment.

## Examples

### Compute Prediction Error for an ARIX Model

Compute the prediction error for an ARIX model.

Use the error data to compute the variance of the noise source  $e(t)$ .

Obtain noisy data.

```
noise = [(1:150)';(151:-1:2)'];
```

```
load iddata1 z1;  
z1.y = z1.y+noise;
```

`noise` is a triangular wave that is added to the output signal of `z1`, an `iddata` object.

Estimate an ARIX model for the noisy data.

```
sys = arx(z1,[2 2 1],'IntegrateNoise',true);
```

Compute the prediction error of the estimated model.

```
K = 1;  
err = pe(z1,sys,K);
```

`pe` computes the 1-step prediction error for the output of the identified model, `sys`.

Compute the variance of the noise source,  $e(t)$ .

```
noise_var = err.y'*err.y/(299-nparams(sys)-order(sys));
```

Compare the computed value with model's noise variance.

---

`sys.NoiseVariance`

The output of `sys.NoiseVariance` matches the computed variance.

**See Also**

`peOptions` | `predict` | `resid` | `sim` | `lsim` | `compare` | `ar` |  
`arx` | `n4sid` | `iddata` | `idpar`

# peOptions

---

**Purpose** Option set for pe

**Syntax**  
opt = peOptions  
opt = peOptions(Name,Value)

**Description** opt = peOptions creates the default options set for pe.  
opt = peOptions(Name,Value) creates an option set with the options specified by one or more Name,Value pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments, where Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

### InitialCondition

Specify the handling of initial conditions.

InitialCondition takes one of the following:

- 'z' — Zero initial conditions.
- 'e' — Estimate initial conditions such that the prediction error for observed output is minimized.
- 'd' — Similar to 'e', but absorbs nonzero delays into the model coefficients.
- x0 — Numerical column vector denoting initial states. For multi-experiment data, use a matrix with Ne columns, where Ne is the number of experiments. Use this option for state-space and nonlinear models only.
- io — Structure with the following fields:
  - Input
  - Output

Use the `Input` and `Output` fields to specify the input/output history for a time interval that starts before the start time of the data used by `pe`. In case the data used by `pe` is a time series model, specify `Input` as `[]`. Use a row vector to denote a constant signal value. The number of columns in `Input` and `Output` must always equal the number of input and output channels, respectively. For multi-experiment data, specify `io` as a struct array of  $N_e$  elements, where  $N_e$  is the number of experiments.

- `x0obj` — Specification object created using `idpar`. Use this object for discrete-time state-space models only. Use `x0obj` to impose constraints on the initial states by fixing their value or specifying minimum/maximum bounds.

**Default:** `[]` (Initial conditions are determined based on how the model was estimated.)

### **InputOffset**

Removes offset from time domain input data during prediction error calculation.

Specify as a column vector of length  $N_u$ , where  $N_u$  is the number of inputs.

For multi-experiment data, specify `InputOffset` as an  $N_u$ -by- $N_e$  matrix. Here  $N_u$  is the number of inputs and  $N_e$  is the number of experiments.

Each entry specified by `InputOffset` will be subtracted from the corresponding input data.

Specify input offset for only time domain data.

**Default:** `[]`

### **OutputOffset**

Removes offset from time domain output data during prediction error calculation.

Specify as a column vector of length  $N_y$ , where  $N_y$  is the number of outputs.

In case of multi-experiment data, specify `OutputOffset` as a  $N_y$ -by- $N_e$  matrix. Here  $N_y$  is the number of outputs and  $N_e$  is the number of experiments.

Each entry specified by `OutputOffset` will be subtracted from the corresponding output data.

Specify output offset for only time domain data.

**Default:** []

## **OutputWeight**

Weight of output for initial condition estimation.

`OutputWeight` takes one of the following:

- [] — No weighting is used. This is the same as using `eye(Ny)` for the output weight, where  $N_y$  is the number of outputs.
- 'noise' — Inverse of the noise variance stored with the model.
- matrix — A positive semi-definite matrix of dimension  $N_y$ -by- $N_y$ , where  $N_y$  is the number of outputs.

**Default:** []

## **Output Arguments**

**opt**

Option set containing the specified options for `pe`.

## **Examples**

### **Create Default Options Set for Prediction Error Calculation**

Create a default options set for `pe`.

```
opt = peOptions;
```



## Specify Options for Prediction Error Calculation

Create an options set for `pe` using zero initial conditions and set the input offset to 5.

```
opt = peOptions('InitialCondition','z','InputOffset',5);
```

Alternatively, use dot notation to set the values of `opt`.

```
opt = peOptions;  
opt.InitialCondition = 'z';  
opt.InputOffset = 5;
```

## See Also

`pe` | `idpar`

**Purpose** Prediction error estimate of linear or nonlinear model

**Syntax**

```
sys = pem(data,init_sys)
sys = pem(data,init_sys,opt)
```

**Description** `sys = pem(data,init_sys)` updates the parameters of `init_sys`, a linear or nonlinear model, to fit the given estimation data, `data`. The prediction error minimization algorithm is used to update the free parameters of `init_sys`.

`sys = pem(data,init_sys,opt)` configures the estimation options using the option set `opt`. This syntax is valid for linear models only.

## Input Arguments

### **data**

Estimation data.

Specify `data` as an `iddata` or `idfrd` object containing the measured input/output data.

The input-output dimensions of `data` and `init_sys` must match.

You can specify frequency-domain data only when `init_sys` is a linear model.

### **init\_sys**

Linear or nonlinear identified model that configures the initial parameterization of `sys`.

`init_sys` may be a linear or nonlinear model and must have finite parameter values. You may obtain `init_sys` by performing an estimation using measured data, or by direct construction. `idnlarx` and `idnlhw` models can be obtained only by estimation.

You can configure initial guesses, specify minimum/maximum bounds, and fix or free for estimation any parameter of `init_sys`.

- For linear models, use the `Structure` property. For more information, see “Imposing Constraints on the Values of Model Parameters”.

- For nonlinear models grey-box models, use the `InitialStates` and `Parameters` properties. Parameter constraints cannot be prescribed for nonlinear ARX and Hammerstein-Wiener models.

### **opt**

Estimation options.

`opt` is an option set that specifies:

- estimation algorithm settings
- handling of the estimating focus
- initial conditions
- data offsets.

You can specify an option set only when `init_sys` is a linear model.

You must create an option set using one of the following functions. The function used to create the option set depends on the initial model type.

<b>Model Type</b>	<b>Option Set Function</b>
<code>idss</code>	<code>ssestOptions</code>
<code>idtf</code>	<code>tfestOptions</code>
<code>idproc</code>	<code>procestOptions</code>
<code>idpoly</code>	<code>polyestOptions</code>
<code>idgrey</code>	<code>greyestOptions</code>

## **Output Arguments**

### **sys**

Identified model.

`sys` is obtained by estimating the free parameters of `init_sys` using the prediction error minimization algorithm.

## Examples

### Refine Estimated State-Space Model

Estimate a discrete-time state-space model using the subspace method and refine it by minimizing the prediction error.

Estimate a discrete-time state-space model using `n4sid`, which applies the subspace method.

```
load iddata7 z7;  
z7a = z7(1:300);  
opt = n4sidOptions('Focus','simulation');  
init_sys = n4sid(z7a,4,opt);
```

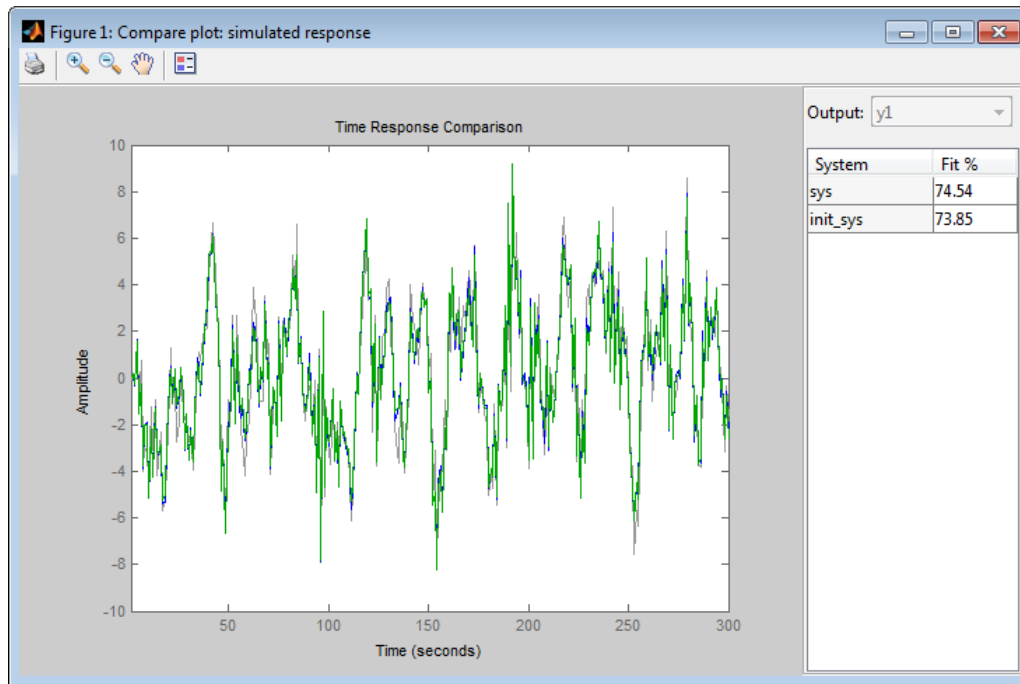
`init_sys`, the estimated state-space model, provides a 73.85% fit to the estimation data (see `init_sys.Report.Fit.FitPercent`). Use `pem` to improve the closeness of the fit.

Obtain a refined estimated model by using `pem`.

```
sys = pem(z7a,init_sys);
```

Analyze the results.

```
compare(z7a,sys,init_sys);
```



sys refines the estimated model and provides a 74.54% fit to the estimation data (see `init_sys.Report.Fit.FitPercent`).

### Configure Estimation Using Process Model

Create a process model structure and update its parameter values to minimize prediction error.

Create a process model and initialize its coefficients.

```
init_sys = idproc('P2UDZ');  
init_sys.Kp = 10;  
init_sys.Tw = 0.4;  
init_sys.Zeta = 0.5;  
init_sys.Td = 0.1;  
init_sys.Tz = 0.01;
```

Use `init_sys`, a process model created by direct construction, to configure the estimation by `pem`. The  $K_p$ ,  $T_w$ ,  $Zeta$ ,  $T_d$  and  $T_z$  coefficients of `init_sys` are configured with their initial guesses.

Estimate a prediction error minimizing model using measured data.

```
load iddata1 z1;
opt = procestOptions('Display','on','SearchMethod','lm');
sys = pem(z1,init_sys,opt);
```

As `init_sys` is an `idproc` model, use the corresponding option set command, `procestOptions` to create an estimation configuring option set. The estimation is set to use the Levenberg-Marquardt search method ('SearchMethod','lm') and will display the iterative search results during estimation ('Display','on').

`sys` is an estimated process model, which provides a 70.63% fit to the measured data (see `sys.Report.Fit.FitPercent`).

## Estimate Nonlinear Grey-Box Model

Estimate the parameters of a nonlinear grey box model to fit DC motor data.

Load the experimental data, and specify the signal attributes such as start time, and units.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'dcmotor...
data = iddata(y, u, 0.1);
set(data, 'Tstart', 0, 'TimeUnit', 's');
```

Configure the nonlinear grey-box model (`idnlgrey`) model.

For this example, use the shipped file `dcmotor_m.m`. To view this file, enter `edit dcmotor_m.m` at the MATLAB command prompt.

```
file_name = 'dcmotor_m';
order = [2 1 2];
parameters = [1; 0.28];
initial_states = [0; 0];
```

```
Ts = 0;  
init_sys = idnlgrey(file_name,order,parameters,initial_states,Ts);  
set(init_sys,'TimeUnit','s');  
  
setinit(init_sys,'Fixed',{false false});
```

`init_sys` is a nonlinear grey-box model with its structure described by `dcmotor_m.m`. The model has one input, two outputs and two states, as specified by `order`.

`setinit(init_sys,'Fixed',{false false})` specifies that the initial states of `init_sys` are free estimation parameters.

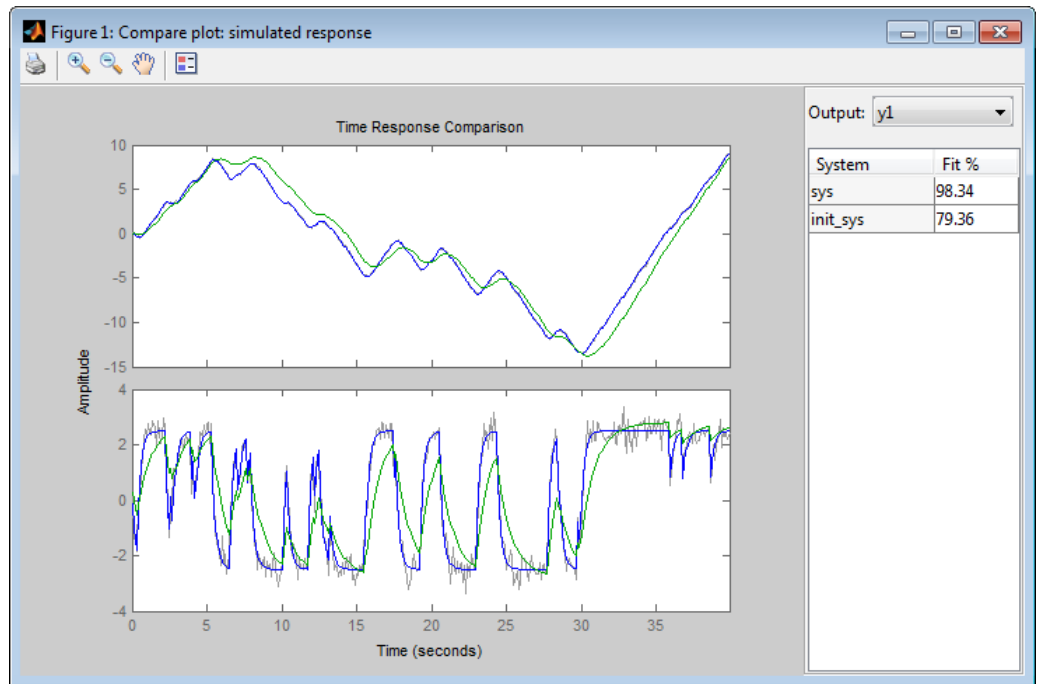
Estimate the model parameters and initial states.

```
sys = pem(data,init_sys);
```

`sys` is an `idnlgrey` model, which encapsulates the estimated parameters and their covariance.

Analyze the estimation result.

```
compare(data,sys,init_sys);
```



sys provides a 98.34% fit to the estimated data.

## Algorithms

PEM uses numerical optimization to minimize the *cost function*, a weighted norm of the prediction error, 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:



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

$e(t)$  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.

## Alternatives

You can use model type specific estimation commands for all model types, except for `idnlgrey` models. These commands achieve the same results as `pem` when an initial model of matching type is provided as input argument. The following table summarizes the dedicated estimation commands for each model type.

Function	Model Type
<code>ssest</code>	<code>idss</code>
<code>tfest</code>	<code>idtf</code>
<code>polyest</code>	<code>idpoly</code>
<code>procest</code>	<code>idproc</code>
<code>greyest</code>	<code>idgrey</code>
<code>nlarx</code>	<code>idnlarx</code>
<code>nlhw</code>	<code>idnlhw</code>

## See Also

`idtf` | `idpoly` | `idss` | `idgrey` | `idproc` | `armax` | `oe` | `bj` | `n4sid` | `ssest` | `tfest` | `procest` | `greyest` | `nlhw` | `nlarx` | `resid` | `compare` | `idfilt` | `iddata` | `idfrd` | `tfestOptions` | `procestOptions` | `polyestOptions` | `greyestOptions` | `ssestOptions`

# pexcit

---

**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` | `feedback` | `idnlarx`

**Purpose**

Plot iddata or model objects

**Syntax**

```
plot(data)
plot(d1,...,dN)
plot(d1,PlotStyle1,...,dN,PlotStyleN)
plot(model)
```

**Description**

data is the output-input data to be graphed, given as an iddata object. A split plot is obtained with the outputs on top and the inputs at the bottom.

One plot for each I/O channel combination is produced. Pressing the **Enter** key advances the plot. Typing **Ctrl+C** aborts the plotting in an orderly fashion.

To plot a specific interval, use `plot(data(200:300))`. To plot specific input/output channels, use `plot(data(:,ky,ku))`, consistent with the subreferencing of iddata objects.

If `data.intersample = 'zoh'`, the input is piecewise constant between sampling points, and it is then graphed accordingly.

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

With `PlotStyle`, the color, line style, and marker of each data set can be specified

```
plot(d1,'y:*',d2,'b')
```

just as in the regular plot command.

model is an idnlarx, or idnlhw model.

**See Also**

iddata

# pole

---

**Purpose** Compute poles of dynamic system

**Syntax** `pole(sys)`

**Description** `pole(sys)` computes the poles  $p$  of the SISO or MIMO dynamic system model `sys`.

If `sys` has internal delays, poles are obtained by first setting all internal delays to zero (creating a zero-order Padé approximation) so that the system has a finite number of zeros. For some systems, setting delays to 0 creates singular algebraic loops, which result in either improper or ill-defined, zero-delay approximations. For these systems, `pole` returns an error. This error does not imply a problem with the model `sys` itself.

**Algorithms** For state-space models, the poles are the eigenvalues of the  $A$  matrix, or the generalized eigenvalues of  $A - \lambda E$  in the descriptor case.

For SISO transfer functions or zero-pole-gain models, the poles are simply the denominator roots (see `roots`).

For MIMO transfer functions (or zero-pole-gain models), the poles are computed as the union of the poles for each SISO entry. If some columns or rows have a common denominator, the roots of this denominator are counted only once.

**Limitations** Multiple poles are numerically sensitive and cannot be computed to high accuracy. A pole  $\lambda$  with multiplicity  $m$  typically gives rise to a cluster of computed poles distributed on a circle with center  $\lambda$  and radius of order

$$\rho \approx \varepsilon^{1/m}$$

where  $\varepsilon$  is the relative machine precision (`eps`).

**See Also** `damp` | `esort` | `dsort` | `pzmap` | `zero`

**Purpose** Access polynomial coefficients and uncertainties of identified model

**Syntax**

```
[A,B,C,D,F] = polydata(sys)
[A,B,C,D,F,dA,dB,dC,dD,dF] = polydata(sys)
___ = polydata(sys,J1,...,JN)
___ = polydata(___, 'cell')
```

**Description** [A,B,C,D,F] = polydata(sys) returns the coefficients of the polynomials A, B, C, D, and F that describe the identified model **sys**. The polynomials describe the idpoly representation of **sys** as follows.

- For discrete-time **sys**:

$$A(q^{-1})y(t) = \frac{B(q^{-1})}{F(q^{-1})}u(t-nk) + \frac{C(q^{-1})}{D(q^{-1})}e(t).$$

$u(t)$  are the inputs to **sys**.  $y(t)$  are the outputs.  $e(t)$  is a white noise disturbance.

- For continuous-time **sys**:

$$A(s)Y(s) = \frac{B(s)}{F(s)}U(s)e^{-\tau s} + \frac{C(s)}{D(s)}E(s).$$

$U(s)$  are the Laplace transformed inputs to **sys**.  $Y(s)$  are the Laplace transformed outputs.  $E(s)$  is the Laplace transform of a white noise disturbance.

If **sys** is an identified model that is not an idpoly model, **polydata** converts **sys** to idpoly form to extract the polynomial coefficients.

[A,B,C,D,F,dA,dB,dC,dD,dF] = polydata(sys) also returns the uncertainties dA, dB, dC, dD, and dF of each of the corresponding polynomial coefficients of **sys**.

\_\_\_ = polydata(sys,J1,...,JN) returns the polynomial coefficients for the J1,...,JN entry in the array **sys** of identified models.

`___ = polydata( ___, 'cell' )` returns all polynomials as cell arrays of double vectors, regardless of the input and output dimensions of `sys`.

## Input Arguments

### **sys**

Identified model or array of identified models. `sys` can be continuous-time or discrete-time. `sys` can be SISO or MIMO.

### **J1,...,JN**

Indices selecting a particular model from an N-dimensional array `sys` of identified models.

## Output Arguments

### **A,B,C,D,F**

Polynomial coefficients of the `idpoly` representation of `sys`.

- If `sys` is a SISO model, each of A, B, C, D, and F is a row vector. The length of each row vector is the order of the corresponding polynomial.
  - For discrete-time `sys`, the coefficients are ordered in ascending powers of  $q^{-1}$ . For example,  $B = [1 \ -4 \ 9]$  means that  $B(q^{-1}) = 1 - 4q^{-1} + 9q^{-2}$ .
  - For continuous-time `sys`, the coefficients are ordered in descending powers of  $s$ . For example,  $B = [1 \ -4 \ 9]$  means that  $B(s) = s^2 - 4s + 9$ .
- If `sys` is a MIMO model, each of A, B, C, D, and F is a cell array. The dimensions of the cell arrays are determined by the input and output dimensions of `sys` as follows:
  - A —  $N_y$ -by- $N_y$  cell array
  - B, F —  $N_y$ -by- $N_u$  cell array
  - C, D —  $N_y$ -by-1 cell array

$N_y$  is the number of outputs of `sys`, and  $N_u$  is the number of inputs.

Each entry in a cell array is a row vector that contains the coefficients of the corresponding polynomial. The polynomial coefficients are ordered the same way as the SISO case.

**dA,dB,dC,dD,dF**

Uncertainties in the estimated polynomial coefficients of `sys`.

`dA`, `dB`, `dC`, `dD`, and `dF` are row vectors or cell arrays whose dimensions exactly match the corresponding `A`, `B`, `C`, `D`, and `F` outputs.

Each entry in `dA`, `dB`, `dC`, `dD`, and `dF` gives the standard deviation of the corresponding estimated coefficient. For example, `dA{1,1}(2)` gives the standard deviation of the estimated coefficient returned at `A{1,1}(2)`.

**Examples****Polynomial Coefficients of Identified Model and Uncertainties**

Extract the polynomial coefficients, and corresponding standard deviations, of a two-input, two-output identified `idpoly` model.

Load system data and estimate a 2-input, 2-output model.

```
load iddata1 z1
load iddata2 z2
data = [z1 z2(1:300)];

nk = [1 1; 1 0];
na = [2 2; 1 3];
nb = [2 3; 1 4];
nc = [2;3];
nd = [1;2];
nf = [2 2;2 1];

sys = polyest(data,[na nb nc nd nf nk]);
```

The data loaded into `z1` and `z2` is discrete-time `iddata` with a sampling time of 0.1 s. Therefore, `sys` is a two-input, two-output discrete-time `idpoly` model of the form:

$$A(q^{-1})y(t) = \frac{B(q^{-1})}{F(q^{-1})}u(t - nk) + \frac{C(q^{-1})}{D(q^{-1})}e(t).$$

The inputs to `polyest` set the order of each polynomial in `sys`.

Use `polydata` to access the estimated polynomial coefficients of `sys` and the uncertainties in those coefficients.

```
[A,B,C,D,F,dA,dB,dC,dD,dF] = polydata(sys);
```

The outputs `A`, `B`, `C`, `D`, and `F` are cell arrays of coefficient vectors or arrays. The dimensions of the cell arrays are determined by the input and output dimensions of `sys`. For example, examine `A`.

```
A
```

```
A =
```

```
    [1x3 double]    [1x3 double]  
    [1x2 double]    [1x4 double]
```

`A` is a 2-by-2 cell array because `sys` has two outputs. Each entry in `A` is a row vector containing identified polynomial coefficients. For example, examine the second diagonal entry in `A`.

```
A{2,2}
```

```
ans =
```

```
    1.0000    -0.8682    -0.2244     0.4467
```

For discrete-time `sys`, the coefficients are arranged in order of increasing powers of  $q^{-1}$ . Therefore, `A{2,2}` corresponds to the polynomial  $1 - 0.8682q^{-1} - 0.2244q^{-2} + 0.4467q^{-3}$ .

Examine the uncertainties `dA` in the estimated coefficients of `A`.

```
dA
```

```
dA =
```

```
    [1x3 double]    [1x3 double]
```



```
[1x2 double]    [1x4 double]
```

The dimensions of `dA` match those in `A`. Each entry in `dA` gives the standard deviation of the corresponding estimated polynomial coefficient of `A`. For example, examine the uncertainties of the second diagonal entry in `A`.

```
dA{2,2}
```

```
ans =
```

```
0    0.2806    0.4204    0.2024
```

The lead coefficient of `A{2,2}` is fixed at 1, and therefore has no uncertainty. The remaining entries in `dA{2,2}` are the uncertainties in the  $q^{-1}$ ,  $q^{-2}$ , and  $q^{-3}$  coefficients, respectively.

## See Also

`idpoly` | `iddata` | `tfdata` | `zpkdata` | `idssdata` | `polyest`

# polyest

---

**Purpose** Estimate polynomial model using time or frequency domain data

**Syntax**

```
sys = polyest(data,[na nb nc nd nf nk])
sys = polyest(data,[na nb nc nd nf nk],Name,Value)
sys = polyest(data,init_sys)
sys = polyest(data, __ , opt)
```

**Description** `sys = polyest(data,[na nb nc nd nf nk])` estimates a polynomial model, `sys`, using the time or frequency domain data, `data`.

`sys` is of the form

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

Here,  $A(q)$ ,  $B(q)$ ,  $F(q)$ ,  $C(q)$  and  $D(q)$  are polynomial matrices;  $u(t)$  is the input;  $nk$  is the input delay;  $y(t)$  is the output; and  $e(t)$  is the disturbance signal. `na`, `nb`, `nc`, `nd` and `nf` are the orders of the  $A(q)$ ,  $B(q)$ ,  $C(q)$ ,  $D(q)$  and  $F(q)$  polynomials, respectively.

`sys = polyest(data,[na nb nc nd nf nk],Name,Value)` estimates a polynomial model with additional attributes of the estimated model structure specified by one or more `Name, Value` pair arguments.

`sys = polyest(data,init_sys)` estimates a polynomial model using the dynamic system `init_sys` to configure the initial parameterization.

`sys = polyest(data, __ , opt)` estimates a polynomial model using the option set, `opt`, to specify estimation behavior.

## Tips

- In most situations, all the polynomials of an identified polynomial model are not simultaneously active. Set one or more of the orders `na`, `nc`, `nd` and `nf` to zero to simplify the model structure.

For example, you can create an Output Error (OE) model by specifying `na`, `nc` and `nd` as zero.

- For a simplified polynomial model, you may use a dedicated estimating function, such as `oe`, `bj`, `arx` and `armax`.

## Input Arguments

### **data**

Estimation data.

For time domain estimation, **data** is an `iddata` object containing the input and output signal values.

You can estimate only discrete-time models using time domain data. For estimating continuous-time models using time domain data, see `tfest`.

For frequency domain estimation, **data** can be one of the following:

- recorded frequency response data (`frd` or `idfrd`)
- `iddata` object with its properties specified as follows:
  - `InputData` — Fourier transform of the input signal
  - `OutputData` — Fourier transform of the output signal
  - `Domain` — 'Frequency'

It may be more convenient to use `oe` or `tfest` to estimate a model for frequency domain data.

### **na**

Order of the polynomial  $A(q)$ .

**na** is an  $N_y$ -by- $N_u$  matrix of nonnegative integers.  $N_y$  is the number of outputs and  $N_u$  is the number of inputs.

**na** must be zero if you are estimating a model using frequency domain data.

### **nb**

Order of the polynomial  $B(q) + 1$ .

**nb** is an  $N_y$ -by- $N_u$  matrix of nonnegative integers.  $N_y$  is the number of outputs and  $N_u$  is the number of inputs.

### **nc**

Order of the polynomial  $C(q)$ .

**nc** is a column vector of nonnegative integers of length  $N_y$ .  $N_y$  is the number of outputs.

**nc** must be zero if you are estimating a model using frequency domain data.

## **nd**

Order of the polynomial  $D(q)$ .

**nd** is a column vector of nonnegative integers of length  $N_y$ .  $N_y$  is the number of outputs.

**nd** must be zero if you are estimating a model using frequency domain data.

## **nf**

Order of the polynomial  $F(q)$ .

**nf** is an  $N_y$ -by- $N_u$  matrix of nonnegative integers.  $N_y$  is the number of outputs and  $N_u$  is the number of inputs.

## **nk**

Input-output delay expressed as fixed leading zeros of the  $B$  polynomial.

Specify **nk** as a matrix of nonnegative integers.

**nk** must be zero when estimating a continuous-time model.

## **opt**

Estimation options.

**opt** is an options set that specifies the following:

- estimation objective
- handling of initial conditions
- numerical search method to be used in estimation

Use `polyestOptions` to create the options set.

### **init\_sys**

Dynamic system that configures the initial parameterization of `sys`.

If `init_sys` is an `idpoly` model, `polyest` uses the parameters and constraints defined in `init_sys` as the initial guess for estimating `sys`.

If `init_sys` is not an `idpoly` model, the software first converts `init_sys` to an identified polynomial. `polyest` uses the parameters of the resulting model as the initial guess for estimation.

Use the `Structure` property of `init_sys` to configure initial guesses and constraints for  $A(q)$ ,  $B(q)$ ,  $F(q)$ ,  $C(q)$  and  $D(q)$ .

To specify an initial guess for, say, the  $A(q)$  term of `init_sys`, set `init_sys.Structure.a.Value` as the initial guess.

To specify constraints for, say, the  $B(q)$  term of `init_sys`:

- set `init_sys.Structure.b.Minimum` to the minimum  $B(q)$  coefficient values
- set `init_sys.Structure.b.Maximum` to the maximum  $B(q)$  coefficient values
- set `init_sys.Structure.b.Free` to indicate which  $B(q)$  coefficients are free for estimation

You can similarly specify the initial guess and constraints for the other polynomials.

If `opt` is not specified, and `init_sys` was created by estimation, then the estimation options from `init_sys.Report.OptionsUsed` are used.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **ioDelay**

Transport delays. `ioDelay` is a numeric array specifying a separate transport delay for each input/output pair.

For continuous-time systems, specify transport delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify transport delays as integers denoting delay of a multiple of the sampling period  $T_s$ .

For a MIMO system with  $N_y$  outputs and  $N_u$  inputs, set `ioDelay` to a  $N_y$ -by- $N_u$  array, where each entry is a numerical value representing the transport delay for the corresponding input/output pair. You can also set `ioDelay` to a scalar value to apply the same delay to all input/output pairs.

**Default:** 0 for all input/output pairs

### **InputDelay**

Input delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. For continuous-time systems, specify input delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify input delays in integer multiples of the sampling period  $T_s$ . For example, `InputDelay = 3` means a delay of three sampling periods.

For a system with  $N_u$  inputs, set `InputDelay` to an  $N_u$ -by-1 vector, where each entry is a numerical value representing the input delay for the corresponding input channel. You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0 for all input channels

### **IntegrateNoise**

Logical vector specifying integrators in the noise channel.

`IntegrateNoise` is a logical vector of length  $N_y$ , where  $N_y$  is the number of outputs.

Setting `IntegrateNoise` to true for a particular output results in the model:

$$A(q)y(t) = \frac{B(q)}{F(q)}u(t - nk) + \frac{C(q)}{D(q)} \frac{e(t)}{1 - q^{-1}}$$

Where,  $\frac{1}{1 - q^{-1}}$  is the integrator in the noise channel,  $e(t)$ .  
Use `IntegrateNoise` to create an ARIMAX model.

For example,

```
load iddata1 z1;
z1 = iddata(cumsum(z1.y),cumsum(z1.u),z1.Ts,'InterSample','foh');
sys = polyest(z1, [2 2 2 0 0 1],'IntegrateNoise',true);
```

## Output Arguments

### **sys**

Estimated polynomial model.

`sys` is an `idpoly` model.

If `data.Ts` is zero, `sys` is a continuous-time model representing:

$$Y(s) = \frac{B(s)}{F(s)}U(s) + E(s)$$

Here,  $Y(s)$ ,  $U(s)$  and  $E(s)$  are the Laplace transforms of the time domain signals  $y(t)$ ,  $u(t)$  and  $e(t)$ . respectively.

## Examples

Identify a polynomial model of ARIMAX structure. Use zero values for the orders of the inactive polynomials,  $F$  and  $D$  in this case.

```
load iddata1 z1;
% integrated data
data = iddata(cumsum(z1.y),cumsum(z1.u),z1.Ts,'InterSample','foh');
na = 2; nb = 2; nc = 2; nd = 0; nf = 0; nk = 1;
sys = polyest(data, [na nb nc nd nf nk], true); % "ARIMAX" form
```

---

Estimate a polynomial model with redundant parameterization; all the polynomials ( $A$ ,  $B$ ,  $C$ ,  $D$ , and  $F$ ) are active.

```
load iddata2 z2
na = 2; nb = 2; nc = 3; nd = 3; nf = 2; nk = 1;
sys = polyest(z2, [na nb nc nd nf nk]);
```

---

Estimate a multi-output ARMAX model for a data set obtained by concatenating two single-input, single-output data sets.

```
load iddata1 z1
load iddata2 z2
data = [z1, z2(1:300)];
```

`data` is a data set with 2 inputs and 2 outputs. The first input affects only the first output. Similarly, the second input affects only the second output.

In the estimated ARMAX model, the cross terms, modelling the effect of the first input on the second output and vice-versa, should be negligible. If higher orders are assigned to those dynamics, their estimations will show a high level of uncertainty.

```
na = [2 2; 2 2]; nb = [2 2; 3 4]; nk = [1 1; 0 0];
nc = [2;2]; nd = [0;0]; nf = [0 0; 0 0];
sys = polyest(data, [na nb nc nd nf nk])
```

The  $F$  and  $D$  polynomials of `sys` are inactive.

Analyze the results.

```
h = bodeplot(model);
showConfidence(h,3)
```

The responses from the cross terms show larger uncertainty.



**Alternatives**

- To estimate a polynomial model using time-series data, use `ar`.
- Use `polyest` to estimate a polynomial of arbitrary structure. If the structure of the estimated polynomial model is known, that is, you know which polynomials will be active, then use the appropriate dedicated estimating function. For examples, for an ARX model, use `arx`. Other polynomial model estimating functions include, `oe`, `armax`, and `bj`.
- To estimate a continuous-time transfer function, use `tfest`. You can also use `oe`, but only with continuous-time frequency-domain data.

**See Also**

`polyestOptions` | `idpoly` | `ar` | `arx` | `armax` | `oe` | `bj` | `tfest` | `procest` | `ssest` | `iddata` | `pem` | `forecast`

# polyestOptions

---

**Purpose** Option set for polyest

**Syntax**  
`opt = polyestOptions`  
`opt = polyestOptions(Name,Value)`

**Description** `opt = polyestOptions` creates the default options set for polyest.  
`opt = polyestOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### InitialCondition

Specify how initial conditions are handled during estimation.

`InitialCondition` takes one of the following values:

- 'zero' — The initial condition is set to zero.
- 'estimate' — The initial state is treated as an independent estimation parameter.
- 'backcast' — The initial state is estimated using the best least squares fit.
- 'auto' — The software chooses the method to handle initial states based on the estimation data.

**Default:** 'auto'

### Focus

Defines how the errors  $e$  between the measured and the modeled outputs are weighed at specific frequencies during the minimization of the prediction error.

Higher weighting at specific frequencies emphasizes the requirement for a good fit at these frequencies.

**Focus** can take the following values:

- '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.

This method provides a stable model.

- 'prediction' — 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.
- 'stability' — Same as 'prediction' but with model stability enforced.
- Passbands — Row vector or matrix containing frequency values that define desired passbands. For example:

```
[w1,wh]
[w11,w1h;w21,w2h;w31,w3h;...]
```

where  $w1$  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.

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

- A single-input-single-output (SISO) linear system.
- The {A,B,C,D} format, which specifies the state-space matrices of the filter.
- The {numerator, denominator} format, 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. To obtain a good model fit for a specific frequency range, you must choose the filter with a passband in this range. The estimation result is the same if you first prefilter the data using `idfilt`.

- Weighting vector — 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.

**Default:** 'prediction'

## **EstCovar**

Controls whether parameter covariance data is generated or not.

If `EstCovar` is `true`, then use `getcov` to fetch the covariance matrix from the estimated model.

**Default:** `true`

## **Display**

Specifies whether estimation progress should be displayed.

`Display` requires one of the following strings:

- 'on' — Information on model structure and estimation results are displayed in a progress viewer window
- 'off' — No progress or results information is displayed

**Default:** 'off'

## **InputOffset**

Removes offset from time domain input data during estimation.

Specify as a column vector of length  $Nu$ , where  $Nu$  is the number of inputs.

Use [] to indicate no offset.

For multiexperiment data, specify **InputOffset** as a  $Nu$ -by- $Ne$  matrix.  $Nu$  is the number of inputs, and  $Ne$  is the number of experiments.

Each entry specified by **InputOffset** is subtracted from the corresponding input data.

**Default:** []

## **OutputOffset**

Removes offset from time domain output data during estimation.

Specify as a column vector of length  $Ny$ , where  $Ny$  is the number of outputs.

Use [] to indicate no offset.

For multiexperiment data, specify **OutputOffset** as a  $Ny$ -by- $Ne$  matrix.  $Ny$  is the number of outputs, and  $Ne$  is the number of experiments. Each entry specified by **OutputOffset** is subtracted from the corresponding output data.

**Default:** []

## **SearchMethod**

Search method used for iterative parameter estimation.

**SearchMethod** is a string that can take the following values:

- gn — The subspace Gauss-Newton direction. Singular values of the Jacobian matrix less than  $GnPinvConst * \epsilon * \max(\text{size}(J)) * \text{norm}(J)$

are discarded when computing the search direction.  $J$  is the Jacobian matrix. The Hessian matrix is approximated by  $J^T J$ . If there is no improvement in this direction, the function tries the gradient direction.

- **gna** — An adaptive version of subspace Gauss-Newton approach, suggested by Wills and Ninness [2]. Eigenvalues less than  $\text{gamma} \cdot \max(\text{sv})$  of the Hessian are neglected, where  $\text{sv}$  are the singular values of the Hessian. The Gauss-Newton direction is computed in the remaining subspace. *gamma* has the initial value `InitGnaTol` (see `Advanced` for more information) 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  $2 \cdot \text{LMStep}$  each time a search is successful without any bisections.
- **lm** — Uses the Levenberg-Marquardt method. This means that the next parameter value is  $-\text{pinv}(H+d \cdot I) \cdot \text{grad}$  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.
- **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.
- **grad** — The steepest descent gradient search method.
- **auto** — A choice among the above is made in the algorithm. The descent direction is calculated using `gn`, `gna`, `lm` and `grad` successively, in that order, at each iteration until a sufficient reduction in error is achieved.

**Default:** 'auto'

## SearchOption

`SearchOption` is an options set for the search algorithm with the fields:

SearchMethod	SearchOption
gn,gna,lm,grad and auto	<ul style="list-style-type: none"> <li>• <b>Tolerance</b> — 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 <b>Tolerance</b>, the iterations are stopped. 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. <b>Default:</b> 0.01</li> <li>• <b>MaxIter</b> — Maximum number of iterations during loss-function minimization. The iterations stop when <b>MaxIter</b> is reached or another stopping criterion is satisfied, such as <b>Tolerance</b>. Setting <b>MaxIter</b> = 0 returns the result of the startup procedure. Use <code>sys.Report.Termination.Iterations</code> to get the actual number of iterations during an estimation, where <code>sys</code> is an <code>idtf</code> model. <b>Default:</b> 20</li> <li>• <b>Advanced</b> — Search settings: <ul style="list-style-type: none"> <li>▪ <b>GnPinvConst</b> — Singular values of the Jacobian that are smaller than <math>\text{GnPinvConst} * \max(\text{size}(J)) * \text{norm}(J) * \text{eps}</math> are discarded when computing the search direction and <b>SearchMethod</b> is 'gn'. <b>GnPinvConst</b> must be a positive real value. <b>Default:</b> 10000</li> <li>▪ <b>InitGnaTol</b> — Initial value of <i>gamma</i> when <b>SearchMethod</b> is 'gna'. <b>Default:</b> .0001</li> </ul> </li> </ul>

SearchMethod	SearchOption
	<ul style="list-style-type: none"><li data-bbox="583 361 1248 453">▪ <b>LMStartValue</b> — Starting value of search-direction length <math>d</math> in the Levenberg-Marquardt method. Applicable when <b>SearchMethod</b> is 'lm'. <b>Default:</b> .001</li><li data-bbox="583 522 1267 673">▪ <b>LMStep</b> — Size of the Levenberg-Marquardt step. The next value of the search-direction length <math>d</math> in the Levenberg-Marquardt method is <b>LMStep</b> times the previous one. Applicable when <b>SearchMethod</b> is 'lm'. <b>Default:</b> 2</li><li data-bbox="583 743 1233 847">▪ <b>MaxBisections</b> — Maximum number of bisections used by the line search along the search direction. <b>Default:</b> 25</li><li data-bbox="583 873 1267 1024">▪ <b>MaxFunEvals</b> — Iterations are stopped if the number of calls to the model file exceeds this value. MaxFunEvals must be a positive integer value. <b>Default:</b> Inf</li><li data-bbox="583 1050 1267 1201">▪ <b>MinParChange</b> — Smallest parameter update allowed per iteration MinParChange must be a positive, real value. <b>Default:</b> 0</li><li data-bbox="583 1227 1267 1388">▪ <b>RelImprovement</b> — Iterations are stopped if the relative improvement of the criterion function is less than <b>RelImprovement</b>. RelImprovement must be a positive integer value. <b>Default:</b> 0</li></ul>



SearchMethod	SearchOption
	<ul style="list-style-type: none"> <li>▪ <b>StepReduction</b> — Suggested parameter update is reduced by the factor <code>StepReduction</code> after each try until either <code>MaxBisections</code> tries are completed or a lower value of the criterion function is obtained.</li> </ul> <p><code>StepReduction</code> must be a positive, real value that is greater than 1.</p> <p><b>Default:</b> 2</p>
lsqnonlin	<ul style="list-style-type: none"> <li>• <b>TolFun</b> — Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values.</li> </ul> <p>The value of <code>TolFun</code> is the same as that of <code>sys.SearchOption.Advanced.TolFun</code>.</p> <p><b>Default:</b> 1e-5</p> <ul style="list-style-type: none"> <li>• <b>TolX</b> — Termination tolerance on the estimated parameter values.</li> </ul> <p>The value of <code>TolX</code> is the same as that of <code>sys.SearchOption.Advanced.TolX</code>.</p> <p><b>Default:</b> 1e-6</p> <ul style="list-style-type: none"> <li>• <b>MaxIter</b> — Maximum number of iterations during loss-function minimization. The iterations stop when <code>MaxIter</code> is reached or another stopping criterion is satisfied, such as <code>TolFun</code> etc..</li> </ul> <p>The value of <code>MaxIter</code> is the same as that of <code>sys.SearchOption.Advanced.MaxIter</code>.</p> <p><b>Default:</b> 20</p> <ul style="list-style-type: none"> <li>• <b>Advanced</b> — Options set for <code>lsqnonlin</code>.</li> </ul> <p>For more information, see “Optimization Options”.</p>

SearchMethod	SearchOption
	Use <code>optimset('lsqnonlin')</code> to create an options set for <code>lsqnonlin</code> and then modify it to specify its various options.

## Advanced

Advanced is a structure with the following fields:

- **ErrorThreshold** — Specifies when to adjust the weight of large errors from quadratic to linear.

Errors larger than **ErrorThreshold** times the estimated standard deviation have a linear weight in the 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 the chapter “Computing the Estimate” of [1].)

**ErrorThreshold** = 0 disables robustification and leads to a purely quadratic criterion. When estimating with frequency-domain data, **ErrorThreshold** is set to zero. **ErrorThreshold** = 1.6 is a good choice for estimation when data contains outliers.

**Default:** 0

- **MaxSize** — Specifies the maximum number of elements in a segment when input-output data is split into segments.

**MaxSize** must be a positive integer.

**Default:** 250000

- **StabilityThreshold** — Specifies thresholds for stability tests.

**StabilityThreshold** is a structure with the following fields:

- **s** — Specifies the location of the right-most pole to test the stability of continuous-time models. A model is considered stable when its right-most pole is to the left of **s**.

**Default:** 0

- `z` — 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 `z` from the origin.

**Default:** `1+sqrt(eps)`

- `AutoInitThreshold` — Specifies when to automatically estimate the initial conditions.

When `InitialCondition = 'Auto'`, the initial condition is estimated when the ratio of the prediction-error norm with a zero initial condition to the norm with an estimated initial condition exceeds `AutoInitialState`.

**Default:** `1.05`

## Output Arguments

**opt**

Option set containing the specified options for `polyest`.

## Definitions

## Examples

### Create Default Options Set for Polynomial Estimation

Create a default options set for `polyest`.

```
opt = polyestOptions;
```

### Specify Options for Polynomial Estimation

Create an options set for `polyest` using the `'stability'` for Focus and set the Display to `'on'`.

```
opt = polyestOptions('Focus','stability','Display','on');
```

Alternatively, use dot notation to set the values of `opt`.

```
opt = polyestOptions;  
opt.Focus = 'stability';  
opt.Display = 'on';
```

## References

- [1] Ljung, L. *System Identification: Theory for the User*. Upper Saddle River, NJ: Prentice-Hall PTR, 1999.
- [2] Wills, Adrian, B. Ninness, and S. Gibson. "On Gradient-Based Search for Multivariable System Estimates". *Proceedings of the 16th IFAC World Congress, Prague, Czech Republic, July 3–8, 2005*. Oxford, UK: Elsevier Ltd., 2005.

**See Also** polyest

**Purpose**

Powers and products of standard regressors

**Syntax**

```
R = polyreg(model)
R = polyreg(model, 'MaxPower', n)
R = polyreg(model, 'MaxPower', n, 'CrossTerm', CrossTermVal)
```

**Description**

`R = polyreg(model)` creates an array *R* of polynomial regressors up to the power 2. If a model order has input *u* and output *y*, `na=nb=2`, and delay `nk=1`, polynomial regressors are  $y(t-1)^2$ ,  $u(t-1)^2$ ,  $y(t-2)^2$ ,  $u(t-2)^2$ . *model* is an `idnlarx` object. You must add these regressors to the *model* by assigning the `CustomRegressors model` property or by using `addreg`.

`R = polyreg(model, 'MaxPower', n)` creates an array *R* of polynomial regressors up to the power *n*. Excludes terms of power 1 and cross terms, such as  $y(t-1)*u(t-1)$ .

`R = polyreg(model, 'MaxPower', n, 'CrossTerm', CrossTermVal)` creates an array *R* of polynomial regressors up to the power *n* and includes cross terms (products of standard regressors) when *CrossTermVal* is 'on'. By default, *CrossTermVal* is 'off'.

**Examples**

Create polynomial regressors up to order 2:

```
% Estimate a nonlinear ARX model with
% na=nb=2 and nk=1.
% Nonlinearity estimator is wavenet.
load iddata1
m = nlarx(z1,[2 2 1])
% Create polynomial regressors:
R = polyreg(m);
% Estimate model:
m = nlarx(z1,[2 2 1], 'wavenet', 'CustomReg', R);
% View all model regressors (standard and custom):
getreg(m)
```

---

Create polynomial regressors up to order 3:

# polyreg

---

```
R = polyreg(m, 'MaxPower',3, 'CrossTerm', 'on')
```

If the model  $m$  that has three standard regressors  $a$ ,  $b$  and  $c$ ,  $R$  includes  $a^2$ ,  $b^2$ ,  $c^2$ ,  $a*b$ ,  $a*c$ ,  $b*c$ ,  $a^2*b$ ,  $a^2*c$ ,  $a*b^2$ ,  $a*b*c$ ,  $a*c^2$ ,  $b^2*c$ ,  $b*c^2$ ,  $a^3$ ,  $b^3$ , and  $c^3$ .

## See Also

[addreg](#) | [customreg](#) | [getreg](#) | [idnlarx](#) | [nlarx](#)

## How To

- “Identifying Nonlinear ARX Models”

**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 nth degree.

t=poly1d('Coefficients',C) creates a polynomial nonlinearity estimator object with coefficients C.

t=poly1d(n) a polynomial nonlinearity estimator object of nth degree.

Use evaluate(p,x) to compute the value of the function defined by the poly1d object p at x.

**Tips** Use poly1d to define a nonlinear function  $y = F(x)$ , where  $F$  is a single-variable polynomial function of  $x$ :

$$F(x) = c(1)x^n + c(2)x^{(n-1)} + \dots + c(n)x + c(n+1)$$

**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
```

# poly1d

---

Property Name	Description
Degree	Positive integer specifies the degree of the polynomial Default=1. For example: <code>poly1d('Degree',3)</code>
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 property 'Degree'.

## See Also

`nlhw`



**Purpose**

K-step ahead prediction

**Syntax**

```
yp = predict(sys,data,K)
yp = predict(sys,data,K,opt)
[yp,x0e,sys_pred] = predict(sys,data,K, ___ )
predict(sys,data,K___ )
```

**Description**

`yp = predict(sys,data,K)` predicts the output of an identified model, `sys`, `K` steps ahead using input-output data history from `data`.

`yp = predict(sys,data,K,opt)` predicts the output using the option set `opt` to configure prediction behavior.

`[yp,x0e,sys_pred] = predict(sys,data,K, ___ )` also returns the estimated initial state, `x0e`, and a predictor system, `sys_pred`.

`predict(sys,data,K___ )` plots the predicted output.

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.

**Difference between forecast and predict**

`predict` predicts the response over the time span of `data`. `forecast` performs prediction into the unseen future, which is a time range beyond the last instant of measured data. `predict` is a tool for validating the quality of an estimated model. Use `predict` to determine if the

# predict

---

prediction result matches the observed response in `data.OutputData`.  
If `sys` is a good prediction model, consider using it with `forecast` (only supports linear models).

## Input Arguments

### **sys**

Identified model.

`sys` may be a linear or nonlinear identified model.

### **data**

Measured input-output data.

Specify `data` as an `iddata` object.

If `sys` is a time-series model, which has no input signals, then specify `data` as an `iddata` object with no inputs, or a matrix of past (already observed) time-series data.

### **K**

Prediction horizon.

Specify `K` as a positive integer that is a multiple of the data sample-time.

**Default:** 1

### **opt**

Prediction options.

`opt` is an option set that specifies the following:

- handling of initial conditions
- data offsets

Use `predictOptions` to create the options set.

## Output Arguments

### **yp**

Predicted output.

`yp` is an `iddata` object.

Outputs up to the time `t-K` and inputs up to the time instant `t` are used to predict the output at the time instant `t`. The time variable takes values in the range represented by `data.SamplingInstants`.

When `K = Inf`, the predicted output is a pure simulation of the system.

For multi-experiment data, `yp` contains a predicted data set for each experiment. The time span of the predicted outputs matches that of the observed data.

When `sys` is specified using an `idnlhw` or `idnlgrey` model, `yp` is the same as the simulated response computed using `data.InputData` as input.

### **`x0e`**

Estimated initial states.

### **`sys_pred`**

Predictor system.

`sys_pred` is a dynamic system whose simulation, using `[data.OutputData data.InputData]` as input, yields `yp.OutputData` as the output.

For discrete-time data, `sys_pred` is always a discrete-time model.

For multi-experiment data, `sys_pred` is an array of models, with one entry for each experiment.

When `sys` is a nonlinear model, `sys_pred` is `[]`.

## **Examples**

### **Predict Time-Series Model Response**

Simulate a time-series model.

```
init_sys = idpoly([1 -0.99],[],[1 -1 0.2]);  
e = iddata([],randn(400,1));  
data = sim(init_sys,e);
```

# predict

---

`data` is an `iddata` object containing the simulated response data of a time-series model.

Estimate an ARMAX model for the simulated data.

```
na = 1;  
nb = 2;  
sys = armax(data(1:200),[na nb]);
```

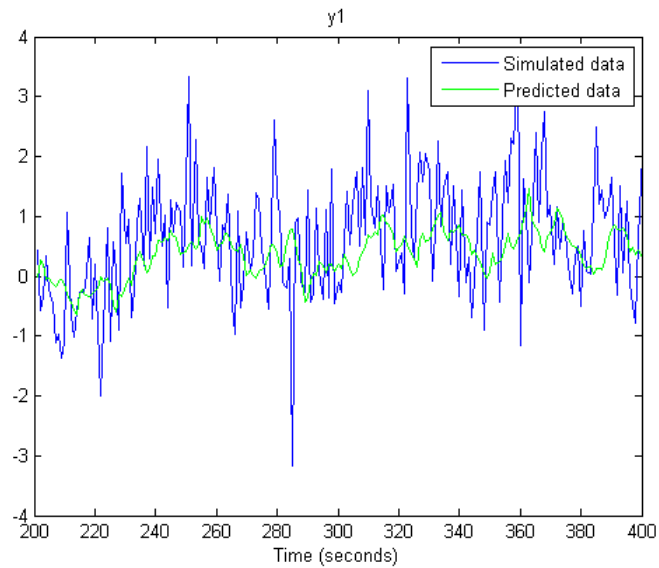
`sys` is an `idpoly` model encapsulating the identified ARMAX model for the simulated data `data`.

Obtain a 4 step-ahead prediction for the estimated model.

```
K = 4;  
yp = predict(sys,data,K);
```

Analyze the prediction.

```
plot(data(201:400),yp(201:400));  
legend('Simulated data','Predicted data');
```



Use `compare` to substitute the use of `predict` and `plot` here.

For example:

```
compare(data,sys,K);
```

### See Also

```
predictOptions | compare | pe | lsim | sim | simsd | ar |  
arx | n4sid | iddata | idpar | forecast
```

# predictOptions

---

**Purpose** Option set for predict

**Syntax**  
`opt = predictOptions`  
`opt = predictOptions(Name,Value)`

**Description** `opt = predictOptions` creates the default options set for predict.  
`opt = predictOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### InitialCondition

Specify the handling of initial conditions.

`InitialCondition` takes one of the following:

- 'z' — Zero initial conditions.
- 'e' — Estimate initial conditions such that the prediction error for observed output is minimized.
- 'd' — Similar to 'e', but absorbs nonzero delays into the model coefficients.
- `x0` — Numerical column vector denoting initial states. For multi-experiment data, use a matrix with  $N_e$  columns, where  $N_e$  is the number of experiments. Use this option for state-space and nonlinear models only.
- `io` — Structure with the following fields:
  - Input
  - Output

Use the `Input` and `Output` fields to specify the history for a time interval that starts before the start time of the data used by `predict`. In case the data used by `predict` is a time series model, specify `Input` as `[]`. Use a row vector to denote a constant signal value. The number of columns in `Input` and `Output` must always equal the number of input and output channels, respectively. For multi-experiment data, specify `io` as a struct array of  $N_e$  elements, where  $N_e$  is the number of experiments.

- `x0obj` — Specification object created using `idpar`. Use this object for discrete-time state-space models only. Use `x0obj` to impose constraints on the initial states by fixing their value or specifying minimum/maximum bounds.

For an `idnlgrey` model, `InitialCondition` can also be one of the following:

- `'fixed'` — `sys.InitialState` determines the values of the initial states, but all the states are considered fixed for estimation.
- `'model'` — `sys.InitialState` determines the values of the initial states, which states to estimate and their minimum/maximum values.

**Default:** `[]` (Initial conditions are determined based on how the model was estimated.)

## InputOffset

Input signal offset.

Specify as a column vector of length  $N_u$ , where  $N_u$  is the number of inputs.

Use `[]` to indicate no offset.

For multi-experiment data, specify `InputOffset` as a  $N_u$ -by- $N_e$  matrix. Here  $N_u$  is the number of inputs and  $N_e$  is the number of experiments.

Each entry specified by `InputOffset` will be subtracted from the corresponding input data before it is used to simulate the model.

# predictOptions

---

**Default:** []

## **OutputOffset**

Output signal offset.

Specify as a column vector of length  $N_y$ , where  $N_y$  is the number of outputs.

Use [] to indicate no offset.

For multiexperiment data, specify **OutputOffset** as a  $N_y$ -by- $N_e$  matrix. Here  $N_y$  is the number of outputs and  $N_e$  is the number of experiments.

Each entry specified by **OutputOffset** will be subtracted from the corresponding output data.

**Default:** []

## **OutputWeight**

Weight of output for initial condition estimation.

**OutputWeight** takes one of the following:

- [] — No weighting is used. This is the same as using `eye(Ny)` for the output weight, where  $N_y$  is the number of outputs.
- 'noise' — Inverse of the noise variance stored with the model.
- matrix — A positive semi-definite matrix of dimension  $N_y$ -by- $N_y$ , where  $N_y$  is the number of outputs.

**Default:** []

## **Output Arguments**

**opt**

Option set containing the specified options for `predict`.

## **Examples**

### **Create Default Options Set for Model Prediction**

Create a default options set for `predict`.



```
opt = predictOptions;
```

## Specify Options for Model Prediction

Create an options set for `predict` using zero initial conditions and set the input offset to 5.

```
opt = predictOptions('InitialCondition','z','InputOffset',5);
```

Alternatively, use dot notation to set the values of `opt`.

```
opt = predictOptions;  
opt.InitialCondition = 'z';  
opt.InputOffset = 5;
```

## See Also

`predict` | `absorbdelay` | `idpar`

# 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 linear or nonlinear identified model.

`present` thus gives more detailed information about the model than the standard `display` function.

**See Also** `getpvec` | `getcov` | `tfddata` | `idssdata` | `polydata` | `zpkdata`

**Purpose**

Estimate process model using time or frequency data

**Syntax**

```
sys = procest(data,type)
sys = procest(data,type,Name,Value)
sys = procest(data,init_sys)
sys = procest(data, __ ,opt)
```

**Description**

`sys = procest(data,type)` estimates a process model, `sys`, using time or frequency domain data, `data`. `type` defines the structure of `sys`.

`sys = procest(data,type,Name,Value)` estimates a process model with additional options specified by one or more `Name,Value` pair arguments.

`sys = procest(data,init_sys)` estimates a process model using the dynamic system `init_sys` to configure the initial parameterization.

`sys = procest(data, __ ,opt)` estimates a polynomial model using an option set, `opt`, to specify estimation behavior.

**Input Arguments****data**

Estimation data.

For time domain estimation, `data` must be an `iddata` object containing the input and output signal values.

Time series models, which are models that contain no measured inputs, cannot be estimated using `procest`. Use `ar`, `arx` or `armax` for time series models instead.

For frequency domain estimation, `data` can be one of the following:

- recorded frequency response data (`frd` or `idfrd`)
- `iddata` object with its properties specified as follows:
  - `InputData` — Fourier transform of the input signal
  - `OutputData` — Fourier transform of the output signal
  - `Domain` — 'Frequency'

data must have at least one input and one output.

## **type**

Process model structure.

**type** is an acronym that defines the structure of a process model. The acronym string is made up of:

- P — All 'Type' acronyms start with this letter.
- 0, 1, 2, or 3 — Number of time constants (poles) to be modeled. Possible integrations (poles in the origin) are not included in this number.
- I — Integration is enforced (self-regulating process).
- D — Time delay (dead time).
- Z — Extra numerator term, a zero.
- U — Under-damped modes (complex-valued poles) permitted. If U is not included in **type**, all poles are restricted to be real. The number of poles must be 2 or 3.

For information regarding how **type** affects the structure of a process model, see `idproc`.

For multiple input/output pairs use a cell array of acronyms, with one entry for each input/output pair.

## **opt**

Estimation options.

**opt** is an options set that specifies the following:

- estimation objective
- handling on initial conditions and disturbance component
- numerical search method to be used in estimation

Use `procestOptions` to create the options set.

**init\_sys**

Dynamic system that configures the initial parameterization of `sys`.

If `init_sys` is an `idproc` model, `procest` uses the parameters and constraints defined in `init_sys` as the initial guess for estimating `sys`.

If `init_sys` is not an `idproc` model, the software first converts `init_sys` to an identified process model. `procest` uses the parameters of the resulting model as the initial guess for estimation.

Use the `Structure` property of `init_sys` to configure initial guesses and constraints for  $Kp$ ,  $T_{p1}$ ,  $T_{p2}$ ,  $T_{p3}$ ,  $T_w$ ,  $Zeta$ ,  $T_d$  and  $T_z$ .

To specify an initial guess for, say, the  $T_{p1}$  parameter of `init_sys`, set `init_sys.Structure.Tp1.Value` as the initial guess.

To specify constraints for, say, the  $T_{p2}$  parameter of `init_sys`:

- set `init_sys.Structure.Tp2.Minimum` to the minimum  $T_{p2}$  value
- set `init_sys.Structure.Tp2.Maximum` to the maximum  $T_{p2}$  value
- set `init_sys.Structure.Tp2.Free` to indicate if  $T_{p2}$  is a free parameter for estimation

You can similarly specify the initial guess and constraints for the other parameters.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**InputDelay**

Input delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. Specify input delays in the time unit stored in the `TimeUnit` property.

For a system with  $N_u$  inputs, set `InputDelay` to an  $N_u$ -by-1 vector, where each entry is a numerical value representing the input delay for the corresponding input channel. You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0 for all input channels

## Output Arguments

**sys**

Identified process model.

`sys` is an `idproc` model with a structure defined by `type`.

## Examples

Estimate a process model with one pole and dead time for heat exchanger data.

```
load iddemo_heatexchanger_data
```

Collect the measured data using `iddata` and plot it.

```
data = iddata(pt,ct,Ts);  
data.InputName = '\Delta CTemp';  
data.InputUnit = 'C';  
data.OutputName = '\Delta PTemp';  
data.OutputUnit = 'C';  
data.TimeUnit = 'minutes';
```

Estimate a process model.

```
sysP1D = procest(data, 'P1D')
```

---

Detect over-parameterization of process models.

```
load iddata1 z1  
load iddata2 z2  
data = [z1, z2(1:300)];
```

`data` is a data set with 2 inputs and 2 outputs. The first input affects only the first output. Similarly, the second input affects only the second output.

In the estimated process model, the cross terms, modelling the effect of the first input on the second output and vice-versa, should be negligible. If higher orders are assigned to those dynamics, their estimations will show a high level of uncertainty.

```
type = 'P2UZ'; % denotes a model with complex-conugate pair of poles,
sys = procest(data,type) % use same orders for all I/O pairs
```

To evaluate the uncertainties, plot the frequency response.

```
w = linspace(0,20*pi,100);
h = bodeplot(sys,w);
showConfidence(h)
```

---

Estimate a process model by specifying initial guesses for parameter values and bounding them.

```
G = idfrd(idtf([10 2],[1 1.3 1.2]','iod',0.45),logspace(-2,2,256));
```

Create the estimation initializing model.

```
init_sys = idproc('P2UZD');
init_sys.Structure.Kp.Value = 1;
init_sys.Structure.Tw.Value = 2;
init_sys.Structure.Zeta.Value = 0.1;
init_sys.Structure.Td.Value = 0;
init_sys.Structure.Tz.Value = 1;
init_sys.Structure.Kp.Minimum = 0.1;
init_sys.Structure.Kp.Maximum = 10;
init_sys.Structure.Td.Maximum = 1;
init_sys.Structure.Tz.Maximum = 10;
```

Create an option set for specifying the estimation search method.

## procest

---

```
opt = procestOptions('Display','full','InitialCondition','Zero');  
opt.SearchMethod = 'lm';  
opt.SearchOption.MaxIter = 100;
```

Estimate the process model.

```
sys = procest(G,init_sys,opt);
```

Compare the data to the estimated model.

```
compare(G,sys,init_sys);
```

### See Also

[procestOptions](#) | [idproc](#) | [ssest](#) | [tfest](#) | [polyest](#) | [ar](#) |  
[arx](#) | [oe](#) | [bj](#)



**Purpose** Options set for procest

**Syntax**  
opt = procestOptions  
opt = procestOptions(Name,Value)

**Description** opt = procestOptions creates the default options set for procest.  
opt = procestOptions(Name,Value) creates an option set with the options specified by one or more Name,Value pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments, where Name is the argument name and Value is the corresponding value. Name must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

### InitialCondition

Specify how initial conditions are handled during estimation.

InitialCondition takes one of the following values:

- 'zero' — The initial condition is set to zero.
- 'estimate' — The initial condition is treated as an independent estimation parameter.
- 'backcast' — The initial condition is estimated using the best least squares fit.
- 'auto' — The software chooses the method to handle initial condition based on the estimation data.

**Default:** 'auto'

### DisturbanceModel

Specify how the handling of additive noise ( $H$ ) during estimation for the model

$$y = G(s)u + H(s)e$$

Here,  $e$  is white noise,  $u$  is the input and  $y$  is the output.

$H(s)$  is stored in the `NoiseTF` property of the numerator and denominator of `idproc` models.

`DisturbanceModel` takes one of the following strings:

- 'none' —  $H$  is fixed to one.
- 'estimate' —  $H$  is treated as an estimation parameter. The software uses the value of the `NoiseTF` property as the initial guess.
- 'ARMA1' — The software estimates  $H$  as a first-order ARMA model

$$\frac{1 + cs}{1 + ds}$$

- 'ARMA2' — The software estimates  $H$  as a second-order ARMA model

$$\frac{1 + c_1s + c_2s^2}{1 + d_1s + d_2s^2}$$

- 'fixed' — The software fixes the value of the `NoiseTF` property of the `idproc` model as the value of  $H$ .

---

**Note** Noise model cannot be estimated using frequency domain data.

---

**Default:** 'estimate'

## Focus

Defines how the errors  $e$  between the measured and the modeled outputs are weighed at specific frequencies during the minimization of the prediction error.

Higher weighting at specific frequencies emphasizes the requirement for a good fit at these frequencies.

**Focus** can take the following values:

- '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.

This method provides a stable model.

- prediction — Automatically calculates the weighting function as a product of the input spectrum and the inverse of the noise model. The weighting function minimizes the one-step-ahead prediction. This approach typically favors fitting small time intervals (higher frequency range). From a statistical-variance point of view, this weighting function is optimal. However, this method neglects the approximation aspects (bias) of the fit. Use 'stability' when you want to ensure a stable model.
- Passbands — Row vector or matrix containing frequency values that define desired passbands. For example:

```
[w1,wh]
[w11,w1h;w21,w2h;w31,w3h;...]
```

where *w1* 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.

- SISO filter — Enter any SISO linear filter in any of the following ways:
  - A single-input-single-output (SISO) linear system
  - The {A,B,C,D} format, which specifies the state-space matrices of the filter
  - The {numerator, denominator} format, which specifies the numerator and denominator of the filter transfer function

# procestOptions

---

This option calculates the weighting function as a product of the filter and the input spectrum to estimate the transfer function. To obtain a good model fit for a specific frequency range, you must choose the filter with a passband in this range. You receive an estimation result that is the same as if you had first prefiltered using `idfilt`.

- **Weighting vector** — 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.

**Default:** 'prediction'

## **EstCovar**

Controls whether parameter covariance data is generated or not.

If `EstCovar` is true, then use `getcov` to fetch the covariance matrix from the estimated model.

**Default:** true

## **Display**

Specifies whether estimation progress should be displayed.

`Display` requires one of the following strings:

- 'on' — Information on model structure and estimation results are displayed in a progress viewer window
- 'off' — No progress or results information is displayed

**Default:** 'off'

## **InputOffset**

Removes offset from time domain input data during estimation.

Specify `InputOffset` as one of the following:

- `'estimate'` — The software treats the input offsets as an estimation parameter.
- `'auto'` — The software chooses the method to handle input offsets based on the estimation data and the model structure. The estimation either assumes zero input offset or estimates the input offset.

For example, the software estimates the input offset for a model that contains an integrator.

- A column vector of length  $Nu$ , where  $Nu$  is the number of inputs.

Use `[]` to specify no offsets.

In case of multi-experiment data, specify `InputOffset` as a  $Nu$ -by- $Ne$  matrix. Here  $Nu$  is the number of inputs and  $Ne$  is the number of experiments.

Each entry specified by `InputOffset` will be subtracted from the corresponding input data.

- A parameter object, constructed using `param.Continuous`, that imposes constraints on how the software will estimate the input offset.

For example, create a parameter object for a 2-input model estimation. Specify the first input offset as fixed to zero and the second input offset as an estimation parameter.

```
opt = procestOptions;
u0 = param.Continuous('u0',[0;NaN]);
u0.Free(1) = false;
opt.Inputoffset = u0;
```

**Default:** `'auto'`

## OutputOffset

Removes offset from time domain output data during estimation.

# proceStOptions

---

Specify as a column vector of length  $N_y$ , where  $N_y$  is the number of outputs.

Use `[]` to indicate no offset.

For multiexperiment data, specify `OutputOffset` as a  $N_y$ -by- $N_e$  matrix.  $N_y$  is the number of outputs, and  $N_e$  is the number of experiments. Each entry specified by `OutputOffset` is subtracted from the corresponding output data.

**Default:** `[]`

## **OutputWeight**

Specifies criterion used during minimization.

`OutputWeight` can have the following values:

- `'noise'` — 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.
- positive semi-definite symmetric matrix ( $W$ ) — Minimize the trace of the weighted prediction error matrix  $\text{trace}(E^*EW)$ , where  $E$  is the matrix of prediction errors, with one column for each output, and  $W$  is the positive semi-definite symmetric matrix of size equal to the number of outputs. Use  $W$  to specify the relative importance of outputs in multiple-input multiple-output models, or the reliability of corresponding data.

This option is relevant only for multi-input, multi-output models.

- `[]` — The software chooses between the `'noise'` or using the identity matrix for  $W$ .

**Default:** `[]`

## **SearchMethod**

Search method used for iterative parameter estimation.

SearchMethod is a string that can take the following values:

- `gn` — The subspace Gauss-Newton direction. Singular values of the Jacobian matrix less than  $\text{GnPinvConst} \cdot \text{eps} \cdot \max(\text{size}(J)) \cdot \text{norm}(J)$  are discarded when computing the search direction.  $J$  is the Jacobian matrix. The Hessian matrix is approximated by  $J^T J$ . If there is no improvement in this direction, the function tries the gradient direction.
- `gna` — An adaptive version of subspace Gauss-Newton approach, suggested by Wills and Ninness [2]. Eigenvalues less than  $\text{gamma} \cdot \max(\text{sv})$  of the Hessian are neglected, where  $\text{sv}$  are the singular values of the Hessian. The Gauss-Newton direction is computed in the remaining subspace. *gamma* has the initial value `InitGnaTol` (see `Advanced` for more information) 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  $2 \cdot \text{LMStep}$  each time a search is successful without any bisections.
- `lm` — Uses the Levenberg-Marquardt method. This means that the next parameter value is  $-\text{pinv}(H+d \cdot I) \cdot \text{grad}$  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.
- `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.
- `grad` — The steepest descent gradient search method.
- `auto` — A choice among the above is made in the algorithm. The descent direction is calculated using `gn`, `gna`, `lm` and `grad` successively, in that order, at each iteration until a sufficient reduction in error is achieved.

**Default:** 'auto'

## SearchOption

SearchOption is an options set for the search algorithm with the fields:

SearchMethod	SearchOption
gn,gna,lm,grad and auto	<ul style="list-style-type: none"> <li>• <b>Tolerance</b> — 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 <b>Tolerance</b>, the iterations are stopped. 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.  <b>Default:</b> 0.01</li> <li>• <b>MaxIter</b> — Maximum number of iterations during loss-function minimization. The iterations stop when <b>MaxIter</b> is reached or another stopping criterion is satisfied, such as <b>Tolerance</b>.  Setting <b>MaxIter</b> = 0 returns the result of the startup procedure.  Use <code>sys.Report.Termination.Iterations</code> to get the actual number of iterations during an estimation, where <code>sys</code> is an <code>idtf</code> model.  <b>Default:</b> 20</li> <li>• <b>Advanced</b> — Search settings: <ul style="list-style-type: none"> <li>▪ <b>GnPinvConst</b> — Singular values of the Jacobian that are smaller than <code>GnPinvConst*max(size(J)*norm(J)*eps)</code> are discarded when computing the search direction and <b>SearchMethod</b> is 'gn'.  <b>GnPinvConst</b> must be a positive real value.  <b>Default:</b> 10000</li> </ul> </li> </ul>



**SearchMethodOption**

- **InitGnaTo1** — Initial value of *gamma* when SearchMethod is 'gna'.

**Default:** .0001
- **LMStartValue** — Starting value of search-direction length *d* in the Levenberg-Marquardt method. Applicable when SearchMethod is 'lm'.

**Default:** .001
- **LMStep** — 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. Applicable when SearchMethod is 'lm'.

**Default:** 2
- **MaxBisections** — Maximum number of bisections used by the line search along the search direction.

**Default:** 25
- **MaxFunEvals** — Iterations are stopped if the number of calls to the model file exceeds this value.

MaxFunEvals must be a positive integer value.

**Default:** Inf
- **MinParChange** — Smallest parameter update allowed per iteration

MinParChange must be a positive, real value.

**Default:** 0
- **RelImprovement** — Iterations are stopped if the relative improvement of the criterion function is less than RelImprovement.

SearchMethod	SearchOption
	<p>RelImprovement must be a positive integer value.</p> <p><b>Default:</b> 0</p> <ul style="list-style-type: none"> <li>StepReduction — 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.</li> </ul> <p>StepReduction must be a positive, real value that is greater than 1.</p> <p><b>Default:</b> 2</p>
lsqnonlin	<ul style="list-style-type: none"> <li>TolFun — Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values.</li> </ul> <p>The value of TolFun is the same as that of sys.SearchOption.Advanced.TolFun.</p> <p><b>Default:</b> 1e-5</p> <ul style="list-style-type: none"> <li>TolX — Termination tolerance on the estimated parameter values.</li> </ul> <p>The value of TolX is the same as that of sys.SearchOption.Advanced.TolX.</p> <p><b>Default:</b> 1e-6</p> <ul style="list-style-type: none"> <li>MaxIter — Maximum number of iterations during loss-function minimization. The iterations stop when MaxIter is reached or another stopping criterion is satisfied, such as TolFun etc..</li> </ul> <p>The value of MaxIter is the same as that of sys.SearchOption.Advanced.MaxIter.</p> <p><b>Default:</b> 20</p> <ul style="list-style-type: none"> <li>Advanced — Options set for lsqnonlin.</li> </ul>

Search Method	Search Option
---------------	---------------

For more information, see “Optimization Options”.

Use `optimset('lsqnonlin')` to create an options set for `lsqnonlin` and then modify it to specify its various options.

## Advanced

Advanced is a structure with the following fields:

- **ErrorThreshold** — Specifies when to adjust the weight of large errors from quadratic to linear.

Errors larger than `ErrorThreshold` times the estimated standard deviation have a linear weight in the 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 the chapter “Computing the Estimate” of [1].) `ErrorThreshold = 0` disables robustification and leads to a purely quadratic criterion. When estimating with frequency-domain data, `ErrorThreshold` is set to zero. `ErrorThreshold = 1.6` is a good choice for estimation when data contains outliers.

**Default:** 0

- **MaxSize** — Specifies the maximum number of elements in a segment when input-output data is split into segments.

`MaxSize` must be a positive integer.

**Default:** 250000

- **StabilityThreshold** — Specifies thresholds for stability tests.

`StabilityThreshold` is a structure with the following fields:

- **s** — Specifies the location of the right-most pole to test the stability of continuous-time models. A model is considered stable when its right-most pole is to the left of `s`.

# procestOptions

---

**Default:** 0

- **z** — 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 **z** from the origin.

**Default:**  $1 + \sqrt{\text{eps}}$

- **AutoInitThreshold** — Specifies when to automatically estimate the initial conditions.

When **InitialCondition** = 'Auto', the initial condition is estimated when the ratio of the prediction-error norm with a zero initial condition to the norm with an estimated initial condition exceeds **AutoInitialState**.

**Default:** 1.05

## Output Arguments

**opt**

Option set containing the specified options for **procest**.

## Examples

### Create Default Options Set for Process Model Estimation

Create a default options set for **procest**.

```
opt = procestOptions;
```

### Specify Options for Process Model Estimation

Create an options set for **procest** using the 'stability' for **Focus** and set the **Display** to 'on'.

```
opt = procestOptions('Focus','stability','Display','on');
```

Alternatively, use dot notation to set the values of **opt**.

```
opt = procestOptions;  
opt.Focus = 'stability';  
opt.Display = 'on';
```

## References

[1] Ljung, L. *System Identification: Theory for the User*. Upper Saddle River, NJ: Prentice-Hall PTR, 1999.

[2] Wills, Adrian, B. Ninness, and S. Gibson. “On Gradient-Based Search for Multivariable System Estimates”. *Proceedings of the 16th IFAC World Congress, Prague, Czech Republic, July 3–8, 2005*. Oxford, UK: Elsevier Ltd., 2005.

## See Also

procest | idproc | idfilt

<b>Purpose</b>	Class representing piecewise-linear nonlinear estimator for Hammerstein-Wiener models
<b>Syntax</b>	<pre>t=pwlinear('NumberOfUnits',N) t=pwlinear('BreakPoints',BP) t=pwlinear(Property1,Value1,...PropertyN,ValueN)</pre>
<b>Description</b>	<p><code>pwlinear</code> is an object that stores the piecewise-linear nonlinear estimator for estimating Hammerstein-Wiener models.</p> <p>You can use the constructor to create the nonlinearity object, as follows:</p> <p><code>t=pwlinear('NumberOfUnits',N)</code> creates a piecewise-linear nonlinearity estimator object with <math>N</math> breakpoints.</p> <p><code>t=pwlinear('BreakPoints',BP)</code> creates a piecewise-linear nonlinearity estimator object with breakpoints at values <math>BP</math>.</p> <p><code>t=pwlinear(Property1,Value1,...PropertyN,ValueN)</code> creates a piecewise-linear nonlinearity estimator object specified by properties in “<code>pwlinear Properties</code>” on page 2-741.</p> <p>Use <code>evaluate(p,x)</code> to compute the value of the function defined by the <code>pwlinear</code> object <code>p</code> at <math>x</math>.</p>
<b>Tips</b>	<p>Use <code>pwlinear</code> to define a nonlinear function <math>y = F(x)</math>, where <math>F</math> is a piecewise-linear (affine) function of <math>x</math> and there are <math>n</math> breakpoints <math>(x_k, y_k)</math>, <math>k = 1, \dots, n</math>. <math>y_k = F(x_k)</math>. <math>F</math> is linearly interpolated between the breakpoints. <math>y</math> and <math>x</math> are scalars.</p> <p><math>F</math> is also linear to the left and right of the extreme breakpoints. The slope of these extension is a function of <math>x_i</math> and <math>y_i</math> breakpoints. The breakpoints are ordered by ascending <math>x</math>-values, which is important when you set a specific breakpoint to a different value.</p> <p>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.</p>

## **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:  <code>pwlinear('NumberOfUnits',5)</code>
BreakPoints	2-by-n matrix containing the breakpoint x and y value, specified using the following format:  $[x_1, x_2, \dots, x_n; y_1, y_2, \dots, y_n]$ .  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** Pole-zero plot of dynamic system

**Syntax**  
`pzmap(sys)`  
`pzmap(sys1,sys2,...,sysN)`  
`[p,z] = pzmap(sys)`

**Description** `pzmap(sys)` creates a pole-zero plot of the continuous- or discrete-time dynamic system model `sys`. For SISO systems, `pzmap` plots the transfer function poles and zeros. For MIMO systems, it plots the system poles and transmission zeros. The poles are plotted as x's and the zeros are plotted as o's.

`pzmap(sys1,sys2,...,sysN)` creates the pole-zero plot of multiple models on a single figure. The models can have different numbers of inputs and outputs and can be a mix of continuous and discrete systems.

`[p,z] = pzmap(sys)` returns the system poles and (transmission) zeros in the column vectors `p` and `z`. No plot is drawn on the screen.

You can use the functions `sgrid` or `zgrid` to plot lines of constant damping ratio and natural frequency in the *s*- or *z*-plane.

**Tips** You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

## Examples **Example 1**

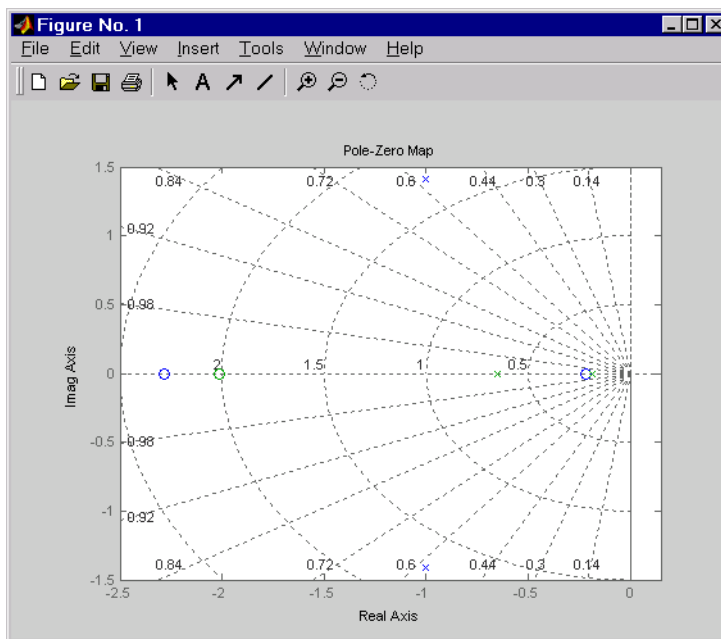
### Pole-Zero Plot of Dynamic System

Plot the poles and zeros of the continuous-time system

$$H(s) = \frac{2s^2 + 5s + 1}{s^2 + 2s + 3}$$

```
H = tf([2 5 1],[1 2 3]); sgrid
pzmap(H)
grid on
```





## Example 2

Plot the pzmap for a 2-input-output discrete-time IDSS model.

```
A = [0.1 0; 0.2 0.9]; B = [.1 .2; 0.1 .02]; C = [10 20; 2 -5]; D = [1 0; 0 1];
sys = idss(A,B,C,D, 'Ts', 0.1);
```

## Algorithms

pzmap uses a combination of pole and zero.

## See Also

damp | esort | dsort | pole | rlocus | sgrid | zgrid | zero | iopzmap

# pzoptions

---

**Purpose** Create list of pole/zero plot options

**Syntax**  
P = pzoptions  
P = pzoption('cstprefs')

**Description** P = pzoptions returns a list of available options for pole/zero plots (pole/zero, input-output pole/zero and root locus) with default values set.. You can use these options to customize the pole/zero plot appearance from the command line.

P = pzoption('cstprefs') initializes the plot options with the options you selected in the Control System Toolbox Preferences Editor. For more information about the editor, see “Toolbox Preferences Editor” in the User’s Guide documentation.

This table summarizes the available pole/zero plot options.

Option	Description
Title, XLabel, YLabel	Label text and style
TickLabel	Tick label style
Grid	Show or hide the grid Specified as one of the following strings: 'off'   'on' <b>Default:</b> 'off'
XlimMode, YlimMode	Limit modes
Xlim, Ylim	Axes limits
IOWGrouping	Grouping of input-output pairs Specified as one of the following strings: 'none'   'inputs'   'output'   'all' <b>Default:</b> 'none'
InputLabel, OutputLabel	Input and output label styles

<b>Option</b>	<b>Description</b>
InputVisible, OutputVisible	Visibility of input and output channels
FreqUnits	<p>Frequency units, specified as one of the following strings:</p> <ul style="list-style-type: none"> <li>• 'Hz'</li> <li>• 'rad/second'</li> <li>• 'rpm'</li> <li>• 'kHz'</li> <li>• 'MHz'</li> <li>• 'GHz'</li> <li>• 'rad/nanosecond'</li> <li>• 'rad/microsecond'</li> <li>• 'rad/millisecond'</li> <li>• 'rad/minute'</li> <li>• 'rad/hour'</li> <li>• 'rad/day'</li> <li>• 'rad/week'</li> <li>• 'rad/month'</li> <li>• 'rad/year'</li> <li>• 'cycles/nanosecond'</li> <li>• 'cycles/microsecond'</li> <li>• 'cycles/millisecond'</li> <li>• 'cycles/hour'</li> <li>• 'cycles/day'</li> </ul>

# pzoptions

---

Option	Description
	<ul style="list-style-type: none"><li>• 'cycles/week'</li><li>• 'cycles/month'</li><li>• 'cycles/year'</li></ul> <p><b>Default:</b> 'rad/s'</p> <p>You can also specify 'auto' which uses frequency units rad/TimeUnit relative to system time units specified in the TimeUnit property. For multiple systems with different time units, the units of the first system are used.</p>
TimeUnits	<p>Time units, specified as one of the following strings:</p> <ul style="list-style-type: none"><li>• 'nanoseconds'</li><li>• 'microseconds'</li><li>• 'milliseconds'</li><li>• 'seconds'</li><li>• 'minutes'</li><li>• 'hours'</li><li>• 'days'</li><li>• 'weeks'</li><li>• 'months'</li><li>• 'years'</li></ul> <p><b>Default:</b> 'seconds'</p>

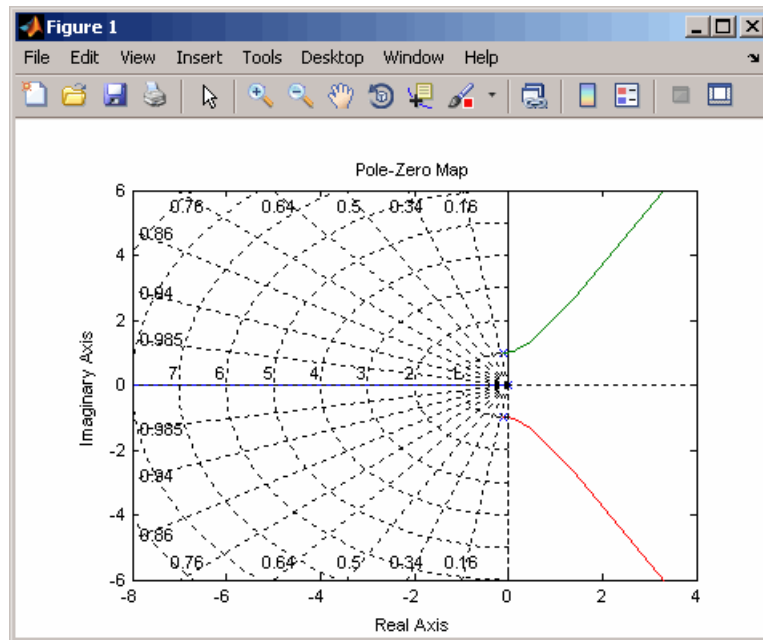
Option	Description
	You can also specify 'auto' which uses time units specified in the TimeUnit property of the input system. For multiple systems with different time units, the units of the first system is used.
ConfidenceRegionNumberSD	Number of standard deviations to use when displaying the confidence region characteristic for identified models (valid only iopzplot).

## Examples

In this example, you enable the grid option before creating a plot.

```
P = pzoptions; % Create set of plot options P
P.Grid = 'on'; % Set the grid to on in options
h = rlocusplot(tf(1,[1,.2,1,0]),P);
```

The following root locus plot is created with the grid enabled.



**See Also**     `getoptions` | `iopzplot` | `pzplot` | `setoptions`

**Purpose** Pole-zero map of dynamic system model with plot customization options

**Syntax**

```
h = pzplot(sys)
pzplot(sys1,sys2,...)
pzplot(AX,...)
pzplot(..., plotoptions)
```

**Description** `h = pzplot(sys)` computes the poles and (transmission) zeros of the dynamic system model `sys` and plots them in the complex plane. The poles are plotted as x's and the zeros are plotted as o's. It also returns the plot handle `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands. Type

```
help pzoptions
```

for a list of available plot options.

`pzplot(sys1,sys2,...)` shows the poles and zeros of multiple models `sys1,sys2,...` on a single plot. You can specify distinctive colors for each model, as in

```
pzplot(sys1, 'r', sys2, 'y', sys3, 'g')
```

`pzplot(AX,...)` plots into the axes with handle `AX`.

`pzplot(..., plotoptions)` plots the poles and zeros with the options specified in `plotoptions`. Type

```
help pzoptions
```

for more detail.

The function `sgrid` or `zgrid` can be used to plot lines of constant damping ratio and natural frequency in the *s*- or *z*-plane.

For arrays `sys` of dynamic system models, `pzmap` plots the poles and zeros of each model in the array on the same diagram.

## Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

## Examples

Use the plot handle to change the color of the plot’s title.

```
sys = rss(3,2,2);  
h = pzplot(sys);  
p = getoptions(h); % Get options for plot.  
p.Title.Color = [1,0,0]; % Change title color in options.  
setoptions(h,p); % Apply options to plot.
```

## See Also

[getoptions](#) | [pzmap](#) | [setoptions](#) | [iopzplot](#)



**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,

$$na: A(q) = 1 + a_1q^{-1} + \dots + a_{na}q^{-na}$$

$$nb: B(q) = b_1 + b_2q^{-1} + \dots + b_{nb}q^{-nb+1}$$

$$nc: C(q) = 1 + c_1q^{-1} + \dots + c_{nc}q^{-nc}$$

See “What Are 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  $n_k = 0$ , shift the input sequence appropriately and use  $n_k = 1$ .

## Algorithms

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 parameters of ARX or AR models recursively

**Syntax**

```
thm = rarx(z,nn,adm,adg)
[thm,yhat,P,phi] = rarx(z,nn,adm,adg,th0,P0,phi0)
```

**Description**

`thm = rarx(z,nn,adm,adg)` estimates the parameters `thm` of single-output ARX model from input-output data `z` and model orders `nn` using the algorithm specified by `adm` and `adg`. If `z` is a time series `y` and `nn = na`, `rarx` estimates the parameters of a single-output AR model.

`[thm,yhat,P,phi] = rarx(z,nn,adm,adg,th0,P0,phi0)` estimates the parameters `thm`, the predicted output `yhat`, final values of the scaled covariance matrix of the parameters `P`, and final values of the data vector `phi` of single-output ARX model from input-output data `z` and model orders `nn` using the algorithm specified by `adm` and `adg`. If `z` is a time series `y` and `nn = na`, `rarx` estimates the parameters of a single-output AR model.

**Definitions**

The general ARX model structure is:

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

The orders of the ARX model are:

$$na: A(q) = 1 + a_1q^{-1} + \dots + a_naq^{-na}$$

$$nb: B(q) = b_1 + b_2q^{-1} + \dots + b_nbq^{-nb+1}$$

Models with several inputs are defined, as follows:

$$A(q)y(t) = B_1(q)u_1(t-nk_1) + \dots + B_{nu}(q)u_{nu}(t-nk_{nu}) + e(t)$$

**Input Arguments**

`z`

Name of the matrix `iddata` object that represents the input-output data or a matrix `z = [y u]`, where `y` and `u` are column vectors.

For multiple-input models, the `u` matrix contains each input as a column vector:

$u = [u_1 \dots u_n]$

**nn**

For input-output models, specifies the structure of the ARX model as:

$nn = [n_a \ n_b \ n_k]$

where  $n_a$  and  $n_b$  are the orders of the ARX model, and  $n_k$  is the delay.

For multiple-input models,  $n_b$  and  $n_k$  are row vectors that define orders and delays for each input.

For time-series models,  $nn = n_a$ , where  $n_a$  is the order of the AR model.

---

**Note** The delay  $n_k$  must be larger than 0. If you want  $n_k = 0$ , shift the input sequence appropriately and use  $n_k = 1$  (see `nkshift`).

---

**adm and adg**

$adm = 'ff'$  and  $adg = lam$  specify the *forgetting factor* algorithm with the forgetting factor  $\lambda = lam$ . This algorithm is also known as recursive least squares (RLS). In this case, the matrix  $P$  has the following interpretation:  $R_y/2 * P$  is approximately equal to the covariance matrix of the estimated parameters.  $R_y$  is the variance of the innovations (the true prediction errors  $e(t)$ ).

$adm = 'ug'$  and  $adg = gam$  specify the *unnormalized gradient* algorithm with gain  $gamma = gam$ . This algorithm is also known as the normalized least mean squares (LMS).

`adm = 'ng'` and `adg = gam` specify the *normalized gradient* or normalized least mean squares (NLMS) algorithm. In these cases, `P` is not applicable.

`adm = 'kf'` and `adg = R1` specify the *Kalman filter based* algorithm 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.

`th0`

Initial value of the parameters in a row vector, consistent with the rows of `thm`.

Default: All zeros.

`P0`

Initial values of the scaled covariance matrix of the parameters.

Default:  $10^4$  times the identity matrix.

`phi0`

The argument `phi0` contains the initial values of the data vector:

$$\varphi(t) = [y(t-1), \dots, y(t-na), u(t-1), \dots, u(t-nb-nk+1)]$$

If  $z = [y(1), u(1); \dots; y(N), u(N)]$ ,  $\text{phi0} = \varphi(1)$  and  $\text{phi} = \varphi(N)$ . For online use of `rarx`, use `phi0`, `th0`, and `P0` as the previous outputs `phi`, `thm` (last row), and `P`.

Default: All zeros.

## Output Arguments

`thm`

Estimated parameters of the model. The  $k$ th row of `thm` contains the parameters associated with time  $k$ ; that is, the estimate parameters are based on the data in 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]
```

For a multiple-input model, the  $b$  are grouped by input. For example, the  $b$  parameters associated with the first input are listed first, and the  $b$  parameters associated with the second input are listed next.

**yhat**

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.

**P**

Final values of the scaled covariance matrix of the parameters.

**phi**

**phi** contains the final values of the data vector:

$$\phi(t) = [y(t-1), \dots, y(t-na), u(t-1), \dots, u(t-nb-nk+1)]$$

## Examples

Adaptive noise canceling: The signal  $y$  contains a component that originates from a known signal  $r$ . Remove this component by recursively estimating the system that relates  $r$  to  $y$  using a sixth-order FIR model and 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 this is an online application, you can plot the best estimate of the signal  $y - \text{noise}$  at the same time as the data  $y$  and  $u$  become available, use the following code:

```
phi = zeros(6,1);
P=1000*eye(6);
th = zeros(1,6);
```

```
axis([0 100 -2 2]);
plot(0,0,'*'), hold on
% Use a while 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` is a function that reads the value of an A/D converter at the indicated time instant.

**See Also**

`nkshift` | `rarmax` | `rbj` | `roe` | `rpem` | `rp1r`

**How To**

- “Algorithms for Recursive Estimation”

**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,

$$nb: B(q) = b_1 + b_2q^{-1} + \dots + b_{nb}q^{-nb+1}$$

$$nc: C(q) = 1 + c_1q^{-1} + \dots + c_{nc}q^{-nc}$$

$$nd: D(q) = 1 + d_1q^{-1} + \dots + d_{nd}q^{-nd}$$

$$nf: F(q) = 1 + f_1q^{-1} + \dots + f_{nf}q^{-nf}$$

See “What Are 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]`



$\hat{y}$  is the predicted value of the output, according to the current model; that is, row  $k$  of  $\hat{y}$  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`.

## Algorithms

The general recursive prediction error algorithm (11.44) of Ljung (1999) is implemented. See also “Algorithms for Recursive Estimation”.

## See Also

`nkshift` | `rarmax` | `rarx` | `roe` | `rpem` | `rp1r`

# realdata

---

<b>Purpose</b>	Determine whether iddata is based on real-valued signals
<b>Syntax</b>	<code>realdata(data)</code>
<b>Description</b>	<p><code>realdata</code> returns 1 if</p> <ul style="list-style-type: none"><li>• <code>data</code> contains only real-valued signals.</li><li>• <code>data</code> contains frequency-domain signals, obtained by Fourier transformation of real-valued signals.</li></ul> <p>Otherwise <code>realdata</code> returns 0.</p> <p>Notice the difference with <code>isreal</code>:</p> <pre>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</pre>

<b>Purpose</b>	Replicate and tile models
<b>Syntax</b>	<pre>rsys = repsys(sys,[M N]) rsys = repsys(sys,N) rsys = repsys(sys,[M N S1,...,Sk])</pre>
<b>Description</b>	<p><code>rsys = repsys(sys,[M N])</code> replicates the model <code>sys</code> into an M-by-N tiling pattern. The resulting model <code>rsys</code> has <code>size(sys,1)*M</code> outputs and <code>size(sys,2)*N</code> inputs.</p> <p><code>rsys = repsys(sys,N)</code> creates an N-by-N tiling.</p> <p><code>rsys = repsys(sys,[M N S1,...,Sk])</code> replicates and tiles <code>sys</code> along both I/O and array dimensions to produce a model array. The indices <code>S</code> specify the array dimensions. The size of the array is <code>[size(sys,1)*M, size(sys,2)*N, size(sys,3)*S1, ...]</code>.</p>
<b>Tips</b>	<p><code>rsys = repsys(sys,N)</code> produces the same result as <code>rsys = repsys(sys,[N N])</code>. To produce a diagonal tiling, use <code>rsys = sys*eye(N)</code>.</p>
<b>Input Arguments</b>	<p><b>sys</b> Model to replicate.</p> <p><b>M</b> Number of replications of <code>sys</code> along the output dimension.</p> <p><b>N</b> Number of replications of <code>sys</code> along the input dimension.</p> <p><b>S</b> Numbers of replications of <code>sys</code> along array dimensions.</p>
<b>Output Arguments</b>	<p><b>rsys</b> Model having <code>size(sys,1)*M</code> outputs and <code>size(sys,2)*N</code> inputs.</p>

If you provide array dimensions  $S_1, \dots, S_k$ , `rsys` is an array of dynamic systems which each have `size(sys,1)*M` outputs and `size(sys,2)*N` inputs. The size of `rsys` is `[size(sys,1)*M, size(sys,2)*N, size(sys,3)*S1, ...]`.

## Examples

Replicate a SISO transfer function to create a MIMO transfer function that has three inputs and two outputs.

```
sys = tf(2,[1 3]);  
rsys = repsys(sys,[2 3]);
```

The preceding commands produce the same result as:

```
sys = tf(2,[1 3]);  
rsys = [sys sys sys; sys sys sys];
```

---

Replicate a SISO transfer function into a 3-by-4 array of two-input, one-output transfer functions.

```
sys = tf(2,[1 3]);  
rsys = repsys(sys, [1 2 3 4]);
```

To check the size of `rsys`, enter:

```
size(rsys)
```

This command produces the result:

```
3x4 array of transfer functions.  
Each model has 1 outputs and 2 inputs.
```

## See Also

`append`

<b>Purpose</b>	Resample time-domain data by decimation or interpolation (requires Signal Processing Toolbox software)
<b>Syntax</b>	<code>resample(data,P,Q)</code> <code>resample(data,P,Q,order)</code>
<b>Description</b>	<p><code>resample(data,P,Q)</code> resamples data such that the data is interpolated by a factor <math>P</math> and then decimated by a factor <math>Q</math>. <code>resample(z,1,Q)</code> results in decimation by a factor <math>Q</math>.</p> <p><code>resample(data,P,Q,order)</code> filters the data by applying a filter of specified order before interpolation and decimation.</p>
<b>Input Arguments</b>	<p><code>data</code> Name of time-domain <code>iddata</code> object. Can be input-output or time-series data.</p> <p>Data must be sampled at equal time intervals.</p> <p><math>P, Q</math> Integers that specify the resampling factor, such that the new sampling interval is <math>Q/P</math> times the original one.</p> <p><math>(Q/P) &gt; 1</math> results in decimation and <math>(Q/P) &lt; 1</math> results in interpolation.</p> <p><code>order</code> Order of the filters applied before interpolation and decimation.</p> <p>Default: 10</p>
<b>Algorithms</b>	If you have installed the Signal Processing Toolbox software, <code>resample</code> calls the Signal Processing Toolbox <code>resample</code> function. The algorithm takes into account the intersample characteristics of the input signal, as described by <code>data.InterSample</code> .
<b>Examples</b>	In this example, you increase the sampling rate by a factor of 1.5 and compare the resampled and the original data signals.

# resample

---

```
plot(u)
ur = resample(u,3,2);
plot(u,ur)
```

## See Also

idresamp

**Purpose** Change shape of model array

**Syntax**

```
sys = reshape(sys,s1,s2,...,sk)
sys = reshape(sys,[s1 s2 ... sk])
```

**Description** `sys = reshape(sys,s1,s2,...,sk)` (or, equivalently, `sys = reshape(sys,[s1 s2 ... sk])`) reshapes the LTI array `sys` into an `s1-by-s2-by-...-by-sk` model array. With either syntax, there must be `s1*s2*...*sk` models in `sys` to begin with.

**Examples** Change the shape of a model array from 2x3 to 6x1.

```
% Create a 2x3 model array.
sys = rss(4,1,1,2,3);
% Confirm the size of the array.
size(sys)
```

This input produces the following output:

```
2x3 array of state-space models
Each model has 1 output, 1 input, and 4 states.
```

Change the shape of the array.

```
sys1 = reshape(sys,6,1);
size(sys1)
```

This input produces the following output:

```
6x1 array of state-space models
Each model has 1 output, 1 input, and 4 states.
```

**See Also** `ndims` | `size`

# resid

---

**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 linear or nonlinear identified model.

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 one 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 from the 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`

# retrend

---

**Purpose** Add offsets or trends to data signals

**Syntax** `data = retrend(data_d,T)`

**Description** `data = retrend(data_d,T)` returns a data object `data` by adding the trend information `T` to each signal in `data_d`. `data_d` is a time-domain `iddata` object. `T` is an `TrendInfo` object.

**Examples** Subtract means from input-output signals, estimate a linear model, and retrend the simulated output:

```
% Load SISO data containing vectors u2 and y2
load dryer2
% Create data object with sampling time of 0.08 sec
data=iddata(y2,u2,0.08)
% Remove the mean from the data
[data_d,T] = detrend(data,0)
% Estimate a linear ARX model
m = arx(data_d,[2 2 1])
% Simulate the model output
% with zero initial states
y_sim = sim(m,data_d(:,[],:));
% Retrend the simulated model output
y_tot = retrend(y_sim,T);
```

**See Also** `getTrend` | `detrend` | `TrendInfo`

**How To** • “Handling Offsets and Trends in Data”

**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,

$$nb: B(q) = b_1 + b_2q^{-1} + \dots + b_{nb}q^{-nb+1}$$

$$nf: F(q) = 1 + f_1q^{-1} + \dots + f_{nf}q^{-nf}$$

See “What Are 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.

$$thm(k,:) = [b1, \dots, bnb, f1, \dots, fnf]$$

$\hat{y}$  is the predicted value of the output, according to the current model; that is, row  $k$  of  $\hat{y}$  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  $n_k$  be larger than 0. If you want  $n_k = 0$ , shift the input sequence appropriately and use  $n_k = 1$ .

## **Algorithms**

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

$$A(q)y(t) = \frac{B_1(q)}{F_1(q)}u_1(t - nk_1) + \dots + \frac{B_{nu}(q)}{F_{nu}(q)}u_{nu}(t - nk_{nu}) + \frac{C(q)}{D(q)}e(t)$$

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 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`.

$\hat{y}$  is the predicted value of the output, according to the current model; that is, row  $k$  of  $\hat{y}$  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`.

## Algorithms

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`

<b>Purpose</b>	Estimate general input-output models using recursive pseudolinear regression method
<b>Syntax</b>	<pre>thm = rplr(z,nn,adm,adg) [thm,yhat,P,phi] = rplr(z,nn,adm,adg,th0,P0,phi0)</pre>
<b>Description</b>	<p>This is a direct alternative to <code>rpem</code> and has essentially the same syntax. See <code>rpem</code> for an explanation of the input and output arguments.</p> <p><code>rplr</code> differs from <code>rpem</code> in that it uses another gradient approximation. See Section 11.5 in Ljung (1999). Several of the special cases are well-known algorithms.</p> <p>When applied to ARMAX models, (<math>nn = [na \ nb \ nc \ 0 \ 0 \ nk]</math>), <code>rplr</code> gives the extended least squares (ELS) method. When applied to the output-error structure (<math>nn = [0 \ nb \ 0 \ 0 \ nf \ nk]</math>), the method is known as HARF in the <code>adm = 'ff'</code> case and SHARF in the <code>adm = 'ng'</code> case.</p> <p><code>rplr</code> can also be applied to the time-series case in which an ARMA model is estimated with</p> <pre>z = y nn = [na nc]</pre> <p>You can thus use <code>rplr</code> as an alternative to <code>rarmax</code> for this case.</p>
<b>References</b>	For more information about HARF and SHARF, see Chapter 11 in Ljung (1999).
<b>See Also</b>	<code>nkshift</code>   <code>rarmax</code>   <code>rarx</code>   <code>rbj</code>   <code>roe</code>   <code>rpem</code>

# rsample

---

**Purpose** Random sampling of linear identified systems

**Syntax**  
`sys_array = rsample(sys,N)`  
`sys_array = rsample(sys,N,sd)`

**Description** `sys_array = rsample(sys,N)` creates N random samples of the identified linear system, `sys`. `sys_array` contains systems with the same structure as `sys`, whose parameters are perturbed about their nominal values, based on the parameter covariance.

`sys_array = rsample(sys,N,sd)` specifies the standard deviation level, `sd`, for perturbing the parameters of `sys`.

**Tips**

- For systems with large parameter uncertainties, the randomized systems may contain unstable elements. These unstable elements may make it difficult to analyze the properties of the identified system. Execution of analysis commands, such as `step`, `bode`, `sim`, etc., on such systems can produce unreliable results. Instead, use a dedicated Monte-Carlo analysis command such as `simsd`.

**Input Arguments**

**sys**  
Identifiable system.

**N**  
Number of samples to be generated.

**Default:** 10

**sd**  
Standard deviation level for perturbing the identifiable parameters of `sys`.

**Default:** 1



## Output Arguments

### **sys\_array**

Array of random samples of `sys`.

If `sys` is an array of models, then the size of `sys_array` is equal to `[size(sys) N]`. There are `N` randomized samples for each model in `sys`.

The parameters of the samples in `sys_array` vary from the original identifiable model within 1 standard deviation of their nominal values.

## Examples

### **Random sample of an Identified System**

Estimate a 3rd order discrete-time state-space model and analyze the uncertainty in its time (step) and frequency (bode) responses.

```
load iddata2 z2;
sys = n4sid(z2,3);
N = 20;
sys_array = rsample(sys,N);
opt = bodeoptions; opt.PhaseMatching = 'on';
figure, bodeplot(sys_array,'g',sys,'r.',opt)
figure, stepplot(sys_array,'g',sys,'r.-')
```

### **Specify Standard Deviation Level for Parameter Perturbation**

```
load iddata2 z2;
sys = n4sid(z2,3);
N = 20;
sd = 2;
sys_array = rsample(sys,N,sd);
figure, bode(sys_array);
```

Estimate an ARMAX model and compare the frequency response confidence region corresponding to 2-std deviations (asymptotic estimate) to values obtained by random sampling for the same value of standard deviation.

```
load iddata1 z1
sys = armax(z1,[2 2 2 1]);
rsys = rsample(sys,20,2);
```

# rsample

---

```
opt = bodeoptions; opt.PhaseMatching = 'on';  
opt.ConfidenceRegionNumberSD = 2;  
bodeplot(rsys, 'g', sys, 'r', opt)
```

To view the confidence region, right click on the plot and choose **Characteristics > Confidence Region**.

## See Also

[simstd](#) | [init](#) | [noisecnv](#) | [noise2meas](#) | [iopzmap](#) | [bode](#) | [step](#)

<b>Purpose</b>	Class representing saturation nonlinearity estimator for Hammerstein-Wiener models
<b>Syntax</b>	<code>s=saturation(LinearInterval,L)</code>
<b>Description</b>	<p>saturation is an object that stores the saturation nonlinearity estimator for estimating Hammerstein-Wiener models.</p> <p>You can use the constructor to create the nonlinearity object, as follows:</p> <p><code>s=saturation(LinearInterval,L)</code> creates a saturation nonlinearity estimator object, initialized with the linear interval L.</p> <p>Use <code>evaluate(s,x)</code> to compute the value of the function defined by the saturation object s at x.</p>
<b>Tips</b>	<p>Use saturation to define a nonlinear function <math>y = F(x)</math>, where <math>F</math> is a function of <math>x</math> and has the following characteristics:</p> $\begin{array}{ll} a \leq x < b & F(x) = x \\ a > x & F(x) = a \\ b \leq x & F(x) = b \end{array}$ <p><math>y</math> and <math>x</math> are scalars.</p>
<b>saturation Properties</b>	<p>You can specify the property value as an argument in the constructor to specify the object.</p> <p>After creating the object, you can use <code>get</code> or dot notation to access the object property values. For example:</p> <pre>% List LinearInterval property value get(s) s.LinearInterval</pre> <p>You can also use the <code>set</code> function to set the value of particular properties. For example:</p>

# saturation

---

```
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=n1hw(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 `n1hw`.

## See Also

`n1hw`

**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 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

$$nn = na$$

## segment

---

The output argument `segm` is a matrix, where the  $k$ th 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 exhibits 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.

`l1` is the guaranteed life of each of the models. That is, any created candidate model is not abolished until after at least `l1` time steps. The default is `l1 = 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.

## Algorithms

The algorithm is based on  $M$  parallel models, each recursively estimated by an algorithm of Kalman filter type. Each model 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 11 samples) that has the lowest posterior probability is abolished. A new model is started, assuming that the system parameters have changed, 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.

# selstruc

---

**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 exported to 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

$$\begin{aligned} V_{\text{mod}} &= \log\left(V\left(1 + \frac{2d}{N}\right)\right) \\ &= \log(V) + \frac{2d}{N}, N \gg d \end{aligned}$$

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.  $\log(V) + \frac{2d}{N}$  is the Akaike's Information Criterion (AIC). See `aic` for more details.

`c = 'mdl'` returns in `nn` the structure that minimizes Rissanen's Minimum Description Length (MDL) criterion.



$$V_{\text{mod}} = V \left( 1 + \frac{d \log(N)}{N} \right)$$

When  $c$  equals a numerical value, the structure that minimizes

$$V_{\text{mod}} = V \left( 1 + \frac{cd}{N} \right)$$

is selected.

The output argument `vmod` has the same format as `v`, but it contains the logarithms of the accordingly modified criteria.

## Examples

```
load iddata5;
data = z5;
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 or modify model properties

**Syntax**

```
set
set(sys, 'Property', Value)
set(sys, 'Property1', Value1, 'Property2', Value2, ...)
set(sys, 'Property')
set(sys)
```

**Description** `set` is used to set or modify the properties of a dynamic system model. Like its Handle Graphics<sup>®</sup> counterpart, `set` uses property name/property value pairs to update property values.

`set(sys, 'Property', Value)` assigns the value `Value` to the property of the model `sys` specified by the string `'Property'`. This string can be the full property name (for example, `'UserData'`) or any unambiguous case-insensitive abbreviation (for example, `'user'`). The specified property must be compatible with the model type. For example, if `sys` is a transfer function, `Variable` is a valid property but `StateName` is not. For a complete list of available system properties for any linear model type, see the reference page for that model type.

`set(sys, 'Property1', Value1, 'Property2', Value2, ...)` sets multiple property values with a single statement. Each property name/property value pair updates one particular property.

`set(sys, 'Property')` displays admissible values for the property specified by `'Property'`.

`set(sys)` displays all assignable properties of `sys` and their admissible values.

**Examples** Consider the SISO state-space model created by

```
sys = ss(1,2,3,4);
```

You can add an input delay of 0.1 second, label the input as torque, reset the  $D$  matrix to zero, and store its DC gain in the `'Userdata'` property by

```
set(sys,'inputd',0.1,'inputn','torque','d',0,'user',dcgain(sys))
```

Note that `set` does not require any output argument. Check the result with `get` by typing

```
get(sys)
    a: 1
    b: 2
    c: 3
    d: 0
    e: []
    StateName: {''}
    InternalDelay: [0x1 double]
    Ts: 0
    InputDelay: 0.1
    OutputDelay: 0
    InputName: {'torque'}
    OutputName: {''}
    InputGroup: [1x1 struct]
    OutputGroup: [1x1 struct]
    Name: ''
    Notes: {}
    UserData: -2
```

## Tips

For discrete-time transfer functions, the convention used to represent the numerator and denominator depends on the choice of variable (see `tf` for details). Like `tf`, the syntax for `set` changes to remain consistent with the choice of variable. For example, if the `Variable` property is set to `'z'` (the default),

```
set(h,'num',[1 2],'den',[1 3 4])
```

produces the transfer function

$$h(z) = \frac{z+2}{z^2+3z+4}$$

However, if you change the Variable to 'z^-1' by

```
set(h,'Variable','z^-1'),
```

the same command

```
set(h,'num',[1 2],'den',[1 3 4])
```

now interprets the row vectors [1 2] and [1 3 4] as the polynomials  $1 + 2z^{-1}$  and  $1 + 3z^{-1} + 4z^{-2}$  and produces:

$$\bar{h}(z^{-1}) = \frac{1 + 2z^{-1}}{1 + 3z^{-1} + 4z^{-2}} = zh(z)$$

---

**Note** Because the resulting transfer functions are different, make sure to use the convention consistent with your choice of variable.

---

## See Also

get | frd | ss | tf | zpk | idfrd | idss | idtf | idgrey | idproc |  
idpoly | idnlrx | idnlhw | idnlgrey

---

<b>Purpose</b>	Set parameter covariance data in identified model
<b>Syntax</b>	<code>sys = setcov(sys0,cov)</code>
<b>Description</b>	<p><code>sys = setcov(sys0,cov)</code> modifies the parameter covariance of <code>sys0</code> to the value specified by <code>cov</code>.</p> <p>The model parameter covariance is calculated and stored automatically when a model is estimated. Therefore, you do not need to set the parameter covariance explicitly for estimated models. Use this function for analysis, such as to study how the parameter covariance affects the response of a model obtained by explicit construction.</p>
<b>Input Arguments</b>	<p><b>sys0</b> Identified model.</p> <p><b>cov</b> Parameter covariance matrix.</p> <p><code>cov</code> is one of the following:</p> <ul style="list-style-type: none"><li>• an <math>np</math>-by-<math>np</math> semi-positive definite symmetric matrix, where <math>np</math> is equal to the number of parameters of <code>sys0</code>.</li><li>• a structure with the following fields that describe the parameter covariance in a factored form:<ul style="list-style-type: none"><li>▪ <code>R</code> — usually the Cholesky factor of inverse of covariance.</li><li>▪ <code>T</code> — transformation matrix.</li><li>▪ <code>Free</code> — logical vector of length <math>np</math> indicating if a parameter is free. Here <math>np</math> is equal to the number of parameters of <code>sys0</code>.</li></ul></li></ul> <p><math display="block">\text{cov}(\text{Free},\text{Free}) = \text{T}*\text{inv}(\text{R}'*\text{R})*\text{T}'.</math></p>
<b>Output Arguments</b>	<p><b>sys</b> Identified model.</p>

The values of all the properties of `sys` are the same as those in `sys0`, except for the parameter covariance values which are modified as specified by `cov`.

## Examples

### Raw Covariance

Set raw covariance data for an identified model.

Create a covariance matrix for the transfer function

$$\text{sys0} = \frac{4}{s^2 + 2s + 1}$$

For this example, set the covariance values for only the denominator parameters.

```
sys0 = idtf(4,[1 2 1]);  
np = nparams(sys0);  
cov = zeros(np);  
den_index = 2:3;  
cov(den_index,den_index)=diag([0.04 0.001]);
```

`sys0` contains `np` model parameters.

`cov(den_index,den_index)=diag([0.04 0.001])` creates a covariance matrix, `cov`, with nonzero entries for the denominator parameters.

Set the covariance for `sys0`.

```
sys = setcov(sys0,cov);
```

## See Also

`getcov` | `rsample` | `sim` | `setpvec`

---

<b>Purpose</b>	Set initial states of idnlgrey model object
<b>Syntax</b>	<code>setinit(model,property,values)</code>
<b>Input Arguments</b>	<p><code>model</code> Name of the idnlgrey model object.</p> <p><code>property</code> Name of the InitialStates model property field, such as 'Name', 'Unit', 'Value', 'Minimum', 'Maximum', and 'Fixed'.</p> <p><code>values</code> Values of the specified property Property. <code>values</code> are an Nx-by-1 cell array of values, where Nx is the number of states.</p>
<b>Description</b>	<code>setinit(model,property,values)</code> sets the values of the property field of the InitialStates model property. <code>property</code> can be 'Name', 'Unit', 'Value', 'Minimum', 'Maximum', and 'Fixed'.
<b>See Also</b>	<code>getinit</code>   <code>getpar</code>   <code>idnlgrey</code>   <code>setpar</code>

# setoptions

---

**Purpose** Set plot options for response plot

**Syntax**  
`setoptions(h, PlotOpts)`  
`setoptions(h, 'Property1', 'value1', ...)`  
`setoptions(h, PlotOpts, 'Property1', 'value1', ...)`

**Description** `setoptions(h, PlotOpts)` sets preferences for response plot using the plot handle. `h` is the plot handle, `PlotOpts` is a plot options handle containing information about plot options.

There are two ways to create a plot options handle:

- Use `getoptions`, which accepts a plot handle and returns a plot options handle.

```
p = getoptions(h)
```

- Create a default plot options handle using one of the following commands:
  - `bodeoptions` — Bode plots
  - `hsvoptions` — Hankel singular values plots
  - `nicholsoptions` — Nichols plots
  - `nyquistoptions` — Nyquist plots
  - `pzoptions` — Pole/zero plots
  - `sigmaoptions` — Sigma plots
  - `timeoptions` — Time plots (step, initial, impulse, etc.)

For example,

```
p = bodeoptions
```

returns a plot options handle for Bode plots.

`setoptions(h, 'Property1', 'value1', ...)` assigns values to property pairs instead of using `PlotOpts`. To find out what



properties and values are available for a particular plot, type `help <function>options`. For example, for Bode plots type

```
help bodeoptions
```

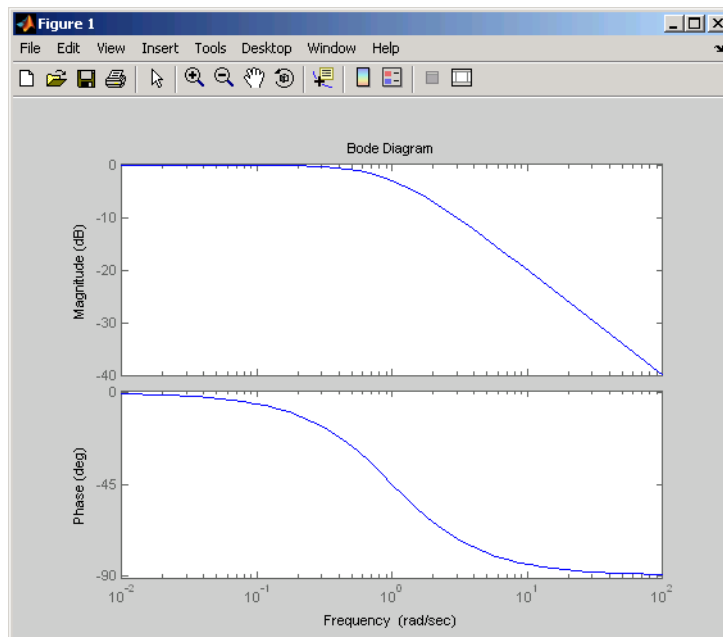
For a list of the properties and values available for each plot type, see “Properties and Values Reference”.

`setoptions(h, PlotOpts, 'Property1', 'value1', ...)` first assigns plot properties as defined in `@PlotOptions`, and then overrides any properties governed by the specified property/value pairs.

## Examples

To change frequency units, first create a Bode plot.

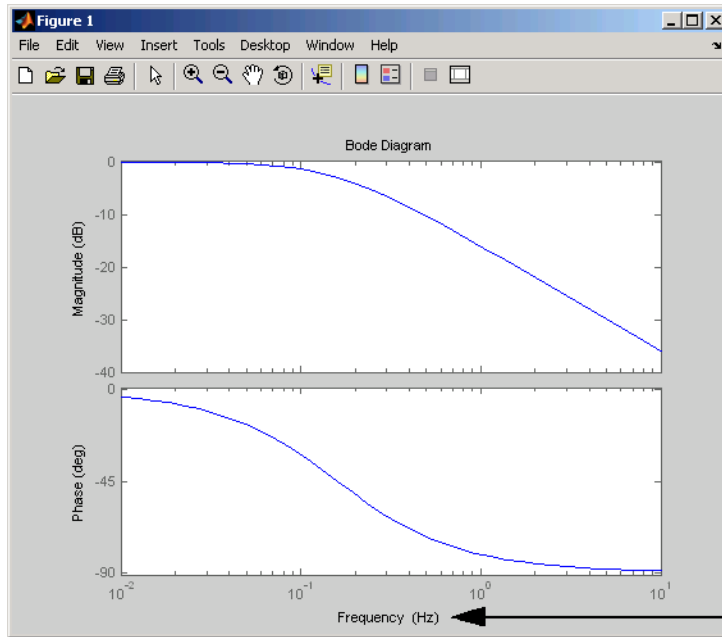
```
sys=tf(1,[1 1]);  
h=bodeplot(sys) % Create a Bode plot with plot handle h.
```



# setoptions

Now, change the frequency units from rad/s to Hz.

```
p=getoptions(h); % Create a plot options handle p.  
p.FreqUnits = 'Hz'; % Modify frequency units.  
setoptions(h,p); % Apply plot options to the Bode plot and  
% render.
```



To change the frequency units using property/value pairs, use this code.

```
sys=tf(1,[1 1]);  
h=bodeplot(sys);  
setoptions(h,'FreqUnits','Hz');
```

The result is the same as the first example.

## See Also

getoptions

---

<b>Purpose</b>	Set initial parameter values of idnlgrey model object
<b>Syntax</b>	<code>setpar(model,property,values)</code>
<b>Input Arguments</b>	<p><code>model</code> Name of the idnlgrey model object.</p> <p><code>property</code> Name of the Parameters model property field, such as 'Name', 'Unit', 'Value', 'Minimum', or 'Maximum'.  Default: 'Value'.</p> <p><code>values</code> Values of the specified property Property. <code>values</code> are an Np-by-1 cell array of values, where Np is the number of parameters.</p>
<b>Description</b>	<code>setpar(model,property,values)</code> sets the model parameter values in the property field of the Parameters model property. <code>property</code> can be 'Name', 'Unit', 'Value', 'Minimum', and 'Maximum'.
<b>See Also</b>	<code>getinit</code>   <code>getpar</code>   <code>idnlgrey</code>   <code>setinit</code>

# setpname

---

## Purpose

Set mnemonic parameter names for linear black-box model structures

---

**Note** `setpname` will be removed in a future release. Use the `Structure.Info` field of a linear model instead.

---

## 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 in `idproc`.

This function is particularly useful when certain parameters are to be fixed.

**Purpose** Specify format for B and F polynomials of multi-input polynomial model for backward compatibility

**Syntax**

```
model= setPolyFormat(model, 'cell')  
model= setPolyFormat(model, 'double')
```

**Description**

`model= setPolyFormat(model, 'cell')` converts the B and F polynomials of a multi-input polynomial model, `model`, from double matrices to cell arrays. Each cell array has Nu-elements of double vectors, where Nu is the number of inputs.

`model= setPolyFormat(model, 'double')` allows you to continue using double matrices for the B and F polynomials. The model displays a message that it is in backward compatibility mode.

- Tips**
- The *B* and *F* polynomials, represented by `b` and `f` properties of `idpoly` object, are currently double matrices. For multi-input polynomial models, these polynomials will be cell arrays in a future version. Using `setPolyFormat` allows you to either change to using cell arrays immediately or continue using double arrays without errors in a future version.
  - After using `model = setPolyFormat(model, 'cell')`, update your code to use cell array syntax for operations on the `b` and `f` properties.
  - After using `model = setPolyFormat(model, 'double')`, you can continue using double matrix syntax for operations on the `b` and `f` properties.
  - `setPolyFormat` has no effect on single-input `idpoly` models, where the `b` and `f` properties continue to be represented by double row vectors.

**Examples** Convert the B and F polynomials of an estimated multi-input ARX model to cell arrays:

- 1 Estimate a 3-input ARX model.

# setPolyFormat

---

```
% Load estimation data.
load iddata8
% Estimate model.
m = arx(z8, [3 [2 2 1] [1 1 1]]);
```

The software computes the  $B$  and  $F$  polynomials and stores their values as double matrices in the `b` and `f` properties. Operations on the  $B$  and  $F$  polynomials, such as `m.b`, produce an incompatibility warning.

## 2 Convert the $B$ and $F$ polynomials to cell arrays.

```
m=setPolyFormat(m, 'cell');
```

To verify that the  $B$  and  $F$  polynomials are cell arrays, type `class(m.b)`, which returns:

```
ans =

cell
```

## 3 Extract pole and zero information from the model using cell array syntax.

```
Poles1 = roots(m.f{1});
Zeros1 = roots(m.b{1});
```

---

Continue using double matrices for  $B$  and  $F$  polynomials of an estimated multi-input ARX model:

## 1 Estimate a 3-input ARX model.

```
% Load estimation data.
load iddata8
% Estimate model.
m = arx(z8, [3 [2 2 1] [1 1 1]]);
```

The software computes the  $B$  and  $F$  polynomials, and stores their values in double matrices in the `b` and `f` properties. Operations on the  $B$  and  $F$  polynomials, such as `m.b`, produce an incompatibility warning.

- 2 Designate the model to continue using double matrices for the  $B$  and  $F$  polynomials for backward compatibility.

```
m=setPolyFormat(m,'double')
```

The following message at the MATLAB prompt indicates that the model is backward compatible:

```
(model configured to operate in backward compatibility mode)
```

- 3 Extract pole and zero information from the model using matrix syntax.

```
Poles1 = roots(m.f(1,:))  
Zeros1 = roots(m.b(1,:))
```

## See Also

`idpoly` | `get` | `set` | `polydata` | `tfdata`

## How To

- “Extracting Numerical Model Data”

# setpvec

---

**Purpose** Modify value of model parameters

**Syntax**  
`sys = setpvec(sys0,par)`  
`sys = setpvec(sys0,par,'free')`

**Description** `sys = setpvec(sys0,par)` modifies the value of the parameters of the identified model `sys0` to the value specified by `par`.

`par` must be of length `nparams(sys0)`. `nparams(sys0)` returns a count of all the parameters of `sys0`.

`sys = setpvec(sys0,par,'free')` modifies the value of all the free estimation parameters of `sys0` to the value specified by `par`.

`par` must be of length `nparams(sys0,'free')`. `nparams(sys0,'free')` returns a count of all the free parameters of `sys0`.

## Input Arguments

### **sys0**

Identified model containing the parameters whose value is modified to `par`.

### **par**

Replacement value for the parameters of the identified model `sys0`.

For the syntax `sys = setpvec(sys0,par)`, `par` must be of length `nparams(sys0)`. `nparams(sys0)` returns a count of all the parameters of `sys0`.

For the syntax `sys = setpvec(sys0,par,'free')`, `par` must be of length `nparams(sys0,'free')`. `nparams(sys0,'free')` returns a count of all the free parameters of `sys0`.

Use NaN to denote unknown parameter values.

If `sys0` is an array of models, then specify `par` as a cell array with an entry corresponding to each model in `sys0`.



**Output Arguments****sys**

Identified model obtained from `sys0` by updating the values of the specified parameters.

**Examples**

Modify the parameter values of a transfer function model.

The goal here is to ultimately use the transfer function model to initialize a model estimation.

```
sys0 = idtf(1,[1 2]);  
par = [1; NaN; 0];  
sys = setpvec(sys0,par);
```

---

Modify the value of the free parameters of a transfer function model.

```
sys0 = idtf([1 0],[1 2 0]);  
sys0.Structure.den.Free(3) = false  
par = [1; 2; 1]  
sys = setpvec(sys0,par,'free');
```

**See Also**

`getpvec` | `setcov` | `nparams`

# setstruc

---

**Purpose** Set matrix structure for idss model objects

---

**Note** setstruc will be removed in a future release. Use setpvec and the Structure property of idss objects instead.

---

**Syntax**

```
setstruc(M,As,Bs,Cs,Ds,Ks,X0s)
setstruc(M,Mods)
```

**Description** setstruc(M,As,Bs,Cs,Ds,Ks,X0s)

is the same as

```
set(M,'As',As,'Bs',Bs,'Cs',Cs,'Ds',Ds,'Ks',Ks,'X0s',X0s)
```

Use empty matrices for structure matrices that should not be changed. You can omit trailing arguments.

In the alternative syntax, `Mods` is a structure with field names `As`, `Bs`, etc., with the corresponding values of the fields.

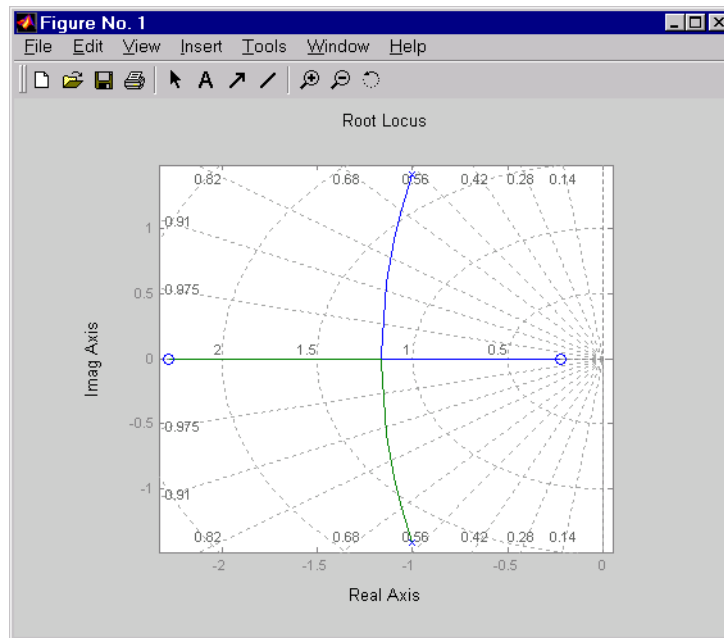
**See Also** idss

- Purpose** Generate s-plane grid of constant damping factors and natural frequencies
- Syntax** `sgrid`  
`sgrid(z,wn)`
- Description** `sgrid` generates, for pole-zero and root locus plots, a grid of constant damping factors from zero to one in steps of 0.1 and natural frequencies from zero to 10 rad/sec in steps of one rad/sec, and plots the grid over the current axis. If the current axis contains a continuous s-plane root locus diagram or pole-zero map, `sgrid` draws the grid over the plot.
- `sgrid(z,wn)` plots a grid of constant damping factor and natural frequency lines for the damping factors and natural frequencies in the vectors `z` and `wn`, respectively. If the current axis contains a continuous s-plane root locus diagram or pole-zero map, `sgrid(z,wn)` draws the grid over the plot.
- Alternatively, you can select **Grid** from the right-click menu to generate the same s-plane grid.
- Examples** Plot s-plane grid lines on the root locus for the following system.

$$H(s) = \frac{2s^2 + 5s + 1}{s^2 + 2s + 3}$$

You can do this by typing

```
H = tf([2 5 1],[1 2 3])
Transfer function:
2 s^2 + 5 s + 1
-----
s^2 + 2 s + 3
rlocus(H)
sgrid
```



**See Also** [pzmap](#) | [rlocus](#) | [zgrid](#)

<b>Purpose</b>	Display confidence regions on response plots for identified models
<b>Syntax</b>	<code>showConfidence(plot_handle)</code> <code>showConfidence(plot_handle, sd)</code>
<b>Description</b>	<code>showConfidence(plot_handle)</code> displays the confidence region on the response plot, with handle <code>plot_handle</code> , for an identified model. <code>showConfidence(plot_handle, sd)</code> displays the confidence region for <code>sd</code> standard deviations.

## Input Arguments

### **plot\_handle**

Response plot handle.

`plot_handle` is the handle for the response plot of an identified model on which the confidence region is displayed. It is obtained as an output of one of the following plot commands: `bodeplot`, `stepplot`, `impzplot`, `nyquistplot` and `iopzplot`.

### **sd**

Standard deviation of the confidence region. A common choice is 3 standard deviations, which gives 99.7% significance.

#### **Default:**

```
getoptions(plot_handle, 'ConfidenceRegionNumberSD')
```

## Examples

### **View Confidence Region for Identified Model**

Show the confidence bounds on the bode plot of an identified ARX model.

Obtain identified model and plot its bode response.

```
load iddata1 z1
sys = arx(z1, [2 2 1]);
h = bodeplot(sys);
```

`z1` is an `iddata` object that contains time domain system response data.

# showConfidence

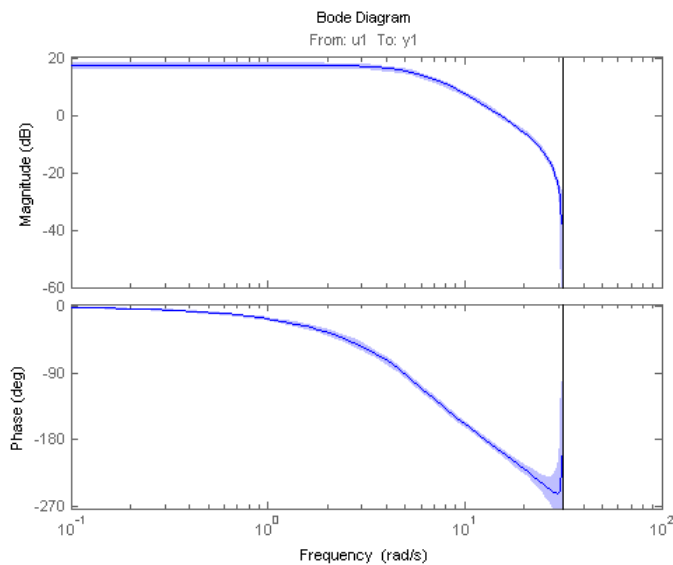
---

`sys` is an `idpoly` model containing the identified polynomial model.

`h` is the plot handle for the bode response plot of `sys`.

Show the confidence bounds for `sys`.

```
showConfidence(h);
```



This plot depicts the confidence region for 1 standard deviation.

## **Specify the Standard Deviation of the Confidence Region**

Show the confidence bounds on the bode plot of an identified ARX model.

Obtain identified model and plot its bode response.

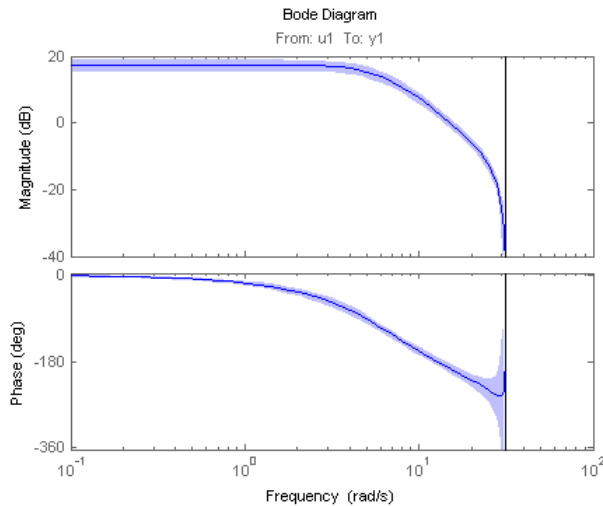
```
load iddata1 z1
sys = arx(z1, [2 2 1]);
h = bodeplot(sys);
```

`z1` is an `iddata` object that contains time domain system response data.  
`sys` is an `idpoly` model containing the identified polynomial model.  
`h` is the plot handle for the bode response plot of `sys`.

Show the confidence bounds for `sys` using 2 standard deviations.

```
sd = 2;  
showConfidence(h,sd);
```

`sd` specifies the number of standard deviations for the confidence region displayed on the plot.



## Alternatives

You may turn on the confidence region display on a response plot interactively. Right-click on the response plot and select **Characteristics > Confidence Region**.

## See Also

`bodeplot` | `stepplot` | `impzplot` | `nyquistplot` | `iopzplot`

# 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 2-807.

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

**Tips** 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:

$$F(x) = (x-r)PL + a_1f((x-r)Qb_1 + c_1) + \dots \\ + a_nf((x-r)Qb_n + c_n) + d$$

where  $f$  is the sigmoid function, given by the following equation:

$$f(z) = \frac{1}{e^{-z} + 1}.$$

$P$  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.

$d$ ,  $\alpha$ , and  $c$  are scalars.

$L$  is a  $p-by-1$  vector.

$b$  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.

# sigmoidnet

Property Name	Description
NumberOfUnits	<p>Integer specifies the number of nonlinearity units in the expansion. Default=10.</p> <p>For example:</p> <pre>sigmoidnet(H, 'NumberOfUnits', 5)</pre>
LinearTerm	<p>Can have the following values:</p> <ul style="list-style-type: none"><li>• 'on'—Estimates the vector <math>L</math> in the expansion.</li><li>• 'off'—Fixes the vector <math>L</math> to zero.</li></ul> <p>For example:</p> <pre>sigmoidnet(H, 'LinearTerm', 'on')</pre>
Parameters	<p>A structure containing the parameters in the nonlinear expansion, as follows:</p> <ul style="list-style-type: none"><li>• RegressorMean: 1-by-m vector containing the means of <math>x</math> in estimation data, <math>r</math>.</li><li>• NonLinearSubspace: m-by-q matrix containing <math>Q</math>.</li><li>• LinearSubspace: m-by-p matrix containing <math>P</math>.</li><li>• LinearCoef: p-by-1 vector <math>L</math>.</li><li>• Dilation: q-by-n matrix containing the values <math>b_n</math>.</li><li>• Translation: 1-by-n vector containing the values <math>c_n</math>.</li><li>• OutputCoef: n-by-1 vector containing the values <math>a_n</math>.</li><li>• OutputOffset: scalar <math>d</math>.</li></ul> <p>Typically, the values of this structure are set by estimating a model with a <code>sigmoidnet</code> nonlinearity.</p>

## Algorithms

`sigmoidnet` uses an iterative search technique for estimating parameters.

## 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 response of identified models to arbitrary inputs

**Syntax**

```
y = sim(sys,data)
y = sim(sys,data,opt)
[y,y_sd] = sim(sys,data, ___)
[y,y_sd,x] = sim(sys,data, ___)
```

**Description** `y = sim(sys,data)` simulates the response an identified model, `sys`, using the input data, `data`. `y` is the simulation output.

`y = sim(sys,data,opt)` simulates the system response using the option set, `opt`, to specify simulation behavior.

`[y,y_sd] = sim(sys,data, ___)` also returns the estimated standard deviation, `y_sd`, for `sys`.

`[y,y_sd,x] = sim(sys,data, ___)` also returns the state trajectory, `x`, for state-space models.

- `sim` extends `lsim` to facilitate additional features relevant to identified models:
  - simulation of nonlinear models
  - simulation with additive noise
  - incorporation of signal offsets
  - computation of response standard deviation (linear models only)
  - frequency-domain simulation (linear models only)
  - simulations using different intersample behavior for different inputs

To obtain the simulated response without any of the above operations, use `lsim`.

- Tips**
- You can specify initial conditions for simulation by creating an option set using `simOptions` and then setting the `InitialCondition` option appropriately.

For multi-experiment data, you can configure each experiment's initial conditions individually.

- You can simulate the initial condition response of time-series models (models with no inputs) using `sim`. To do this, specify `data` as an  $N_s$ -by-0 signal, where  $N_s$  is the number of samples. As with input-output models, you can study the effect of noise on the response by using the `AddNoise` and `NoiseData` simulation options. For more information regarding these simulation options, see `simOptions`.

For example:

```
load iddata9 z9;
sys = ar(z9,4,'ls');
data = iddata([],zeros(512,0),z9.Ts);
opt = simOptions('AddNoise',true);
y = sim(sys,data,opt);
```

- You can specify a custom noise signal to be added to the simulated response by creating an option set using `simOptions` and then setting the `NoiseData` option appropriately.

## Input Arguments

### **sys**

Identified model.

`sys` may be a linear or nonlinear identified model.

### **data**

Simulation input data.

Specify data as an `iddata` object, using only the input channels.

If `sys` is a linear model, you can use either time- or frequency-domain data. If `sys` is a nonlinear model, you can use only time-domain data.

For time-domain simulation of discrete-time systems, `data` may also be specified as a matrix whose columns correspond to each input channel.

If you do not have data from an experiment, use `idinput` to generate signals of various characteristics.

## Output Arguments

### **opt**

Simulation options.

`opt` is an option set that specifies the following:

- signal offsets
- initial condition handling
- additive noise

Use `simOptions` to create the options set.

### **y**

Simulated response.

`y` is an `iddata` model representing the simulated response for `sys` using `data` as the simulation input.

If `data` represents time-domain data, then `y` is the simulated response for the time vector corresponding to `data`.

If `data` represents frequency-domain data, then `y` contains the Fourier transform of the corresponding sampled time-domain output signal obtained by the multiplication of the frequency response of `sys`,  $G(w)$  and  $U(w)$ .

For multi-experiment data, `y` is a corresponding multi-experiment `iddata` object.

### **y\_sd**

Estimated standard deviation of the simulated response for linear models.

`y_sd` is derived using first order sensitivity considerations (Gauss approximation formula).

For nonlinear models, `y_sd` is `[]`.

### **x**

Estimated state trajectory for state-space models.

Relevant only if `sys` is a state-space model (`idss`, `idgrey` or `idnlgrey`).

`x` is an  $N_s$ -by- $N_x$  matrix, where  $N_s$  is the number of samples and  $N_x$  is the number of states.

## Examples

### Simulate Model Response

Simulate the response of an identified model.

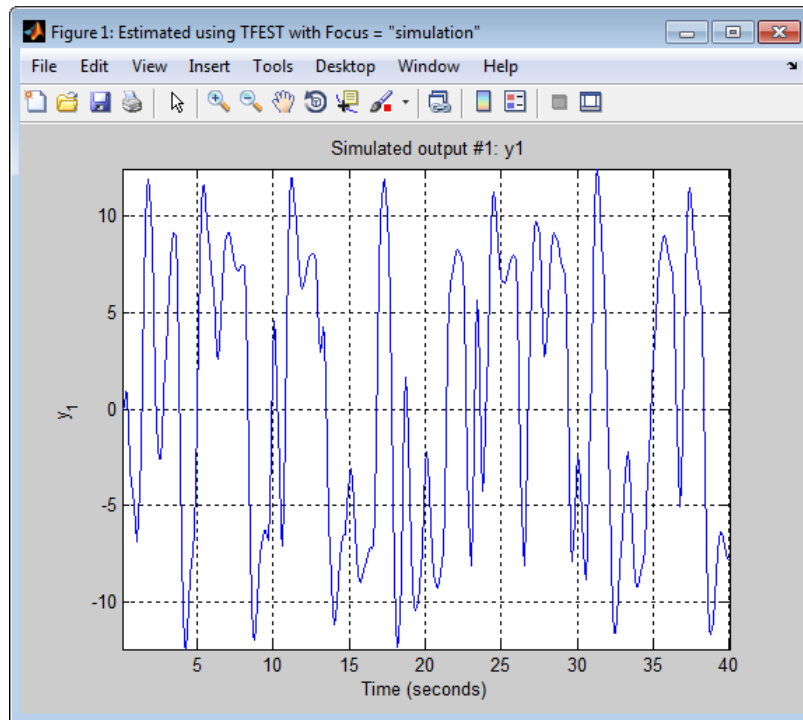
Obtain the identified model.

```
load iddata2 z2;  
sys = tfest(z2,3);
```

`sys` is an `idtf` model that encapsulates the third-order transfer function estimated for the measured data `z2`.

Simulate the model.

```
sim(sys, z2);
```



### Specify Simulation Option

Simulate the model response of an identified model. Specify simulation options to study the contribution of noise to the simulated model response.

Obtain the identified model.

```
load iddata2 z2;  
sys = tfest(z2,3);
```

sys is an idtf model that encapsulates the third-order transfer function estimated for the measured data z2.



Create a simulation option set that adds noise to the simulated model response.

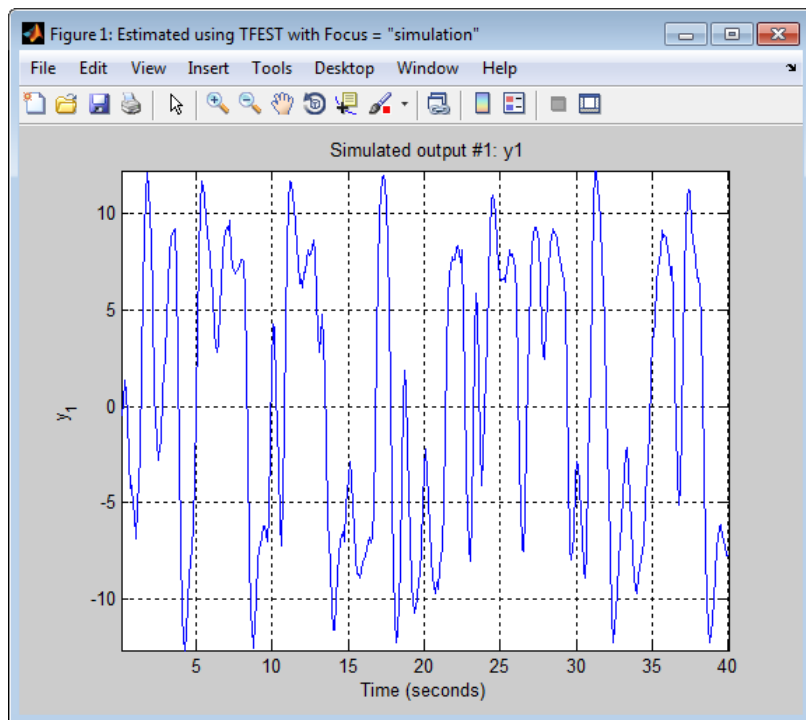
```
e = randn(length(z2.u),1);  
opt = simOptions('AddNoise',true,'NoiseData',e);
```

`e` represents white, Gaussian noise.

`opt` is an option set that specifies the addition of noise data, `e`, to the simulated model response. You specify the noise data vector, `e`, that is added to the simulated model response by using the option `NoiseData`.

Obtain the simulated model response.

```
sim(sys,z2,opt);
```



# sim

---

## Alternatives

- Use `simsd` for a Monte-Carlo method of computing the standard deviation of the response.

## See Also

`simOptions` | `simsd` | `lsim` | `step` | `compare` | `predict` | `forecast` | `idinput`

## 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.
- Matrices: `E` has as many columns as there are noise signals, corresponding to the number of model outputs.

## Input Arguments

- `MODEL`: `idnlarx` model object.
- `U`: Input data for simulation, an `iddata` object (where only the input channels are used) or a matrix. For simulations with noisy data, `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

# sim(idnlarx)

---

`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.

## Output Arguments

- `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 a SISO `idnlarx` model

In this example you simulate a single-input single-output `idnlarx` model `M` around a known equilibrium point, with an input level of 1 and output level of 10.

- 1 Load the sample data.

```
load iddata2;
```

- 2 Estimate an `idnlarx` model from the 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);
```

## 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.

```
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);
```

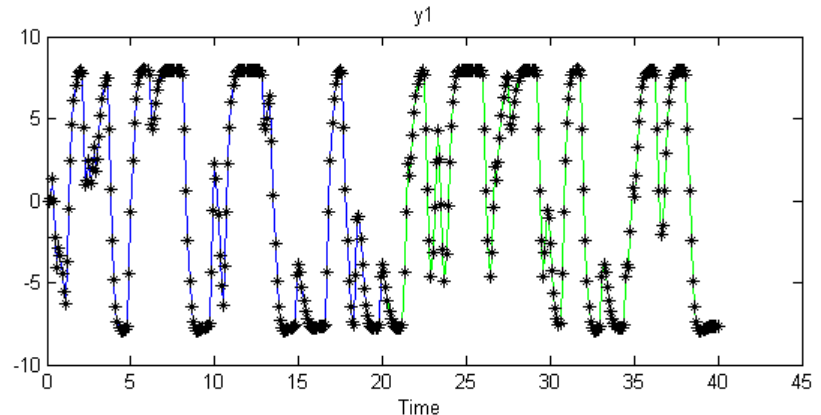
- 6 Verify the two simulations by comparing to a complete simulation using all the input data of z2.

# sim(idnlarx)

```
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 the three responses
plot(ys1,'b', ys2, 'g', ysTotal, 'k*')
```

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.

```
ysim = sim(M, z2.u, 'init', x0)
```

- 5 Compare ysim with the output signal in z2:

```
time = z2.SamplingInstants;  
plot(time, ysim, time, z2.y, '.')
```

## Simulation Near Steady State with Known Input and Unknown Output

In this example you start simulation of a model near steady state, 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)
```

# sim(idnlarx)

---

## See Also

`predict` | `findop(idnlarx)` | `data2state(idnlarx)` |  
`findstates(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)`; simulates the model 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.
- Matrices: `E` has as many columns as there are noise signals, corresponding to the number of model outputs.

# sim(idnlgrey)

---

## Input Arguments

- 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.
- X0INIT: Initial states. Can have the following values:
  - 'zero' : Zero initial state  $x(0)$  with all states fixed (`nlsys.InitialStates.Fixed` is thus ignored).
  - 'fixed' (default): Struct array (`NLSYS.InitialState`) determines the values of the model initial states and all states are fixed. (`NLSYS.InitialStates.Fixed` is ignored). Same as 'model'.
  - vector/matrix: Column vector of initial state values. For multiple experiment DATA, X0INIT 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 1-by-Ne vector of True values). Any false value is ignored.

## Output Arguments

- **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 ith 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.

```
% Specify file.
FileName = 'twotanks_c';
% Specify model orders [ny nu nx].
Order = [1 1 2];
% Specify initial parameters.
Parameters = {0.5; 0.0035; 0.019; ...
              9.81; 0.25; 0.016};
```

## sim(idnlgrey)

---

```
% Specify initial states.  
InitialStates = [0; 0.1];  
Ts = 0;  
nlgr = idnlgrey(FileName, Order, Parameters, ...  
               InitialStates, Ts, ...  
               'Name', 'Two tanks');
```

**3** Create an iddata object z from data z (from twotankdata.mat).

```
z = iddata([], u, 0.2, 'Name', 'Two tanks');
```

**4** Simulate the model using the input signal from the data z.

```
sim(nlgr,z)
```

### See Also

[findstates\(idnlgrey\)](#) | [pe](#) | [pem](#) | [predict](#)

## 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.
- Matrices: `E` has as many columns as there are noise signals, corresponding to the number of model outputs.

## Input Arguments

- `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 simulations with noisy data, `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.

# sim(idnlhw)

---

- 'z': (Default) Vector containing zeros and corresponding to a system starting from rest.

## Output Arguments

- 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 = n1hw(z2,[4 3 2], 'wave', 'pw1');
```

- 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.

```
ysim = sim(M,z2.u,'init',x0)
```

- 5 Compare `ysim` to output signal in `z2`:

```
t = z2.samp;
```

```
plot(t, ysim, t, z2.y)
```

### 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-Wiener model.

```
M = nlhw(z2,[4 3 2], 'wave', 'pw1');
```

- 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)
```

### See Also

[findop\(idnlhw\)](#) | [findstates\(idnlhw\)](#) | [predict](#)

# simOptions

---

**Purpose** Option set for sim

**Syntax**  
`opt = simOptions`  
`opt = simOptions(Name,Value)`

**Description** `opt = simOptions` creates the default options set for sim.  
`opt = simOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### InitialCondition

Specify initial conditions.

`InitialCondition` takes one of the following:

- 'z' — Zero initial conditions.
- `x0` — Numerical column vector denoting initial states. For multi-experiment data, use a matrix with  $N_e$  columns, where  $N_e$  is the number of experiments. Use this option for state-space models (`idss` and `idgrey`) only.
- `io` — Structure with the following fields:
  - `Input`
  - `Output`

Use the `Input` and `Output` fields to specify the history for a time interval that starts before the start time of the data used by `compare`. In case the data used by `compare` is a time series model, specify `Input` as `[]`. Use a row vector to denote a constant signal



value. The number of columns in `Input` and `Output` must always equal the number of input and output channels, respectively. For multi-experiment data, specify `io` as a struct array of  $N_e$  elements, where  $N_e$  is the number of experiments.

**Default:** []

### **InputOffset**

Input signal offset.

Specify as a column vector of length  $N_u$ , where  $N_u$  is the number of inputs.

Use [] to indicate no offset.

For multi-experiment data, specify `InputOffset` as a  $N_u$ -by- $N_e$  matrix. Here  $N_u$  is the number of inputs and  $N_e$  is the number of experiments.

Each entry specified by `InputOffset` will be subtracted from the corresponding input data before it is used to simulate the model.

**Default:** []

### **OutputOffset**

Output signal offset.

Specify as a column vector of length  $N_y$ , where  $N_y$  is the number of outputs.

Use [] to indicate no offset.

For multi-experiment data, specify `OutputOffset` as a  $N_y$ -by- $N_e$  matrix. Here  $N_y$  is the number of outputs and  $N_e$  is the number of experiments.

Each entry specified by `OutputOffset` will be added to the corresponding simulated response of the model.

**Default:** []

## AddNoise

Specify whether noise should be added to the response model or not.

**Default:** false

## NoiseData

Noise signal data.

Specify the noise signal,  $e$ , for the model

$$y(t) = Gu(t) + He(t)$$

Where  $G$  is the transfer function from the input,  $u(t)$ , to the output,  $y(t)$ .

NoiseData is used for simulation only when AddNoise is true.

NoiseData takes one of the following:

- matrix —  $N_s$ -by- $N_y$  matrix, where  $N_s$  is the number of input data samples and  $N_y$  is the number of outputs. Each entry of this matrix is added to the corresponding output data point. Before addition, the noise is scaled according to the NoiseVariance property of the identified model used in simsd.

To obtain the right noise level, specify NoiseData as white noise with zero mean and unit covariance.

- cell array — For multi-experiment data, specify NoiseData as a cell array of  $N_e$  matrices. Here  $N_e$  is the number of experiments.
- [] — Gaussian noise is automatically specified as NoiseData.

**Default:** []

## Output Arguments

### opt

Option set containing the specified options for sim.

## Examples

### Create Default Options Set for Model Simulation

Create a default options set for `sim`.

```
opt = simOptions;
```

### Specify Options for Model Simulation

Create an options set for `sim` using zero initial conditions and set the input offset to 5.

```
opt = simOptions('InitialCondition','z','InputOffset',5);
```

Alternatively, use dot notation to set the values of `opt`.

```
opt = simOptions;  
opt.InitialCondition = 'z';  
opt.InputOffset = 5;
```

**See Also** `sim`

**Purpose** Simulate linear models with uncertainty using Monte Carlo method

**Syntax**

```
simsd(sys,data)
simsd(sys,data,N)
simsd(sys,data,N,opt)
y = simsd(sys,data,N,opt)
[y,y_sd] = simsd(sys,data,N,opt)
```

**Description** `simsd(sys,data)` simulates and plots the response of 10 perturbed realizations of the identified model `sys`. Simulation input data, `data`, is used to compute the simulated response.

The parameters of the perturbed realizations are consistent with the parameter covariance of the original model, `sys`.

`simsd(sys,data,N)` simulates and plots the response of `N` perturbed realizations of the identified model `sys`.

`simsd(sys,data,N,opt)` simulates the system response using the option set, `opt`, to specify simulation behavior.

`y = simsd(sys,data,N,opt)` returns the simulation result as a cell array, `y`. No simulated response plot is produced.

`[y,y_sd] = simsd(sys,data,N,opt)` also returns the estimated standard deviation, `y_sd`, for the simulated response.

The parameter changes in the randomly selected models are scaled to be small (ca 0.1%) compared to the parameter values. The response changes are then scaled up to correspond to one standard deviation. The scaling does not apply to free delays of `idproc` or `idtf` models.

- Tips**
- You can specify initial conditions for simulation by creating an option set using `simsdOptions` and then setting the `InitialCondition` option appropriately.
  - `simsd` yields meaningful results only when `sys` contains information regarding parameter uncertainty. Use `getcov` to examine the parameter uncertainty for `sys`. For models with no parameter uncertainty data, the results of `simsd` match that of `sim`.

## Input Arguments

### **sys**

Identified linear model.

### **data**

Simulation input data.

Specify **data** as a time- or frequency-domain `iddata` object, with input channels only.

For time-domain simulation of discrete-time systems, **data** may also be specified as a matrix whose columns correspond to each input channel.

### **N**

Number of perturbed realizations for simulation.

Specify **N** as a positive integer.

**Default:** 10

### **opt**

Simulation options.

**opt** is an option set that specifies the following:

- signal offsets
- initial condition handling
- additive noise

Use `simsdOptions` to create the options set.

## Output Arguments

### **y**

Simulated response.

**y** is a cell array of  $N+1$  elements, where  $N$  is the number of perturbed realizations. **y**{1} contains the nominal response for **sys**. The remaining elements contain the simulated response for the  $N$  perturbed realizations.

## **y\_sd**

Estimated standard deviation of the simulated response.

y\_sd is derived by averaging the simulations results in y.

## **Examples**

### **Simulate Estimated Model Using Monte-Carlo Method**

Simulate an estimated model using the Monte-Carlo method for a specified number of model perturbations.

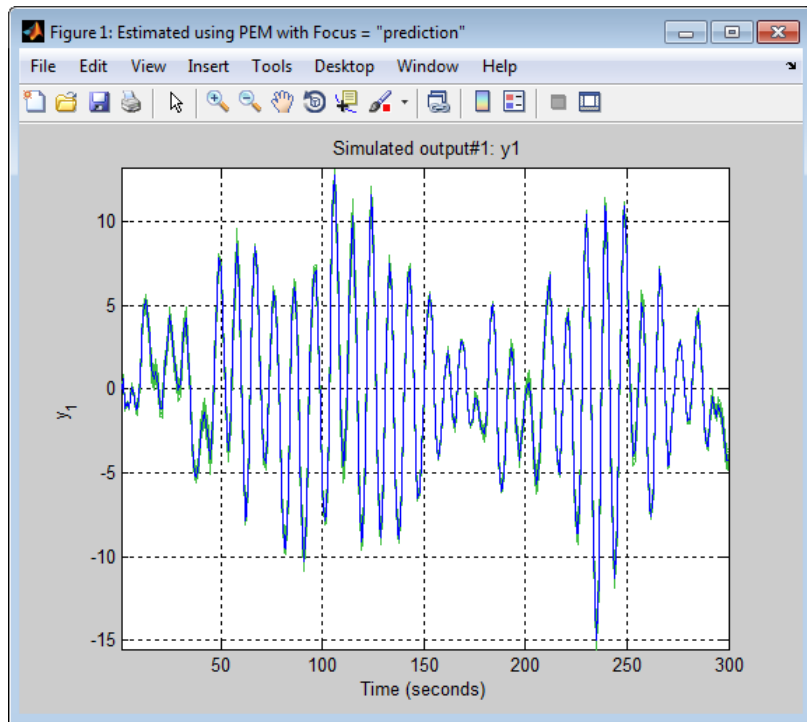
Obtain an identified model.

```
load iddata3  
sys = ssest(z3,2)
```

sys is an idss model that encapsulates the estimated second-order state-space model for the measured data, z3.

Simulate the estimated model using the Monte-Carlo method. Specify the number of random model perturbations.

```
N = 20;  
simsd(sys,z3,N)
```

**See Also**

`simsdOptions` | `getcov` | `sim` | `rsample` | `showConfidence`

# simsoptions

---

**Purpose** Option set for simsoptions

**Syntax**  
`opt = simsoptions`  
`opt = simsoptions(Name,Value)`

**Description** `opt = simsoptions` creates the default options set for simsoptions.  
`opt = simsoptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### InitialCondition

Specify initial conditions.

`InitialCondition` takes one of the following:

- 'z' — Zero initial conditions.
- `x0` — Numerical column vector denoting initial states. For multi-experiment data, use a matrix with  $N_e$  columns, where  $N_e$  is the number of experiments. Use this option for state-space models (`idss` and `idgrey`) only.
- `io` — Structure with the following fields:
  - `Input`
  - `Output`

Use the `Input` and `Output` fields to specify the history for a time interval that starts before the start time of the data used by `compare`. In case the data used by `compare` is a time series model, specify `Input` as `[]`. Use a row vector to denote a constant signal



value. The number of columns in `Input` and `Output` must always equal the number of input and output channels, respectively. For multi-experiment data, specify `io` as a struct array of  $N_e$  elements, where  $N_e$  is the number of experiments.

### **InputOffset**

Input signal offset.

Specify as a column vector of length  $N_u$ , where  $N_u$  is the number of inputs.

Use `[]` to indicate no offset.

For multi-experiment data, specify `InputOffset` as a  $N_u$ -by- $N_e$  matrix. Here  $N_u$  is the number of inputs and  $N_e$  is the number of experiments.

Each entry specified by `InputOffset` will be subtracted from the corresponding input data before it is used to simulate the model.

**Default:** `[]`

### **OutputOffset**

Output signal offset.

Specify as a column vector of length  $N_y$ , where  $N_y$  is the number of outputs.

Use `[]` to indicate no offset.

For multi-experiment data, specify `OutputOffset` as a  $N_y$ -by- $N_e$  matrix. Here  $N_y$  is the number of outputs and  $N_e$  is the number of experiments.

Each entry specified by `OutputOffset` is added to the simulated response of the model.

**Default:** `[]`

### **AddNoise**

# simsoptions

---

Specify whether noise should be added to the response model or not.

**Default:** false

## NoiseData

Noise signal data.

Specify the noise signal,  $e$ , for the model

$$y(t) = Gu(t) + He(t)$$

Where  $G$  is the transfer function from the input,  $u(t)$ , to the output,  $y(t)$ .

NoiseData is used for simulation only when AddNoise is true.

NoiseData takes one of the following:

- matrix —  $N_s$ -by- $N_y$  matrix, where  $N_s$  is the number of input data samples and  $N_y$  is the number of outputs. Each entry of this matrix is added to the corresponding output data point. Before addition, the noise is scaled according to the NoiseVariance property of the identified model used in simsoptions.

To obtain the right noise level, specify NoiseData as white noise with zero mean and unit covariance.

- cell array — For multi-experiment data, specify NoiseData as a cell array of  $N_e$  matrices. Here  $N_e$  is the number of experiments.
- [] — Gaussian noise is automatically specified as NoiseData.

**Default:** []

## Output Arguments

**opt**

Option set containing the specified options for simsoptions.

## Examples

### Create Default Options Set for Model Simulation

Create a default options set for simsoptions.

```
opt = simsdOptions;
```

## Specify Options for Model Simulation

Create an options set for `simsd` using zero initial conditions and set the input offset to 5.

```
opt = simsdOptions('InitialCondition','z','InputOffset',5);
```

Alternatively, use dot notation to set the values of `opt`.

```
opt = simsdOptions;  
opt.InitialCondition = 'z';  
opt.InputOffset = 5;
```

**See Also** `simsd`

# size

---

**Purpose** Query output/input/array dimensions of input–output model and number of frequencies of FRD model

**Syntax**

```
size(sys)
d = size(sys)
Ny = size(sys,1)
Nu = size(sys,2)
Sk = size(sys,2+k)
Nf = size(sys,'frequency')
```

**Description** When invoked without output arguments, `size(sys)` returns a description of type and the input-output dimensions of `sys`. If `sys` is a model array, the array size is also described. For identified models, the number of free parameters is also displayed. The lengths of the array dimensions are also included in the response to `size` when `sys` is a model array.

`d = size(sys)` returns:

- The row vector `d = [Ny Nu]` for a single dynamic model `sys` with `Ny` outputs and `Nu` inputs
- The row vector `d = [Ny Nu S1 S2 ... Sp]` for an `S1-by-S2-by-...-by-Sp` array of dynamic models with `Ny` outputs and `Nu` inputs

`Ny = size(sys,1)` returns the number of outputs of `sys`.

`Nu = size(sys,2)` returns the number of inputs of `sys`.

`Sk = size(sys,2+k)` returns the length of the `k`-th array dimension when `sys` is a model array.

`Nf = size(sys,'frequency')` returns the number of frequencies when `sys` is a frequency response data model. This is the same as the length of `sys.frequency`.

## Examples

### Example 1

Consider the model array of random state-space models

```
sys = rss(5,3,2,3);
```

Its dimensions are obtained by typing

```
size(sys)
3x1 array of state-space models
Each model has 3 outputs, 2 inputs, and 5 states.
```

### Example 2

Consider the process model:

```
sys = idproc({'p1d', 'p2'; 'p3uz', 'p0'});
```

It's input-output dimensions and number of free parameters are obtained by typing:

```
size(sys)
```

Process model with 2 outputs, 2 inputs and 12 free parameters.

### See Also

`isempty` | `issiso` | `ndims` | `nparams`

**Purpose** Estimate frequency response with fixed frequency resolution using spectral analysis

**Syntax**

```
G = spa(data)
G = spa(data,winSize,freq)
G = spa(data,winSize,freq,MaxSize)
```

**Description** `G = spa(data)` estimates frequency response (with uncertainty) and noise spectrum from time- or frequency-domain data. `data` is an `iddata` or `idfrd` object and can be complex valued. `G` is as an `idfrd` object. For time-series data, `G` is the estimated spectrum and standard deviation.

`G = spa(data,winSize,freq)` estimates frequency response at frequencies `freq`. `freq` is a row vector of values in rad/sec. `winSize` is a scalar integer that sets the size of the Hann window.

`G = spa(data,winSize,freq,MaxSize)` can improve computational performance using `MaxSize` to split the input-output data such that each segment contains fewer than `MaxSize` elements. `MaxSize` is a positive integer.

## Definitions **Frequency Response Function**

*Frequency response function* describes the steady-state response of a system to sinusoidal inputs. For a linear system, a sinusoidal input of a specific frequency results in an output that is also a sinusoid with the same frequency, but with a different amplitude and phase. The frequency response function describes the amplitude change and phase shift as a function of frequency.

To better understand the frequency response function, consider the following description of a linear, dynamic system:

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

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

dynamics that take the input to the output. The notation  $G(q)u(t)$  represents the following operation:

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

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

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

$G(q)$  is the *frequency-response function*, which is evaluated on the unit circle,  $G(q=e^{i\omega})$ .

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

The frequency-response function estimated using the Blackman-Tukey approach is given by the following equation:

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

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

### Output Noise Spectrum

The output noise spectrum (spectrum of  $v(t)$ ) is given by the following equation:

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

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

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

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

where the noise spectrum is given by the following equation:

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

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

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

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

where  $e(t)$  is the white noise with variance  $\lambda$  and the noise power spectrum is given by the following equation:

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

## Examples

Estimate frequency response with fixed resolution at 128 equally spaced, logarithmic frequency values between 0 (excluded) and  $\pi$ :

```
load iddata3;
z = z3; % z is an iddata object with Ts=1
g = spa(z);
bode(g)
```



Estimate frequency response with fixed resolution at logarithmically spaced frequencies:

```
% Define frequency vector
w = logspace(-2,pi,128);
% Compute frequency response
g= spa(z,[],w); % [] specifies the default lag window size
h = bodeplot(g);
showConfidence(h,3)
figure
h = spectrumplot(g);
showConfidence(h,3)
% The plots include confidence interval
% of 3 standard deviations
```

## Algorithms

spa applies the Blackman-Tukey spectral analysis method by following these steps:

- 1 Computes the covariances and cross-covariance from  $u(t)$  and  $y(t)$ :

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

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

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

This portion of the algorithm uses the covf function.

- 2** Computes the Fourier transforms of the covariances and the cross-covariance:

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

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

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

where  $W_M(\tau)$  is the Hann window with a width (lag size) of  $M$ . You can specify  $M$  to control the frequency resolution of the estimate, which is approximately equal  $2\pi/M$  rad/sampling interval.

By default, this operation uses 128 equally spaced frequency values between 0 (excluded) and  $\pi$ , where  $w = [1:128]/128*\pi/T_s$  and  $T_s$  is the sampling interval of that data set. The default lag size of the Hann window is  $M = \min(\text{length}(\text{data})/10, 30)$ . For default frequencies, uses fast Fourier transforms (FFT)—which is more efficient than for user-defined frequencies.

---

**Note**  $M = \gamma$  is in Table 6.1 of Ljung (1999). Standard deviations are on pages 184 and 188 in Ljung (1999).

---

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

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

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

spectrum is the spectrum matrix for both the output and the input channels. That is, if  $z = [\text{data.OutputData}, \text{data.InputData}]$ , spectrum contains as spectrum data the matrix-valued power spectrum of  $z$ .

$$S = \sum_{m=-M}^M E z(t+m)z(t)' W_M(T_s) \exp(-i\omega m)$$

' is a complex-conjugate transpose.

## References

Ljung, L. *System Identification: Theory for the User*, Second Ed., Prentice Hall PTR, 1999.

## See Also

etfe | freqresp | idfrd | spafdr | bode | spectrum

## How To

- “Identifying Frequency-Response Models”
- “Spectrum Normalization”

**Purpose** Estimate frequency response and spectrum using spectral analysis with frequency-dependent resolution

**Syntax**  
`g = spafdr(data)`  
`g = spafdr(data,Resol,w)`

**Description** `spafdr` estimates the `idfrd` object containing transfer function and the noise spectrum  $\Phi_v$  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  $\omega$  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  $1/N \cdot T_s$  to  $\pi \cdot T_s$ , where  $T_s$  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`.

## Algorithms

If the data is given in the time domain, it is first converted to the frequency domain. Then averages of  $Y(w)\text{Conj}(U(w))$  and  $U(w)\text{Conj}(U(w))$  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` | `freqresp` | `idfrd` | `nyquist` | `spa` | `spectrum`

# spectrum

---

**Purpose** Output power spectrum of time series models

**Syntax**

```
spectrum(sys)
spectrum(sys,{wmin, wmax})
spectrum(sys,w)
spectrum(sys1,...,sysN,w)
ps = spectrum(sys,w)
[ps,w] = spectrum(sys)
[ps,w,sdps] = spectrum(sys)
```

**Description** `spectrum(sys)` creates an output power spectrum plot of the identified time series model `sys`. The frequency range and number of points are chosen automatically.

`sys` is a time series model, which represents the system:

$$y(t) = He(t)$$

Where,  $e(t)$  is a Gaussian white noise and  $y(t)$  is the observed output. `spectrum` plots  $\text{abs}(H'H)$ , scaled by the variance of  $e(t)$  and the sample time.

If `sys` is an input-output model, it represents the system:

$$y(t) = Gu(t) + He(t)$$

Where,  $u(t)$  is the measured input,  $e(t)$  is a Gaussian white noise and  $y(t)$  is the observed output.

In this case, `spectrum` plots the spectrum of the disturbance component  $He(t)$ .

`spectrum(sys,{wmin, wmax})` creates a spectrum plot for frequencies ranging from `wmin` to `wmax`.

`spectrum(sys,w)` creates a spectrum plot using the frequencies specified in the vector `w`.

`spectrum(sys1, ..., sysN, w)` creates a spectrum plot of several identified models on a single plot. The `w` argument is optional.

You can specify a color, line style and marker for each model. For example:

```
spectrum(sys1, 'r', sys2, 'y--', sys3, 'gx');
```

`ps = spectrum(sys, w)` returns the power spectrum amplitude of `sys` for the specified frequencies, `w`. No plot is drawn on the screen.

`[ps, w] = spectrum(sys)` returns the frequency vector, `w`, for which the output power spectrum is plotted.

`[ps, w, sdps] = spectrum(sys)` returns the estimated standard deviations of the power spectrum.

For discrete-time models with sampling time  $T_s$ , `spectrum` uses the transformation  $z = \exp(j\omega T_s)$  to map the unit circle to the real frequency axis. The spectrum is only plotted for frequencies smaller than the Nyquist frequency  $\pi/T_s$ , and the default value 1 (time unit) is assumed when  $T_s$  is unspecified.

## Input Arguments

### **sys**

Identified model.

If `sys` is a time series model, it represents the system:

$$y(t) = He(t)$$

Where,  $e(t)$  is a Gaussian white noise and  $y(t)$  is the observed output.

If `sys` is an input-output model, it represents the system:

$$y(t) = Gu(t) + He(t)$$

Where,  $u(t)$  is the measured input,  $e(t)$  is a Gaussian white noise and  $y(t)$  is the observed output.

### **wmin**

Minimum frequency of the frequency range for which the output power spectrum is plotted.

Specify `wmin` in `rad/TimeUnit`, where `TimeUnit` is `sys.TimeUnit`.

## **wmax**

Maximum frequency of the frequency range for which the output power spectrum is plotted.

Specify `wmax` in `rad/TimeUnit`, where `TimeUnit` is `sys.TimeUnit`.

## **w**

Frequencies for which the output power spectrum is plotted.

Specify `w` in `rad/TimeUnit`, where `TimeUnit` is `sys.TimeUnit`.

## **sys1,...,sysN**

Identified systems for which the output power spectrum is plotted.

## **Output Arguments**

### **ps**

Power spectrum amplitude.

If `sys` has `Ny` outputs, then `ps` is an array of size `[Ny Ny length(w)]`. Where `ps(:, :, k)` corresponds to the power spectrum for the frequency at `w(k)`.

For amplitude values in dB, type `psdb = 10*log10(ps)`.

### **w**

Frequency vector for which the output power spectrum is plotted.

### **sdps**

Estimated standard deviation of the power spectrum.



**Examples****Noise Spectrum of SISO Linear Identified Model**

Plot the noise spectrum of a single-input, single-output linear identified model.

Obtain the identified model.

```
load iddata1 z1;  
sys = n4sid(z1,2);
```

Plot the noise spectrum for the identified model.

```
spectrum(sys);
```

**Output Spectrum of AR Model for 2-Mode Impulse Response**

Plot the output spectrum of an AR model, computed for a 2-mode impulse response of a dynamic system.

Obtain the identified model.

```
load iddata9 z9  
sys = ar(z9,4,'ls');
```

Plot the output spectrum of the identified model.

```
spectrum(sys);
```

**See Also**

bode | freqresp | ar | arx | armax | nlarx | forecast

# spectrumplot

---

**Purpose** Plot disturbance spectrum of linear identified models

**Syntax**

```
spectrumplot(sys)
spectrumplot(sys,line_spec)
spectrumplot(sys1,line_spec1,...,sysN,line_specN)
spectrumplot(ax, ___)
spectrumplot(___,plot_options)
spectrumplot(sys,w)
h = spectrumplot(___)
```

**Description** `spectrumplot(sys)` plots the disturbance spectrum of the model, `sys`. The software chooses the number of points on the plot and the plot frequency range.

If `sys` is a time-series model, its disturbance spectrum is the same as the model output spectrum. You generally use this function with time-series models.

`spectrumplot(sys,line_spec)` uses `line_spec` to specify the line type, marker symbol, and color.

`spectrumplot(sys1,line_spec1,...,sysN,line_specN)` plots the disturbance spectrum for one or more models on the same axes.

You can mix `sys,line_spec` pairs with `sys` models as in `spectrumplot(sys1,sys2,line_spec2,sys3)`. `spectrumplot` automatically chooses colors and line styles in the order specified by the `ColorOrder` and `LineStyleOrder` properties of the current axes.

`spectrumplot(ax, ___)` plots into the axes with handle `ax`. All input arguments described for the previous syntaxes also apply here.

`spectrumplot(___,plot_options)` uses `plot_options` to specify options such as plot title, frequency units, etc. All input arguments described for the previous syntaxes also apply here.

`spectrumplot(sys,w)` uses `w` to specify the plot frequencies.

- If `w` is specified as a 2-element cell array, `{wmin, wmax}`, the plot spans the frequency range `{wmin, wmax}`.

- If `w` is specified as vector, the spectrum is plotted for the specified frequencies.

Specify `w` as `radians/time_unit`, where `time_unit` must equal `sys.TimeUnit`.

`h = spectrumplot( ___ )` returns the handle to the spectrum plot. You use the handle to customize the plot. All input arguments described for the previous syntaxes also apply here.

## Input Arguments

### **sys**

Identified linear model.

### **line\_spec**

Line style, marker, and color of both the line and marker.

Specify as one-, two-, or three-part string. The elements of the string can appear in any order. The string can specify only the line style, the marker, or the color.

For more information, see `Lineseries Properties`.

### **ax**

Plot axes handle.

Specify as a double-precision value.

You can obtain the current axes handle by using the function, `gca`.

### **plot\_options**

Plot customization options.

Specify as a plot options object.

You use the command, `spectrumpoptions`, to create `plot_options`. For more information, type `help spectrumpoptions`.

### **w**

# spectrumplot

---

Frequency range.

Specify in radians/time\_unit, where time\_unit must equal sys.TimeUnit.

## **Output Arguments**

**h**

Plot handle for spectrum plot, returned as a double-precision value.

## **Examples**

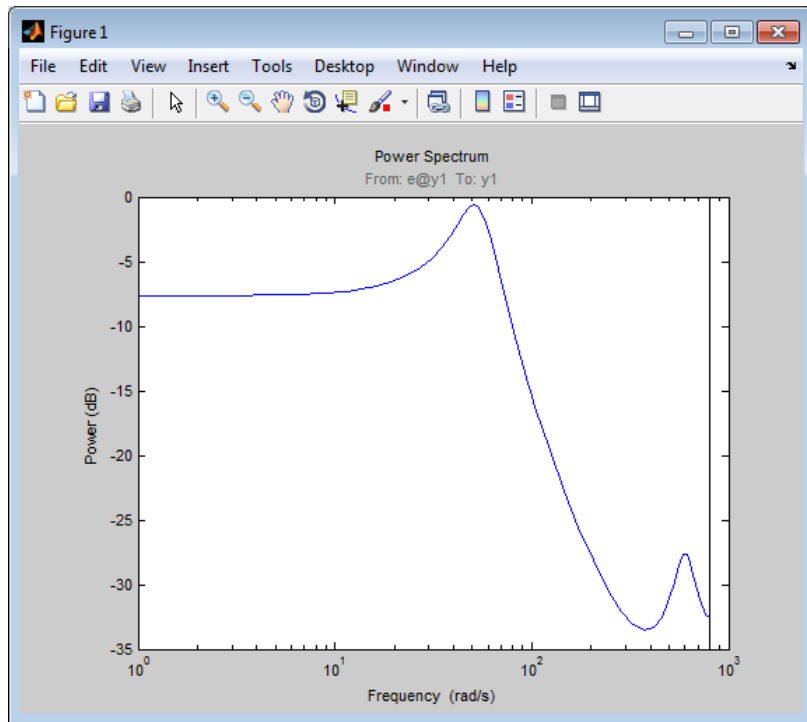
### **Plot Model Output Spectrum for Identified Model**

Obtain the identified model.

```
load iddata9 z9  
sys = ar(z9,4);
```

Plot the output spectrum for the model.

```
spectrumplot(sys);
```



## Specify Line Width and Marker Style on Spectrum Plot

Obtain the identified model.

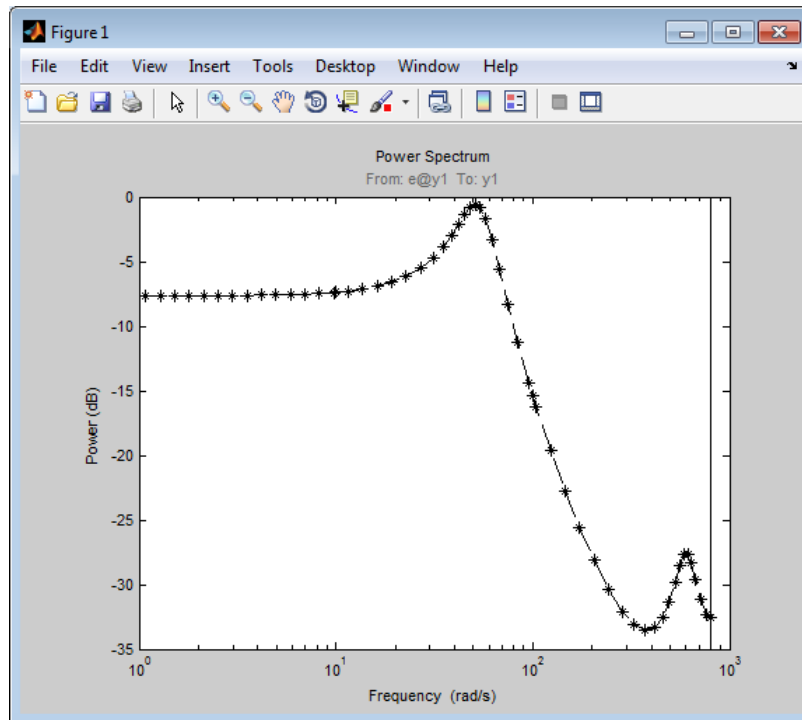
```
load iddata9 z9
sys = ar(z9,4);
```

Specify the line width and marker style for the spectrum plot.

```
spectrumplot(sys, 'k*--');
```

The three-part string, 'k\*--', specifies a dashed line (--). This line is black (k) with star markers (\*).

# spectrumplot



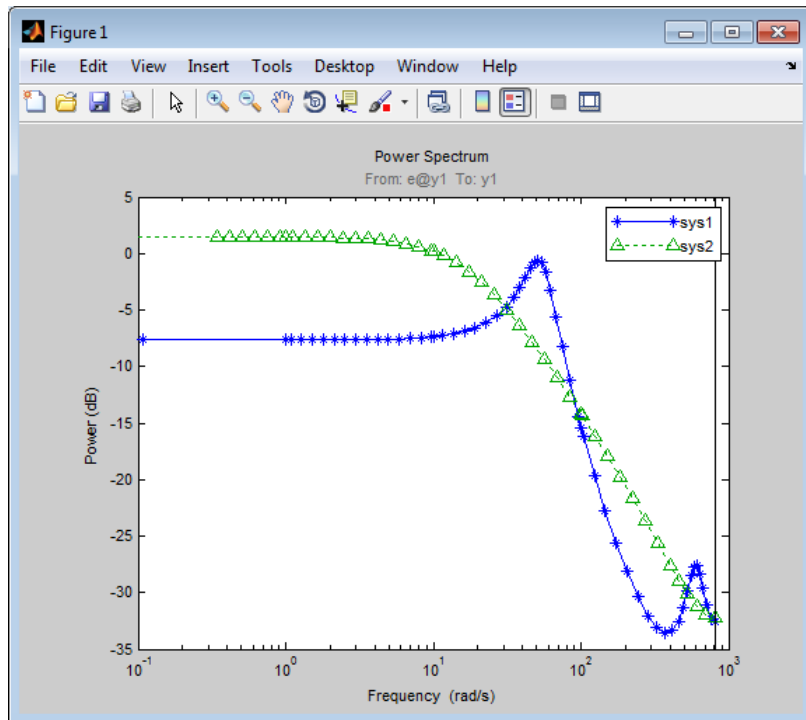
## Plot Multiple Models on the Same Axes

Obtain multiple identified models.

```
load iddata9 z9  
sys1 = ar(z9,4);  
sys2 = ar(z9,2);
```

Plot the output spectrum for both models.

```
spectrumplot(sys1,'b*-',sys2,'g^:');  
legend('sys1','sys2');
```



## Specify Plot Axes for Spectrum Plot

Obtain the axes handle for a plot.

```
load iddata9 z9
sys1 = ar(z9,4);
spectrumpplot(sys1);
ax = gca;
```

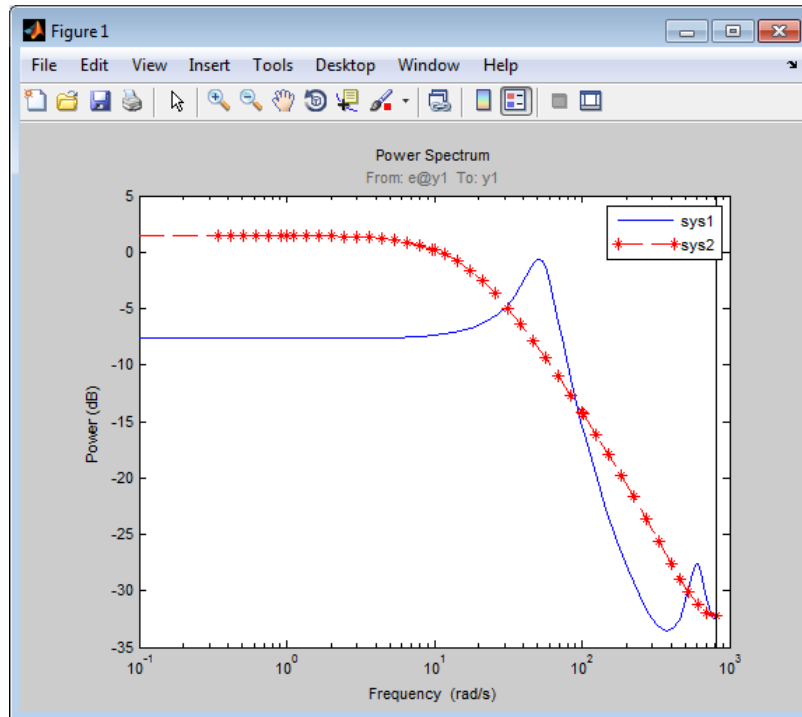
`ax` is the handle for the spectrum plot axes.

Plot the output spectrum for another model on the specified axes.

```
sys2 = ar(z9,2);
```

# spectrumplot

```
hold on;  
spectrumplot(ax,sys2,'r*--');  
  
legend('sys1','sys2');
```



## Specify Plot Options on Spectrum Plot

Obtain the identified model.

```
load iddata9 z9  
sys = ar(z9,4);
```

Specify the plot options.

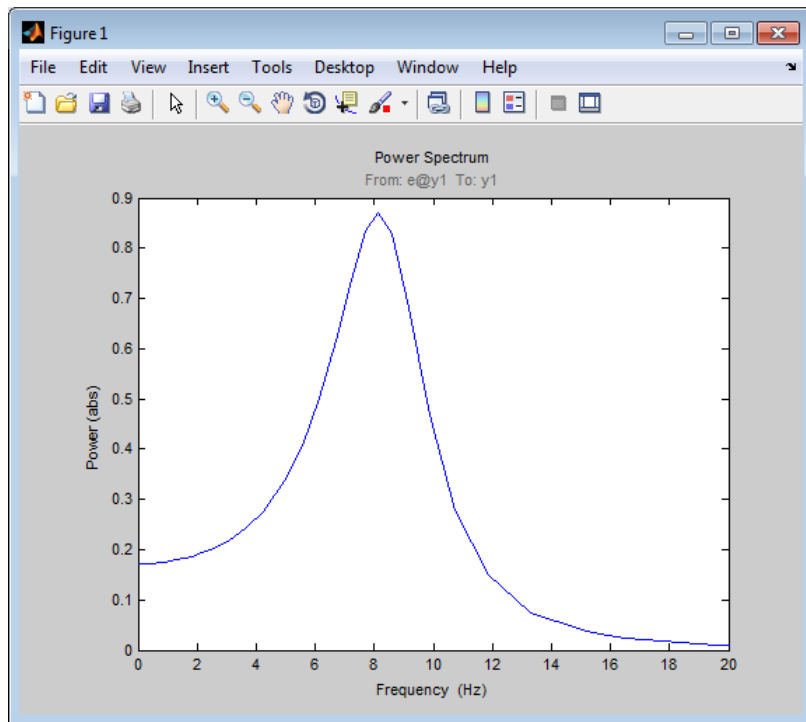
```
plot_options = spectrumoptions;
```



```
plot_options.FreqUnits = 'Hz';  
plot_options.FreqScale = 'linear';  
plot_options.Xlim = {[0 20]};  
plot_options.MagUnits = 'abs';
```

Plot the output spectrum for the model.

```
spectrumplot(sys,plot_options);
```



### Specify Spectrum Plot Frequency Range

Obtain the identified model.

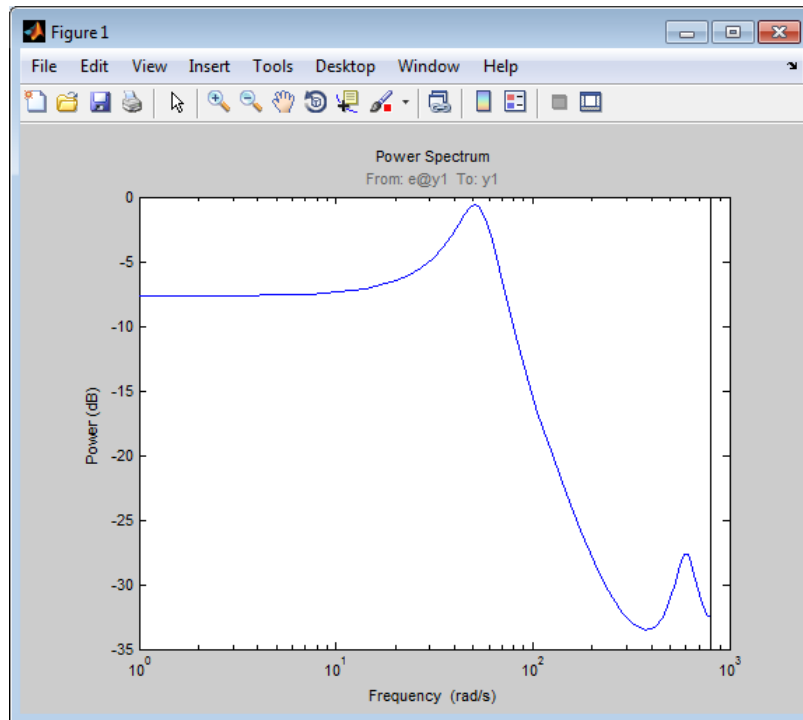
```
load iddata9 z9  
sys = ar(z9,4);
```

# spectrumplot

Specify the frequency range for the output spectrum plot for the model.

```
spectrumplot(sys, {1, 1000});
```

The 2-element cell array {1, 1000} specifies the frequency range from 1 rad/s to 1000 rad/s.



## Get Plot Handle for Spectrum Plot Customization

Obtain the identified model.

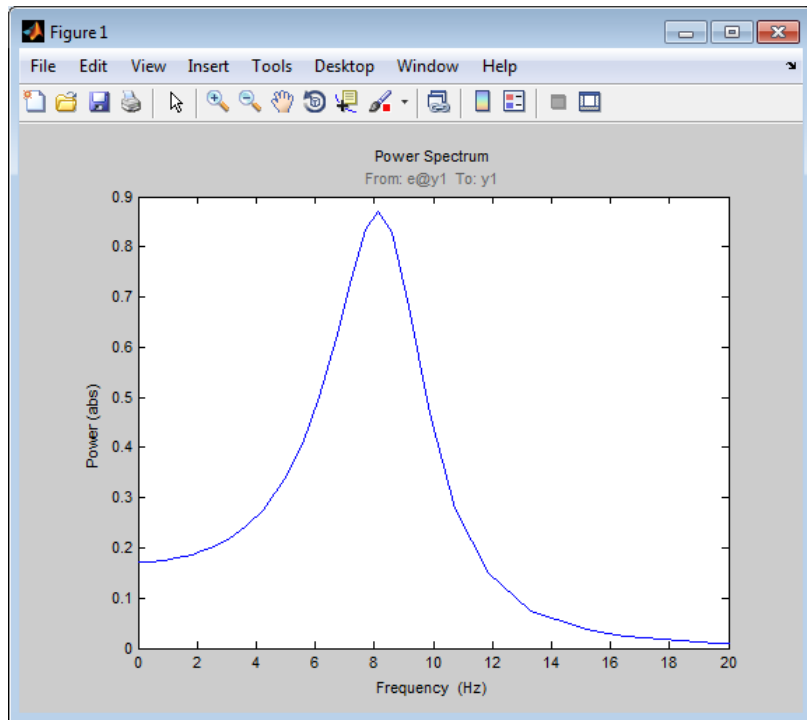
```
load iddata9 z9  
sys = ar(z9,4);
```

Get the plot handle for the model spectrum plot.

```
h = spectrogram(sys);
```

(Optional) Specify the plot options, using the plot handle.

```
setoptions(h, 'FreqUnits', 'Hz', 'FreqScale', 'linear', 'Xlim', {[0 20]}, 'Ma
```



## See Also

[spectrum](#) | [getoptions](#) | [setoptions](#) | [showConfidenceAxes](#)  
[PropertiesLineseries](#) [Properties](#)

**Purpose** State coordinate transformation for state-space model

**Syntax** `sysT = ss2ss(sys,T)`

**Description** Given a state-space model `sys` with equations

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

or the innovations form used by the identified state-space (IDSS) models:

$$\frac{dx}{dt} = Ax + Bu + Ke$$

$$y = Cx + Du + e$$

(or their discrete-time counterpart), `ss2ss` performs the similarity transformation  $\bar{x} = Tx$  on the state vector  $x$  and produces the equivalent state-space model `sysT` with equations.

$$\dot{\bar{x}} = TAT^{-1}\bar{x} + TBu$$

$$y = CT^{-1}\bar{x} + Du$$

or, in the case of an IDSS model:

$$\dot{\bar{x}} = TAT^{-1}\bar{x} + TBu + TKe$$

$$y = CT^{-1}\bar{x} + Du + e$$

`sysT = ss2ss(sys,T)` returns the transformed state-space model `sysT` given `sys` and the state coordinate transformation `T`. The model `sys` must be in state-space form and the matrix `T` must be invertible. `ss2ss` is applicable to both continuous- and discrete-time models.

**Examples** Perform a similarity transform to improve the conditioning of the  $A$  matrix.

```
T = balance(sys.a)
sysb = ss2ss(sys,inv(T))
```

**See Also**

balreal | canon

# ssdata

---

**Purpose** Access state-space model data

**Syntax**  
`[a,b,c,d] = ssdata(sys)`  
`[a,b,c,d,Ts] = ssdata(sys)`

**Description** `[a,b,c,d] = ssdata(sys)` extracts the matrix (or multidimensional array) data A, B, C, D from the state-space model (LTI array) `sys`. If `sys` is a transfer function or zero-pole-gain model (LTI array), it is first converted to state space. See `ss` for more information on the format of state-space model data.

If `sys` appears in descriptor form (nonempty E matrix), an equivalent explicit form is first derived.

If `sys` has internal delays, A, B, C, D are obtained by first setting all internal delays to zero (creating a zero-order Padé approximation). For some systems, setting delays to zero creates singular algebraic loops, which result in either improper or ill-defined, zero-delay approximations. For these systems, `ssdata` cannot display the matrices and returns an error. This error does not imply a problem with the model `sys` itself.

`[a,b,c,d,Ts] = ssdata(sys)` also returns the sample time `Ts`.

You can access the remaining LTI properties of `sys` with `get` or by direct referencing. For example:

```
sys.statename
```

For arrays of state-space models with variable numbers of states, use the syntax:

```
[a,b,c,d] = ssdata(sys, 'cell')
```

to extract the state-space matrices of each model as separate cells in the cell arrays `a`, `b`, `c`, and `d`.

**See Also** `dssdata` | `get` | `getdelaymodel` | `idssdata` | `set` | `ss` | `tfddata` | `zpkdata`

**Purpose**

Estimate state-space model using time or frequency domain data

**Syntax**

```
sys = ssest(data,nx)
sys = ssest(data,nx,Name,Value)
sys = ssest(data,init_sys)
sys = ssest(data,___,opt)
```

**Description**

`sys = ssest(data,nx)` estimates a state-space model, `sys`, using time or frequency domain data, `data`. `sys` is a state-space model of order `nx` and represents:

$$\begin{aligned}\dot{x}(t) &= Ax(t) + Bu(t) + Ke(t) \\ y(t) &= Cx(t) + Du(t) + e(t)\end{aligned}$$

Here,  $A$ ,  $B$ ,  $C$ ,  $D$ , and  $K$  are state-space matrices.  $u(t)$  is the input,  $y(t)$  is the output,  $e(t)$  is the disturbance and  $x(t)$  is the vector of `nx` states. All the entries of  $A$ ,  $B$ ,  $C$ , and  $K$  are free estimation parameters. The elements of the  $D$  matrix, however, are fixed to zero. That is, there is no feedthrough.

`sys = ssest(data,nx,Name,Value)` estimates the model using the additional options specified by one or more `NAME,VALUE` pair arguments.

`sys = ssest(data,init_sys)` estimates a state-space model using the dynamic system `init_sys` to configure the initial parameterization.

`sys = ssest(data,___,opt)` estimates the model using an option set, `opt`.

**Input Arguments****data**

Estimation data.

For time domain estimation, `data` must be an `iddata` object containing the input and output signal values.

For frequency domain estimation, `data` can be one of the following:

- recorded frequency response data (`frd` or `idfrd`)

- `iddata` object with its properties specified as follows:
  - `InputData` — Fourier transform of the input signal
  - `OutputData` — Fourier transform of the output signal
  - `Domain` — 'Frequency'

## **`nx`**

Order of estimated model.

Specify `nx` as a positive integer. `nx` may be a scalar or a vector. If `nx` is a vector, then `ssest` creates a plot which you can use to choose a suitable model order. The plot shows the Hankel singular values for models of different orders. States with relatively small Hankel singular values can be safely discarded. A default choice is suggested in the plot.

## **`opt`**

Estimation options.

`opt` is an options set that specifies, among others, the following:

- estimation objective
- handling of initial conditions
- numerical search method to be used in estimation

Use `ssestOptions` to create the options set.

## **`init_sys`**

Dynamic system that configures the initial parameterization of `sys`.

If `init_sys` is an `idss` model, `ssest` uses the parameter values of `init_sys` as the initial guess for estimating `sys`. Constraints on the parameters of `init_sys`, such as fixed coefficients and minimum/maximum bounds are honored in estimating `sys`.

If `init_sys` is not an `idss` model, the software first converts `init_sys` to an `idss` model. `ssest` uses the parameters of the resulting model as the initial guess for estimation.



Use the `Structure` property of `init_sys` to configure initial guesses and constraints for the  $A$ ,  $B$ ,  $C$ ,  $D$  and  $K$  matrices.

To specify an initial guess for, say, the  $A$  matrix of `init_sys`, set `init_sys.Structure.a.Value` as the initial guess.

To specify constraints for, say, the  $B$  matrix of `init_sys`:

- set `init_sys.Structure.b.Minimum` to the minimum  $B$  matrix value
- set `init_sys.Structure.b.Maximum` to the maximum  $B$  matrix value
- set `init_sys.Structure.b.Free` to indicate if entries of the  $B$  matrix are free parameters for estimation

You can similarly specify the initial guess and constraints for the other matrices.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **Ts**

Sampling time.

For continuous-time models, use `Ts = 0`. For discrete-time models, specify `Ts` as a positive scalar whose value is equal to the data sampling time.

**Default:** 0 (continuous-time)

### **InputDelay**

Input delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. For continuous-time systems, specify input delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify input delays in integer multiples of the sampling period

$T_s$ . For example, `InputDelay = 3` means a delay of three sampling periods.

For a system with  $N_u$  inputs, set `InputDelay` to an  $N_u$ -by-1 vector, where each entry is a numerical value representing the input delay for the corresponding input channel. You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0 for all input channels

### **Form**

Type of canonical form of `sys`.

`Form` is a string that takes one of the following values:

- 'modal' — obtains `sys` in modal form.
- 'companion' — obtain `sys` in companion form.
- 'free' — all entries of the  $A$ ,  $B$  and  $C$  matrices are treated as free.
- 'canonical' — obtain `sys` in the observable canonical form [1].

**Default:** 'free'

### **Feedthrough**

Logical specifying direct feedthrough from input to output.

`Feedthrough` is a logical vector of length  $N_u$ , where  $N_u$  is the number of inputs.

If `Feedthrough` is specified as a logical scalar, it is applied to all the inputs.

### **DisturbanceModel**

Specifies if the noise component, the  $K$  matrix, is to be estimated.

`DisturbanceModel` takes one of the following values:

- 'none' — Noise component is not estimated. The value of the  $K$  matrix, is fixed to zero value.
- 'estimate' — The  $K$  matrix is treated as a free parameter.

`DisturbanceModel` must be 'none' when using frequency domain data.

**Default:** 'estimate' (For time domain data)

## Output Arguments

### **sys**

Identified state space model.

`sys` is an `idss` model, which encapsulates the identified state space model.

### **x0**

Initial states computed during the estimator of `sys`.

If `data` contains multiple experiments, then `x0` is an array with each column corresponding to an experiment.

## Definitions

### **Modal Form**

In modal form,  $A$  is a block-diagonal matrix. The block size is typically 1-by-1 for real eigenvalues and 2-by-2 for complex eigenvalues. However, if there are repeated eigenvalues or clusters of nearby eigenvalues, the block size can be larger.

For example, for a system with eigenvalues  $(\lambda_1, \sigma \pm j\omega, \lambda_2)$ , the modal  $A$  matrix is of the form

$$\begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \sigma & \omega & 0 \\ 0 & -\omega & \sigma & 0 \\ 0 & 0 & 0 & \lambda_2 \end{bmatrix}$$

### Companion Form

In the companion realization, the characteristic polynomial of the system appears explicitly in the rightmost column of the  $A$  matrix. For a system with characteristic polynomial

$$p(s) = s^n + \alpha_1 s^{n-1} + \dots + \alpha_{n-1} s + \alpha_n$$

the corresponding companion  $A$  matrix is

$$A = \begin{bmatrix} 0 & 0 & \dots & \dots & 0 & -\alpha_n \\ 1 & 0 & 0 & \dots & 0 & -\alpha_{n-1} \\ 0 & 1 & 0 & \dots & \vdots & \vdots \\ \vdots & 0 & \dots & \dots & \vdots & \vdots \\ 0 & \dots & \dots & 1 & 0 & -\alpha_2 \\ 0 & \dots & \dots & 0 & 1 & -\alpha_1 \end{bmatrix}$$

The companion transformation requires that the system be controllable from the first input. The companion form is poorly conditioned for most state-space computations; avoid using it when possible.

### Examples

#### Determine Optimal Estimated Model Order

Estimate a state-space model for measured input-output data.  
Determine the optimal model order within a given range.

Obtain measured input-output data.

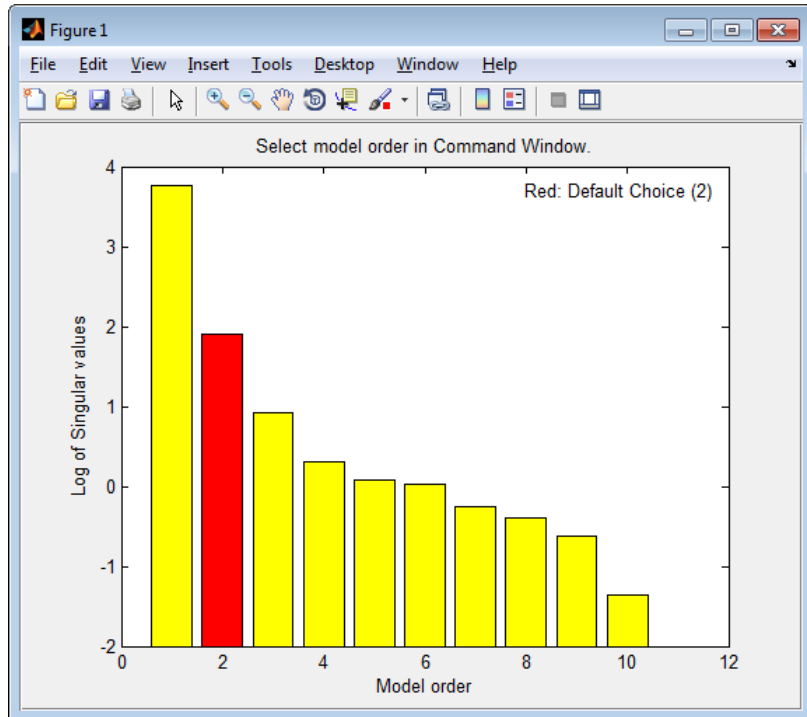
```
load icEngine.mat;
data = iddata(y,u,0.04);
```

`data` is an `iddata` object containing 1500 input-output data samples.  
The data sampling time is 0.04 seconds.

Estimate a state-space model for measured input-output data.  
Determine the optimal model order within a given model order range.

```
nx = 1:10;  
sys = ssest(data,nx);
```

A plot appears:



The plot shows the Hankel singular values (SVD) for models of the orders specified by `nx`. States with relatively small Hankel singular values can be safely discarded. The default order choice is 2.

At the MATLAB command prompt, you are prompted to select the model order for the estimated state-space model. Specify the model order or press **Enter** to use the default order value.

### **Identify State-Space Model With Input Delay**

Identify a state-space model containing an input delay for given data.

Load time domain system response data and use it to identify a state-space model for the system. Specify a known input delay for the model.

```
load iddata7 z7
nx = 4;
sys = ssest(z7(1:300),nx, 'InputDelay', [2;0])
```

`z7` is an `iddata` object that contains time domain system response data.

`nx` specifies a fourth order identified state-space model.

The name-value input argument pair `'InputDelay', [2;0]` specifies an input delay of 2 seconds for the first input and 0 seconds for the second output.

`sys` is an `idss` model containing the identified state-space model.

### **Estimate State-Space Model for Partially Known Model (Structured Estimation)**

Estimate a state-space model using measured input-output data. Configure the parameter constraints and initial values for estimation using a state-space model.

Create an `idss` model to specify the initial parameterization for estimation.

For this example, configure an `idss` model so that it has no state-disturbance element and only the non-zero entries of the  $A$  matrix are estimable. Additionally, fix the values of the  $B$  matrix.

```
A = blkdiag([-0.1 0.4; -0.4 -0.1],[-1 5; -5 -1]);
B = [1; zeros(3,1)];
C = [1 1 1 1];
D = 0;
K = zeros(4,1);
```

```
x0 = [0.1,0.1,0.1,0.1];  
Ts = 0;  
init_sys = idss(A,B,C,D,K,x0,Ts);
```

Setting all entries of  $K = 0$  creates an `idss` model with no state disturbance element.

Use the `Structure` property of `init_sys` to fix the values of some of the parameters.

```
init_sys.Structure.a.Free = (A~=0);  
init_sys.Structure.b.Free = false;  
init_sys.Structure.k.Free = false;
```

The entries in `init_sys.Structure.a.Free` determine whether the corresponding entries in `init_sys.a` are free (identifiable) or fixed. The first line sets `init_sys.Structure.a.Free` to a matrix that is true wherever `A` is non-zero, and false everywhere else. Doing so fixes the value of the zero entries in `init_sys.a`.

The remaining lines fix all the values in `init_sys.b` and `init_sys.k` to the values you specified when you created the model.

Load the measured data and estimate a state-space model using the parameter constraints and initial values specified by `init_sys`.

```
load iddata2 z2;  
sys = ssest(z2,init_sys);
```

`sys` is an `idss` model that encapsulates the fourth-order state-space model estimated for the measured data `z2`. The estimated parameters of `sys` successfully satisfy the constraints specified by `init_sys`.

### Model Order Reduction by Estimation

This example shows how to reduce the order of a model by estimation.

For this example, consider the Simulink® model `idF14Model`. Linearizing this model gives a 9th order model. However, the dynamics

of the model can be captured, without compromising the fit quality too much, using a lower order model.

Obtain the linearized model.

```
open_system('idF14Model');  
io = getlinio('idF14Model');  
sys_lin = linearize('idF14Model',io);
```

`sys_lin` is a 9th order state-space model with two outputs and one input. It represents the linearization of the Simulink model `idF14Model`.

Simulate the step response of the linearized model and create an `iddata` object.

```
Ts = 0.0444;  
t = (0:Ts:4.44)';  
y = step(sys_lin,t);
```

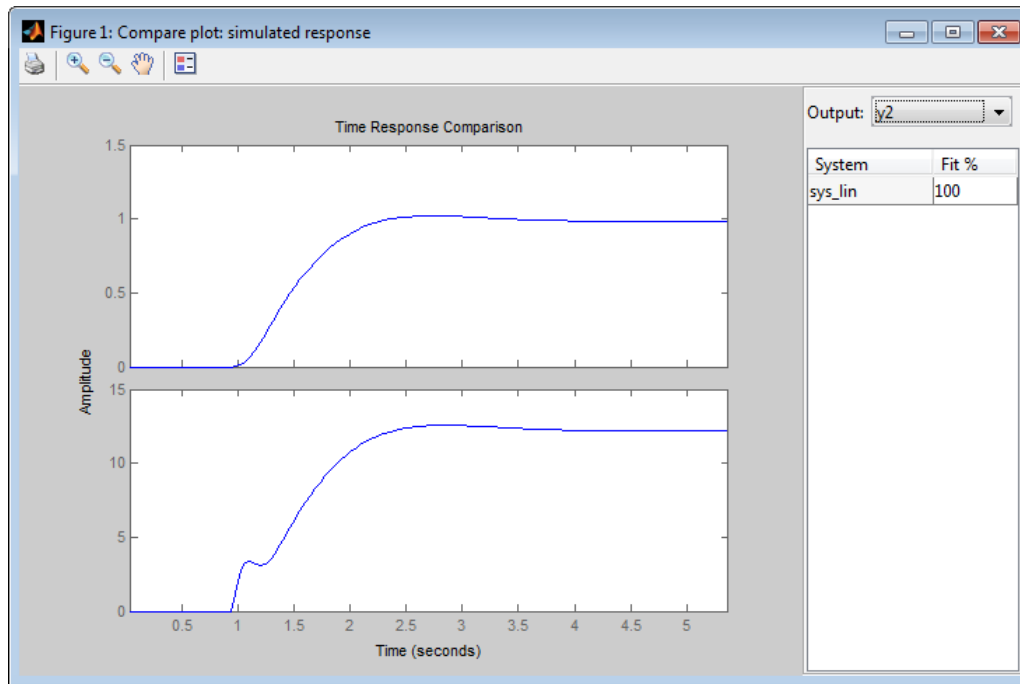
```
data = iddata([zeros(20,2);y],[zeros(20,1); ones(101,1)],Ts);
```

`data` is an `iddata` object that encapsulates the step response of `sys_lin`.

Compare the data to the model linearization.

```
compare(data, sys_lin)
```





Given that the data was obtained by simulating the linearized model, there is a 100% match between the data and model linearization response.

Identify a state-space model with a reduced order that adequately fits the data.

```
nx = 1:9;
sys1 = ssest(data,nx,'DisturbanceModel','none');
```

As the model order input for ssest, nx, is specified as a vector, a plot appears. The plot shows the Hankel singular values (SVD) for models of the orders specified by nx. States with relatively small Hankel singular values can be safely discarded. The plot suggests using a fifth-order model. At the MATLAB command prompt, you are prompted to select

the model order for the estimated state-space model. Specify the model order as 5, or press **Enter** to use the default order value.

`sys1` provides a 98.6% fit for the first output and a 97.2% fit for the second output.

Examine the stopping condition for the search algorithm.

```
sys1.Report.Termination.WhyStop
```

```
ans =
```

```
Maximum number of iterations reached
```

Create an estimation option set that uses the 'lm' search method and allows 50 search iterations at most.

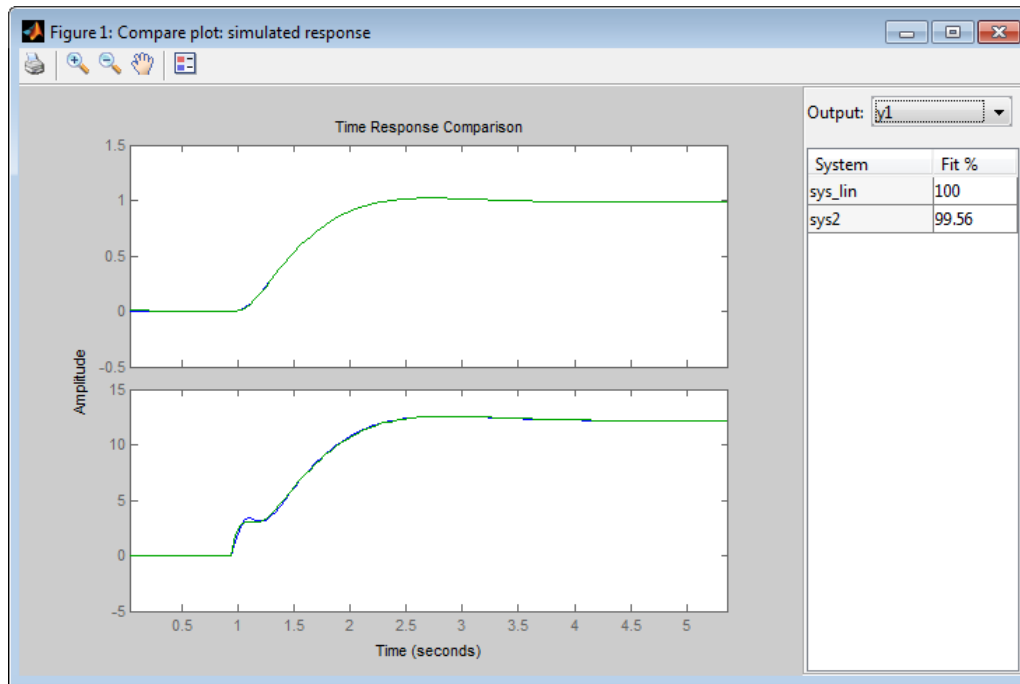
```
opt = ssestOptions('SearchMethod','lm');  
opt.SearchOption.MaxIter = 50;  
opt.Display = 'on';
```

Identify a state-space model using the estimation option set and `sys1` as the estimation initialization model.

```
sys2 = ssest(data, sys1, opt);
```

Compare the response of the linearized and the estimated models.

```
compare(data, sys_lin, sys2);
```



sys2 provides a 99.93% fit for the first output and a 98% fit for the second output while using 4 less states than sys\_lin.

## Algorithms

ssest initializes the parameter estimates using a non-iterative subspace approach. It then refines the parameter values for prediction error minimization using the PEM approach. See pem for more information.

## References

[1] Ljung, L. *System Identification — Theory For the User*, Appendix 4A, pp 132-134, 2nd ed, PTR Prentice Hall, Upper Saddle River, N.J., 1999.

## See Also

ssestOptions | idss | n4sid | tfest | procest | polyst | iddata | idfrd | canon | idgrey | pem

# ssestOptions

---

**Purpose** Option set for ssest

**Syntax**  
`opt = ssestOptions`  
`opt = ssestOptions(Name,Value)`

**Description** `opt = ssestOptions` creates the default options set for `ssest`.  
`opt = ssestOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### InitialState

Specify handling of initial states during estimation.

`InitialState` takes one of the following values:

- 'zero' — The initial state is set to zero.
- 'estimate' — The initial state is treated as an independent estimation parameter.
- 'backcast' — The initial state is estimated using the best least squares fit.
- 'auto' — `ssest` chooses the initial state handling method based on the estimation data. The possible initial state handling methods are 'zero', 'estimate' and 'backcast'.
- vector of doubles — Specify a column vector of length  $Nx$ , where  $Nx$  is the number of states. For multi-experiment data, specify a matrix with  $Ne$  columns, where  $Ne$  is the number of experiments. The specified values are treated as fixed values during the estimation process.

- a parametric initial condition object (`x0obj`) — Specify initial conditions by using `x0par` to create a parametric initial condition object. You can specify minimum/maximum bounds and fix the values of specific states using the parametric initial condition object. The free entries of `x0obj` are estimated together with the `idss` model parameters.

Use this option only for discrete-time state-space models.

**Default:** 'auto'

## **N4Weight**

Weighting scheme used for singular-value decomposition by the N4SID algorithm.

'N4Weight' takes one of the following values:

- 'MOESP' — Uses the MOESP algorithm by Verhaegen [2].
- 'CVA' — Uses the Canonical Variable Algorithm by Larimore [1].
- 'auto' — The estimating function chooses between the MOESP and CVA algorithms.

**Default:** 'auto'

## **N4Horizon**

Forward and backward prediction horizons used by the N4SID algorithm.

'N4Horizon' takes one of the following values:

- A row vector with three elements —  $[r \text{ sy } \text{su}]$ , where  $r$  is the maximum forward prediction horizon. That is, the algorithm uses up to  $r$  step-ahead predictors.  $\text{sy}$  is the number of past outputs, and  $\text{su}$  is the number of past inputs that are used for the predictions. See pages 209 and 210 in [4] 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. k is the number of guesses of [r sy su] combinations. If you specify N4Horizon as a single column, r = sy = su is used.

- 'auto' — The software uses an Akaike Information Criterion (AIC) for the selection of sy and su.

**Default:** auto

## **Focus**

Defines how the errors  $e$  between the measured and the modeled outputs are weighed at specific frequencies during the minimization of the prediction error.

Higher weighting at specific frequencies emphasizes the requirement for a good fit at these frequencies.

Focus can take the following values:

- '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.

This method provides a stable model.

- 'prediction' — 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. It might not result in a stable model. Use 'stability' when you want to ensure a stable model.
- 'stability' — Same as 'prediction' but with model stability enforced.

- Passbands — Row vector or matrix containing frequency values that define desired passbands. For example:

```
[w1,wh]
[w11,w1h;w21,w2h;w31,w3h;...]
```

where `w1` 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.

- SISO filter — Enter any SISO linear filter in any of the following ways:
  - A single-input-single-output (SISO) linear system.
  - The {A,B,C,D} format, which specifies the state-space matrices of the filter.
  - The {numerator, denominator} format, 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. To obtain a good model fit for a specific frequency range, you must choose the filter with a passband in this range. The estimation result is the same if you first prefilter the data using `idfilt`.

- Weighting vector — 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.

**Default:** 'prediction'

## **EstCovar**

Controls whether parameter covariance data is generated or not.

If `EstCovar` is `true`, then use `getcov` to fetch the covariance matrix from the estimated model.

**Default:** `true`

## Display

Specifies whether estimation progress should be displayed.

`Display` requires one of the following strings:

- `'on'` — Information on model structure and estimation results are displayed in a progress viewer window
- `'off'` — No progress or results information is displayed

**Default:** `'off'`

## InputOffset

Removes offset from time domain input data during estimation.

Specify as a column vector of length  $Nu$ , where  $Nu$  is the number of inputs.

Use `[]` to indicate no offset.

For multiexperiment data, specify `InputOffset` as a  $Nu$ -by- $Ne$  matrix.  $Nu$  is the number of inputs, and  $Ne$  is the number of experiments.

Each entry specified by `InputOffset` is subtracted from the corresponding input data.

**Default:** `[]`

## OutputOffset

Removes offset from time domain output data during estimation.

Specify as a column vector of length  $Ny$ , where  $Ny$  is the number of outputs.

Use `[]` to indicate no offset.



For multiexperiment data, specify `OutputOffset` as a  $N_y$ -by- $N_e$  matrix.  $N_y$  is the number of outputs, and  $N_e$  is the number of experiments. Each entry specified by `OutputOffset` is subtracted from the corresponding output data.

**Default:** []

### OutputWeight

Specifies criterion used during minimization.

`OutputWeight` can have the following values:

- 'noise' — 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.

---

**Note** `OutputWeight` must not be 'noise' if `SearchMethod` is 'lsqnonlin'.

---

- positive semi-definite symmetric matrix ( $W$ ) — Minimize the trace of the weighted prediction error matrix  $\text{trace}(E^*E*W)$ , where  $E$  is the matrix of prediction errors, with one column for each output, and  $W$  is the positive semi-definite symmetric matrix of size equal to the number of outputs. Use  $W$  to specify the relative importance of outputs in multiple-input multiple-output models, or the reliability of corresponding data.

This option is relevant only for multi-input, multi-output models.

- [] — The software chooses between the 'noise' or using the identity matrix for  $W$ .

**Default:** []

## SearchMethod

Search method used for iterative parameter estimation.

SearchMethod is a string that can take the following values:

- **gn** — The subspace Gauss-Newton direction. Singular values of the Jacobian matrix less than  $\text{GnPinvConst} \cdot \text{eps} \cdot \max(\text{size}(J)) \cdot \text{norm}(J)$  are discarded when computing the search direction.  $J$  is the Jacobian matrix. The Hessian matrix is approximated by  $J^T J$ . If there is no improvement in this direction, the function tries the gradient direction.
- **gna** — An adaptive version of subspace Gauss-Newton approach, suggested by Wills and Ninness [3]. Eigenvalues less than  $\text{gamma} \cdot \max(sv)$  of the Hessian are neglected, where  $sv$  are the singular values of the Hessian. The Gauss-Newton direction is computed in the remaining subspace.  $\text{gamma}$  has the initial value `InitGnaTol` (see `Advanced` for more information) 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  $2 \cdot \text{LMStep}$  each time a search is successful without any bisections.
- **lm** — Uses the Levenberg-Marquardt method. This means that the next parameter value is  $-\text{pinv}(H+d \cdot I) \cdot \text{grad}$  from the previous one, where  $H$  is the Hessian,  $I$  is the identity matrix, and  $\text{grad}$  is the gradient.  $d$  is a number that is increased until a lower value of the criterion is found.
- **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.
- **grad** — The steepest descent gradient search method.
- **auto** — A choice among the above is made in the algorithm. The descent direction is calculated using **gn**, **gna**, **lm** and **grad** successively, in that order, at each iteration until a sufficient reduction in error is achieved.

**Default:** 'auto'

## SearchOption

SearchOption is an options set for the search algorithm with the fields:

SearchMethod	SearchOption
gn, gna, lm, grad and auto	<ul style="list-style-type: none"> <li>• <b>Tolerance</b> — 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. 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.  <b>Default:</b> 0.01</li> <li>• <b>MaxIter</b> — Maximum number of iterations during loss-function minimization. The iterations stop when MaxIter is reached or another stopping criterion is satisfied, such as Tolerance.  Setting MaxIter = 0 returns the result of the startup procedure.  Use <code>sys.Report.Termination.Iterations</code> to get the actual number of iterations during an estimation, where <code>sys</code> is an <code>idtf</code> model.  <b>Default:</b> 20</li> <li>• <b>Advanced</b> — Search settings:             <ul style="list-style-type: none"> <li>▪ <b>GnPinvConst</b> — Singular values of the Jacobian that are smaller than <code>GnPinvConst*max(size(J)*norm(J)*eps)</code> are discarded when computing the search direction and SearchMethod is 'gn'.  GnPinvConst must be a positive real value.  <b>Default:</b> 10000</li> </ul> </li> </ul>

SearchMethod	SearchOption
	<ul style="list-style-type: none"><li>▪ <b>InitGnaTol</b> — Initial value of <i>gamma</i> when SearchMethod is 'gna'. <b>Default:</b> .0001</li><li>▪ <b>LMStartValue</b> — Starting value of search-direction length <i>d</i> in the Levenberg-Marquardt method. Applicable when SearchMethod is 'lm'. <b>Default:</b> .001</li><li>▪ <b>LMStep</b> — Size of the Levenberg-Marquardt step. The next value of the search-direction length <i>d</i> in the Levenberg-Marquardt method is LMStep times the previous one. Applicable when SearchMethod is 'lm'. <b>Default:</b> 2</li><li>▪ <b>MaxBisections</b> — Maximum number of bisections used by the line search along the search direction. <b>Default:</b> 25</li><li>▪ <b>MaxFunEvals</b> — Iterations are stopped if the number of calls to the model file exceeds this value. MaxFunEvals must be a positive integer value. <b>Default:</b> Inf</li><li>▪ <b>MinParChange</b> — Smallest parameter update allowed per iteration MinParChange must be a positive, real value. <b>Default:</b> 0</li><li>▪ <b>RelImprovement</b> — Iterations are stopped if the relative improvement of the criterion function is less than RelImprovement.</li></ul>

SearchMethod	SearchOption
	<p>RelImprovement must be a positive integer value.</p> <p><b>Default:</b> 0</p> <ul style="list-style-type: none"> <li>StepReduction — 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.</li> </ul> <p>StepReduction must be a positive, real value that is greater than 1.</p> <p><b>Default:</b> 2</p>
lsqnonlin	<ul style="list-style-type: none"> <li>TolFun — Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values.</li> </ul> <p>The value of TolFun is the same as that of <code>sys.SearchOption.Advanced.TolFun</code>.</p> <p><b>Default:</b> 1e-5</p> <ul style="list-style-type: none"> <li>TolX — Termination tolerance on the estimated parameter values.</li> </ul> <p>The value of TolX is the same as that of <code>sys.SearchOption.Advanced.TolX</code>.</p> <p><b>Default:</b> 1e-6</p> <ul style="list-style-type: none"> <li>MaxIter — Maximum number of iterations during loss-function minimization. The iterations stop when MaxIter is reached or another stopping criterion is satisfied, such as TolFun etc..</li> </ul> <p>The value of MaxIter is the same as that of <code>sys.SearchOption.Advanced.MaxIter</code>.</p> <p><b>Default:</b> 20</p> <ul style="list-style-type: none"> <li>Advanced — Options set for lsqnonlin.</li> </ul>

Search Method	Search Option
	<p>For more information, see “Optimization Options”.</p> <p>Use <code>optimset('lsqnonlin')</code> to create an options set for <code>lsqnonlin</code> and then modify it to specify its various options.</p>

## Advanced

Advanced is a structure with the following fields:

- **ErrorThreshold** — Specifies when to adjust the weight of large errors from quadratic to linear.

Errors larger than **ErrorThreshold** times the estimated standard deviation have a linear weight in the 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 the chapter “Computing the Estimate” of [4].) **ErrorThreshold = 0** disables robustification and leads to a purely quadratic criterion. When estimating with frequency-domain data, **ErrorThreshold** is set to zero.

**Default:** 0

- **MaxSize** — Specifies the maximum number of elements in a segment when input-output data is split into segments.

**MaxSize** must be a positive integer.

**Default:** 250000

- **StabilityThreshold** — Specifies thresholds for stability tests.

**StabilityThreshold** is a structure with the following fields:

- **s** — Specifies the location of the right-most pole to test the stability of continuous-time models. A model is considered stable when its right-most pole is to the left of **s**.

**Default:** 0

- `z` — 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 `z` from the origin.

**Default:** `1+sqrt(eps)`

- `AutoInitThreshold` — Specifies when to automatically estimate the initial conditions.

When `InitialCondition = 'Auto'`, the initial condition is estimated when the ratio of the prediction-error norm with a zero initial condition to the norm with an estimated initial condition exceeds `AutoInitialState`.

**Default:** `1.05`

- `DDC` — Specifies if the Data Driven Coordinates algorithm [5] is used to estimate freely parameterized state-space models.

Specify `DDC` as one of the following values:

- `'on'` — The free parameters are projected to a reduced space of identifiable parameters using the Data Driven Coordinates algorithm.
- `'off'` — All the entries of  $A$ ,  $B$ , and  $C$  updated directly using the chosen `SearchMethod`.

**Default:** `'on'`

## Output Arguments

**opt**

Option set containing the specified options for `ssest`.

## Examples

### Create Default Options Set for State Space Estimation

Create a default options set for `ssest`.

```
opt = ssestOptions;
```

## Specify Options for State Space Estimation

Create an options set for `ssest` using the 'backcast' algorithm to initialize the state and set the Display to 'on'.

```
opt = ssestOptions('InitialState','backcast','Display','on');
```

Alternatively, use dot notation to set the values of `opt`.

```
opt = ssestOptions;  
opt.InitialState = 'backcast';  
opt.Display = 'on';
```

## References

- [1] Larimore, W.E. "Canonical variate analysis in identification, filtering and adaptive control." *Proceedings of the 29th IEEE Conference on Decision and Control*, pp. 596–604, 1990.
- [2] Verhaegen, M. "Identification of the deterministic part of MIMO state space models." *Automatica*, Vol. 30, No. 1, 1994, pp. 61–74.
- [3] Wills, Adrian, B. Ninness, and S. Gibson. "On Gradient-Based Search for Multivariable System Estimates". *Proceedings of the 16th IFAC World Congress, Prague, Czech Republic, July 3–8, 2005*. Oxford, UK: Elsevier Ltd., 2005.
- [4] Ljung, L. *System Identification: Theory for the User*. Upper Saddle River, NJ: Prentice-Hall PTR, 1999.
- [5] McKelvey, T., A. Helmersson, and T. Ribarits. "Data driven local coordinates for multivariable linear systems and their application to system identification." *Automatica*, Volume 40, No. 9, 2004, pp. 1629–1635.

**See Also** `ssest`



**Purpose** Quick configuration of state-space model structure

**Syntax** `sys1 = ssform(sys,Name,Value)`

**Description** `sys1 = ssform(sys,Name,Value)` specifies the type of parameterization and whether feedthrough and disturbance dynamics are present for the state-space model `sys` using one or more `Name,Value` pair arguments.

**Input Arguments** **sys**  
State-space model

### Name-Value Pair Arguments

Specify comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### Form

Specify structure of A, B and C matrices. Must be one of the following strings:

- 'free'  
All entries of A, B, C are set free
- 'companion'  
Companion form of the model where the characteristic polynomial appears in the far-right column of the state matrix A
- 'modal'  
Modal decomposition form, where the state matrix A is block diagonal. Each block corresponds to a real or complex-conjugate pair of poles.  
You cannot use this value for models with repeated poles.

- 'canonical'

Observability canonical form of A, B, and C matrices, as described in [1].

## Feedthrough

Specify whether the model has direct feedthrough from the input  $u(t)$  to the output  $y(t)$ , (whether the elements of the matrix D are nonzero).

Must be a logical vector (true or false) of length equal to the number of inputs ( $Nu$ ).

Feedthrough(i) = false sets sys.Structure.d.Value(:,i) to zero and sys.Structure.d.Free(:,i) to false.

Feedthrough(i) = true sets sys.Structure.d.Free(:,i) to true.

---

**Note** Specifying this option for a previously estimated model causes the model parameter covariance information to be lost. Use `translatecov` to recompute the covariance.

---

## DisturbanceModel

Specify whether to estimate the noise component of the model. Must be one of the following strings:

- 'none'

The value of the K matrix is fixed to zero.

- 'estimate'

The K matrix is treated as a free parameter

---

**Note** Specifying this option for a previously estimated model causes the model parameter covariance information to be lost. Use `translatecov` to recompute the covariance.

---

**Output Arguments****sys1**

State-space model with configured parameterization, feedthrough, and disturbance dynamics

**Examples****Convert a State-Space Model to Canonical Form**

Create a state-space model.

```
rng('default');  
A = randn(2)-2*eye(2);  
B = randn(2,1);  
C = randn(1,2);  
D = 0;  
K = randn(2,1);  
model = idss(A,B,C,D,K,'Ts',0);
```

The state-space model has free parameterization and no feedthrough.

Convert the model to observability canonical form.

```
model1 = ssform(model, 'Form', 'canonical');
```

**Estimate State-Space Model Parameters in Canonical Form with Feedthrough**

Load the estimation data.

```
load iddata1 z1;
```

Create a state-space model.

```
rng('default');  
A = randn(2)-2*eye(2);  
B = randn(2,1);  
C = randn(1,2);  
D = 0;  
K = randn(2,1);  
model = idss(A,B,C,D,K,'Ts',0);
```

The state-space model has free parameterization and no feedthrough.

Convert the model to observability canonical form and specify to estimate its feedthrough behavior.

```
model1 = ssform(model, 'Form', 'canonical', 'Feedthrough', true);
```

Estimate the parameters of the model.

```
model2 = ssest(z1, model1);
```

## References

[1] Ljung, L. *System Identification: Theory For the User*, Second Edition, Appendix 4A, pp 132-134, Upper Saddle River, N.J: Prentice Hall, 1999.

## Alternatives

Use the `Structure` property of an `idss` model to specify the parameterization, feedthrough, and disturbance dynamics by modifying the `Value` and `Free` attributes of the A, B, C, D and K parameters.

## See Also

`idss` | `ssest` | `n4sid`

## Related Examples

- “How to Estimate State-Space Models at the Command Line”

## Concepts

- “Supported State-Space Parameterizations”

**Purpose**

Build model array by stacking models or model arrays along array dimensions

**Syntax**

```
sys = stack(arraydim,sys1,sys2,...)
```

**Description**

`sys = stack(arraydim,sys1,sys2,...)` produces an array of dynamic system models `sys` by stacking (concatenating) the models (or arrays) `sys1,sys2,...` along the array dimension `arraydim`. All models must have the same number of inputs and outputs (the same I/O dimensions), but the number of states can vary. The I/O dimensions are not counted in the array dimensions. For more information about model arrays and array dimensions, see “Model Arrays”.

For arrays of state-space models with variable order, you cannot use the dot operator (e.g., `sys.a`) to access arrays. Use the syntax

```
[a,b,c,d] = ssdata(sys,'cell')
```

to extract the state-space matrices of each model as separate cells in the cell arrays `a`, `b`, `c`, and `d`.

**Examples****Example 1**

If `sys1` and `sys2` are two models:

- `stack(1,sys1,sys2)` produces a 2-by-1 model array.
- `stack(2,sys1,sys2)` produces a 1-by-2 model array.
- `stack(3,sys1,sys2)` produces a 1-by-1-by-2 model array.

**Example 2**

Stack identified state-space models derived from the same estimation data and compare their bode responses.

```
load iddata1 z1
sysc = cell(1,5);
opt = ssestOptions('Focus','simulation');
for i = 1:5
```

# stack

---

```
sysc{i} = ssest(z1,i-1,opt);  
end  
sysArray = stack(1, sysc{:});  
bode(sysArray);
```

**Purpose**

Step response plot of dynamic system

**Syntax**

```
step(sys)
step(sys,Tfinal)
step(sys,t)
step(sys1,sys2,...,sysN)
step(sys1,sys2,...,sysN,Tfinal)
step(sys1,sys2,...,sysN,t)
y = step(sys,t)
[y,t] = step(sys)
[y,t] = step(sys,Tfinal)
[y,t,x] = step(sys)
[y,t,x,ysd] = step(sys)
[y,...] = step(sys,...,options)
```

**Description**

step calculates the step response of a dynamic system. For the state space case, zero initial state is assumed. When it is invoked with no output arguments, this function plots the step response on the screen.

step(sys) plots the step response of an arbitrary dynamic system model sys. This model can be continuous or discrete, and SISO or MIMO. The step response of multi-input systems is the collection of step responses for each input channel. The duration of simulation is determined automatically, based on the system poles and zeros.

step(sys,Tfinal) simulates the step response from  $t = 0$  to the final time  $t = T_{\text{final}}$ . Express  $T_{\text{final}}$  in the system time units, specified in the TimeUnit property of sys. For discrete-time systems with unspecified sampling time ( $T_s = -1$ ), step interprets  $T_{\text{final}}$  as the number of sampling periods to simulate.

step(sys,t) uses the user-supplied time vector t for simulation. Express t in the system time units, specified in the TimeUnit property of sys. For discrete-time models, t should be of the form  $T_i:T_s:T_f$ , where  $T_s$  is the sample time. For continuous-time models, t should be of the form  $T_i:dt:T_f$ , where dt becomes the sample time of a discrete approximation to the continuous system (see “Algorithms” on

page 2-907). The `step` command always applies the step input at  $t=0$ , regardless of  $T_i$ .

To plot the step response of several models `sys1, ..., sysN` on a single figure, use

```
step(sys1,sys2,...,sysN)
step(sys1,sys2,...,sysN,Tfinal)
step(sys1,sys2,...,sysN,t)
```

All of the systems plotted on a single plot must have the same number of inputs and outputs. You can, however, plot a mix of continuous- and discrete-time systems on a single plot. This syntax is useful to compare the step responses of multiple systems.

You can also specify a distinctive color, linestyle, marker, or all three for each system. For example,

```
step(sys1,'y: ',sys2,'g--')
```

plots the step response of `sys1` with a dotted yellow line and the step response of `sys2` with a green dashed line.

When invoked with output arguments:

```
y = step(sys,t)
[y,t] = step(sys)
[y,t] = step(sys,Tfinal)
[y,t,x] = step(sys)
```

`step` returns the output response `y`, the time vector `t` used for simulation (if not supplied as an input argument), and the state trajectories `x` (for state-space models only). No plot generates on the screen. For single-input systems, `y` has as many rows as time samples (length of `t`), and as many columns as outputs. In the multi-input case, the step responses of each input channel are stacked up along the third dimension of `y`. The dimensions of `y` are then



*(length of t) × (number of outputs) × (number of inputs)*

and  $y(:, :, j)$  gives the response to a unit step command injected in the  $j$ th input channel. Similarly, the dimensions of  $x$  are

*(length of t) × (number of states) × (number of inputs)*

For identified models (see `idlti` and `idn1model1`) `[y,t,x,yzd] = step(sys)` also computes the standard deviation `ysd` of the response `y` (`ysd` is empty if `sys` does not contain parameter covariance information).

`[y,...] = step(sys,...,options)` specifies additional options for computing the step response, such as the step amplitude or input offset. Use `stepDataOptions` to create the option set `options`.

## Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

## Examples

### Example 1

#### Step Response Plot of Dynamic System

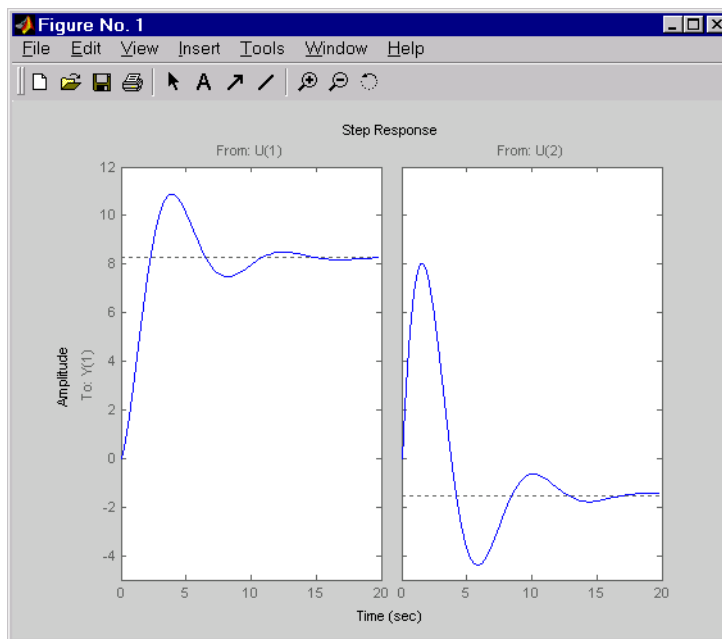
Plot the step response of the following second-order state-space model.

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -0.5572 & -0.7814 \\ 0.7814 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

$$y = [1.9691 \quad 6.4493] \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

```
a = [-0.5572 -0.7814;0.7814 0];
b = [1 -1;0 2];
c = [1.9691 6.4493];
sys = ss(a,b,c,0);
step(sys)
```

# step



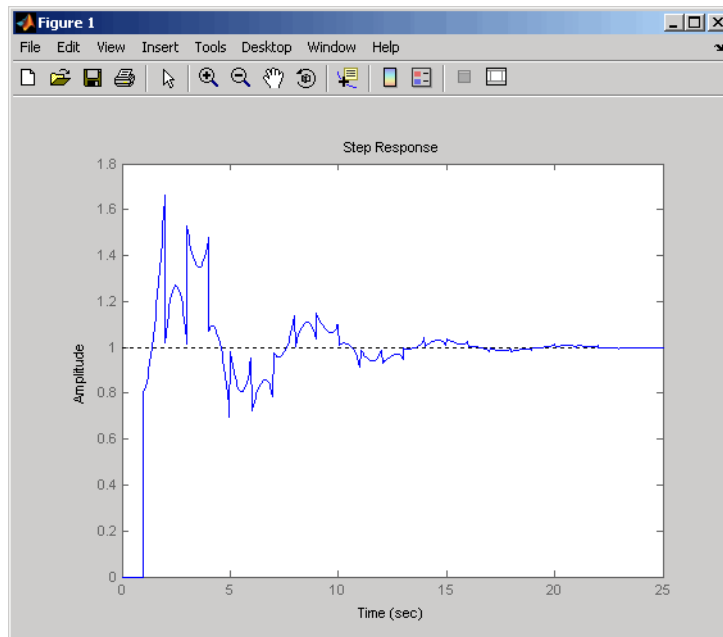
The left plot shows the step response of the first input channel, and the right plot shows the step response of the second input channel.

## Example 2

### Step Response Plot of Feedback Loop with Delay

Create a feedback loop with delay and plot its step response by typing

```
G = exp(-s) * (0.8*s^2+s+2)/(s^2+s);  
T = feedback(ss(G),1);  
step(T)
```



---

**Note** The system step response displayed is chaotic. The step response of systems with internal delays may exhibit odd behavior, such as recurring jumps. Such behavior is a feature of the system and not software anomalies.

---

### Example 3

Compare the step response of a parametric identified model to a non-parametric (empirical) model/ Also view their  $3\text{-}\sigma$  confidence regions.

```
load iddata1 z1
sys1 = ssest(z1,4);
```

parametric model

```
sys2 = impulseest(z1);

non-parametric model

[y1, ~, ~, ysd1] = step(sys1,t);
[y2, ~, ~, ysd2] = step(sys2,t);

plot(t, y1, 'b', t, y1+3*ysd1, 'b:', t, y1-3*ysd1, 'b:')
hold on
plot(t, y2, 'g', t, y2+3*ysd2, 'g:', t, y2-3*ysd2, 'g:')
```

## Example 4

Validation the linearization of a nonlinear ARX model by comparing their small amplitude step responses.

```
nlsys = 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]);
```

Determine an equilibrium operating point for `nlsys` corresponding to a steady-state input value of 1:

```
u0 = 1;
[X,~,r] = findop(nlsys, 'steady', 1);
y0 = r.SignalLevels.Output;
```

Obtain a linear approximation of `nlsys` at this operating point.

```
sys = linearize(nlsys,u0,X)
```

Now validate the usefulness of `sys` by comparing its small-amplitude step response to that of `nlsys`. The nonlinear system `nlsys` is operating an equilibrium level dictated by  $(u_0, y_0)$ . About this steady-state, we introduce a step perturbation of size 0.1. The corresponding response is computed as follows:

```
opt = stepDataOptions;
opt.InputOffset = u0;
opt.StepAmplitude = 0.1;
```

```
t = (0:0.1:10)';
ynl = step(nlsys, t, opt);
```

The linear system `sys` expresses the relationship between the perturbations in input to the corresponding perturbation in output. It is unaware of nonlinear system's equilibrium values. The step response of the linear system is:

```
opt = stepDataOptions;
opt.StepAmplitude = 0.1;
yl = step(sys, t, opt);
```

To compare, add the steady-state offset, `y0`, to the response of the linear system:

```
plot(t, ynl, t, yl+y0)
legend('Nonlinear', 'Linear with offset')
```

### Example 5

Compute the step response of an identified time series model.

A time series model, also called a signal model, is one without measured input signals. The step plot of this model uses its (unmeasured) noise channel as the input channel to which the step signal is applied.

```
load iddata9
sys = ar(z9, 4);
```

`ys` is a model of the form  $A y(t) = e(t)$ , where  $e(t)$  represents the noise channel. For computation of step response,  $e(t)$  is treated as an input channel, and is named "e@y1".

```
step(sys)
```

## Algorithms

Continuous-time models without internal delays are converted to state space and discretized using zero-order hold on the inputs. The sampling period, `dt`, is chosen automatically based on the system dynamics, except when a time vector `t = 0:dt:Tf` is supplied (`dt` is

then used as sampling period). The resulting simulation time steps  $t$  are equisampled with spacing  $dt$ .

For systems with internal delays, Control System Toolbox software uses variable step solvers. As a result, the time steps  $t$  are not equisampled.

## References

[1] L.F. Shampine and P. Gahinet, "Delay-differential-algebraic equations in control theory," *Applied Numerical Mathematics*, Vol. 56, Issues 3–4, pp. 574–588.

## See Also

`impulse` | `stepDataOptions` | `initial` | `lsim` | `ltiview`

## Purpose

Options set for step

## Syntax

```
opt = stepDataOptions  
opt = stepDataOptions(Name,Value)
```

## Description

`opt = stepDataOptions` creates the default options for `step`.  
`opt = stepDataOptions(Name,Value)` creates an options set with the options specified by one or more `Name,Value` pair arguments.

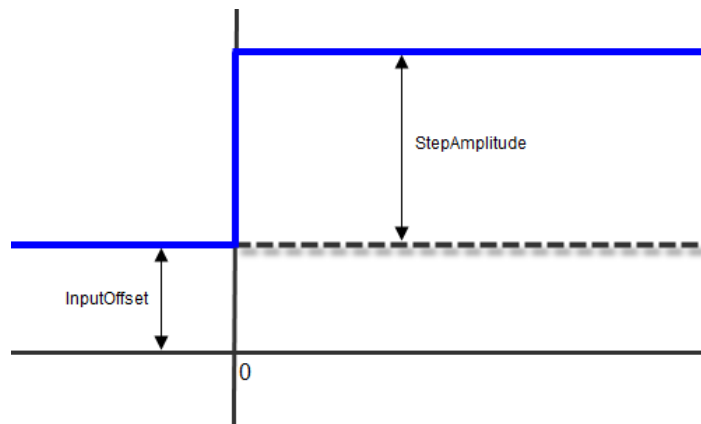
## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### InputOffset

Input signal level for all time  $t < 0$ , as shown in the next figure.



Default: 0

# stepDataOptions

---

## StepAmplitude

Change of input signal level which occurs at time  $t = 0$ , as shown in the previous figure.

**Default:** 1

## Output Arguments

### **opt**

Option set containing the specified options for `step`.

## Specify Input Offset and Step Amplitude Level

Specify the input offset and amplitude level for step response.

```
sys = tf(1,[1,1]);  
opt = stepDataOptions('InputOffset',-1,'StepAmplitude',2);  
[y,t] = step(sys,opt)
```

**See Also** `step`



**Purpose** Rise time, settling time, and other step response characteristics

**Syntax**

```
S = stepinfo(y,t,yfinal)
S = stepinfo(y,t)
S = stepinfo(y)
S = stepinfo(sys)
S = stepinfo(...,'SettlingTimeThreshold',ST)
S = stepinfo(...,'RiseTimeLimits',RT)
```

**Description** `S = stepinfo(y,t,yfinal)` takes step response data  $(t,y)$  and a steady-state value `yfinal` and returns a structure `S` containing the following performance indicators:

- `RiseTime` — Rise time
- `SettlingTime` — Settling time
- `SettlingMin` — Minimum value of  $y$  once the response has risen
- `SettlingMax` — Maximum value of  $y$  once the response has risen
- `Overshoot` — Percentage overshoot (relative to `yfinal`)
- `Undershoot` — Percentage undershoot
- `Peak` — Peak absolute value of  $y$
- `PeakTime` — Time at which this peak is reached

For SISO responses,  $t$  and  $y$  are vectors with the same length `NS`. For systems with `NU` inputs and `NY` outputs, you can specify  $y$  as an `NS`-by-`NY`-by-`NU` array (see `step`) and `yfinal` as an `NY`-by-`NU` array. `stepinfo` then returns a `NY`-by-`NU` structure array `S` of performance metrics for each I/O pair.

`S = stepinfo(y,t)` uses the last sample value of  $y$  as steady-state value `yfinal`. `S = stepinfo(y)` assumes  $t = 1:ns$ .

`S = stepinfo(sys)` computes the step response characteristics for an LTI model `sys` (see `tf`, `zpk`, or `ss` for details).

`S = stepinfo(...,'SettlingTimeThreshold',ST)` lets you specify the threshold `ST` used in the settling time calculation. The response

has settled when the error  $|y(t) - y_{\text{final}}|$  becomes smaller than a fraction ST of its peak value. The default value is  $ST=0.02$  (2%).

`S = stepinfo(..., 'RiseTimeLimits', RT)` lets you specify the lower and upper thresholds used in the rise time calculation. By default, the rise time is the time the response takes to rise from 10 to 90% of the steady-state value ( $RT=[0.1 \ 0.9]$ ). Note that `RT(2)` is also used to calculate `SettlingMin` and `SettlingMax`.

## Examples

### Step Response Characteristics of Fifth-Order System

Create a fifth order system and ascertain the response characteristics.

```
sys = tf([1 5],[1 2 5 7 2]);  
S = stepinfo(sys, 'RiseTimeLimits', [0.05,0.95])
```

These commands return the following result:

```
S =  
  
    RiseTime: 7.4454  
    SettlingTime: 13.9378  
    SettlingMin: 2.3737  
    SettlingMax: 2.5201  
    Overshoot: 0.8032  
    Undershoot: 0  
         Peak: 2.5201  
    PeakTime: 15.1869
```

## See Also

`step` | `lsiminfo`

**Purpose** Plot step response and return plot handle

**Syntax**

```
h = stepplot(sys)
stepplot(sys,Tfinal)
stepplot(sys,t)
stepplot(sys1,sys2,...,sysN)
stepplot(sys1,sys2,...,sysN,Tfinal)
stepplot(sys1,sys2,...,sysN,t)
stepplot(AX,...)
stepplot(..., plotoptions)
```

**Description** `h = stepplot(sys)` plots the step response of the dynamic system model `sys`. It also returns the plot handle `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands.

Type

`help timeoptions`

for a list of available plot options.

For multiinput models, independent step commands are applied to each input channel. The time range and number of points are chosen automatically.

`stepplot(sys,Tfinal)` simulates the step response from  $t = 0$  to the final time  $t = T_{\text{final}}$ . Express `Tfinal` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time systems with unspecified sampling time ( $T_s = -1$ ), `stepplot` interprets `Tfinal` as the number of sampling intervals to simulate.

`stepplot(sys,t)` uses the user-supplied time vector `t` for simulation. Express `t` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time models, `t` should be of the form `Ti:Ts:Tf`, where `Ts` is the sample time. For continuous-time models, `t` should be of the form `Ti:dt:Tf`, where `dt` becomes the sample time of a discrete approximation to the continuous system (see `step`). The `stepplot` command always applies the step input at  $t=0$ , regardless of `Ti`.

# stepplot

---

To plot the step responses of multiple models `sys1,sys2,...` on a single plot, use:

```
stepplot(sys1,sys2,...,sysN)
```

```
stepplot(sys1,sys2,...,sysN,Tfinal)
```

```
stepplot(sys1,sys2,...,sysN,t)
```

You can also specify a color, line style, and marker for each system, as in

```
stepplot(sys1,'r',sys2,'y--',sys3,'gx')
```

```
stepplot(AX,...) plots into the axes with handle AX.
```

```
stepplot(..., plotoptions) plots the step response with the options specified in plotoptions. Type
```

```
help timeoptions
```

for more details.

## Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

## Examples

### Example 1

Use the plot handle to normalize the responses on a step plot.

```
sys = rss(3);  
h = stepplot(sys);  
% Normalize responses.  
setoptions(h,'Normalize','on');
```

### Example 2

Compare the step response of a parametric identified model to a non-parametric (empirical) model, and view their 3- $\sigma$  confidence regions.

```
load iddata1 z1

for parametric model
sys1 = ssest(z1,4);

non-parametric model

sys2 = impulseest(z1);
t = -1:0.1:5;
h = stepplot(sys1,sys2,t);
showConfidence(h, true, 3)
```

The non-parametric model `sys2` shows higher uncertainty.

### Example 3

Plot the step response of a nonlinear (Hammerstein-Wiener) model using a starting offset of 2 and step amplitude of 0.5.

```
load twotankdata
z = iddata(y, u, 0.2, 'Name', 'Two tank system');
sys = nlhw(z, [1 5 3], pwlinear, poly1d);

step(sys, 60, stepDataOptions('InputOffset', 2, 'StepAmplitude', 0.5))
```

### See Also

[getoptions](#) | [setoptions](#) | [showConfidence](#) | [step](#)

# strseq

---

**Purpose** Create sequence of indexed strings

**Syntax** `strvec = strseq(STR,INDICES)`

**Description** `strvec = strseq(STR,INDICES)` creates a sequence of indexed strings in the string vector `strvec` by appending the integer values `INDICES` to the string `STR`.

---

**Note** You can use `strvec` to aid in system interconnection. For an example, see the `sumblk` reference page.

---

**Examples** Create a string vector by indexing the string 'e' at 1, 2, and 4.

```
strseq('e',[1 2 4])
```

This command returns the following result:

```
ans =  
    'e1'  
    'e2'  
    'e4'
```

**See Also** `strcat` | `connect`

**Purpose** Generate model-order combinations for single-output ARX model estimation

**Syntax** `nn = struc(na,nb,nk)`  
`nn = struc(na,nb_1,...,nb_nu,nk_1,...,nk_nu)`

**Description** `nn = struc(na,nb,nk)` generates model-order combinations for single-input ARX model estimation. *na* and *nb* are row vectors that specify range of model orders. *nk* is a row vector that specifies range of model delays. *nn* is a matrix that contains all combinations of the orders and delays.

`nn = struc(na,nb_1,...,nb_nu,nk_1,...,nk_nu)` generates model-order combinations for ARX model with *nu* input channels.

**Tips**

- Use with `arxstruc` or `ivstruc` to compute loss functions for ARX models, one for each model order combination returned by `struc`.

**Examples** Generate model-order combinations for single-input ARX model estimation:

```
% na and nb vary between 1 and 2, nk varies between 4 and 5.
NN = struc(1:2,1:2,4:5);
```

---

Generate model-order combinations, and estimate multi-input ARX model:

```
% Create estimation and validation data sets.
load co2data;
Ts = 0.5; % Sampling interval is 0.5 min
ze = iddata(Output_exp1,Input_exp1,Ts);
zv = iddata(Output_exp2,Input_exp2,Ts);

% Generate model-order combinations for na=2:4,
% nb=2:5 for the first input and 1 or 4 for the second input,
% nk=1:4 for the first input and 0 for the second input.
```

# struc

---

```
NN = struc(2:4, 2:5, [1 4], 1:4, 0);

% Estimate an ARX model for each model order.
V = arxstruc(ze, zv, NN);

% Select a model order.
order=selstruc(V,0);

% Estimate an ARX model of selected order.
M=arx(ze,order);
```

## See Also

[arxstruc](#) | [ivstruc](#) | [selstruc](#)

## Tutorials

- “Estimating Model Orders Using an ARX Model Structure”

## How To

- “Preliminary Step – Estimating Model Orders and Input Delays”



**Purpose**

Access transfer function data

**Syntax**

```
[num,den] = tfdata(sys)
[num,den,Ts] = tfdata(sys)
[num,den,Ts,sdnum,sdden]=tfdata(sys)
[num,den,Ts,...]=tfdata(sys,J1,...,Jn)
```

**Description**

`[num,den] = tfdata(sys)` returns the numerator(s) and denominator(s) of the transfer function for the TF, SS or ZPK model (or LTI array of TF, SS or ZPK models) `sys`. For single LTI models, the outputs `num` and `den` of `tfdata` are cell arrays with the following characteristics:

- `num` and `den` have as many rows as outputs and as many columns as inputs.
- The  $(i, j)$  entries `num{i, j}` and `den{i, j}` are row vectors specifying the numerator and denominator coefficients of the transfer function from input  $j$  to output  $i$ . These coefficients are ordered in *descending* powers of  $s$  or  $z$ .

For arrays `sys` of LTI models, `num` and `den` are multidimensional cell arrays with the same sizes as `sys`.

If `sys` is a state-space or zero-pole-gain model, it is first converted to transfer function form using `tf`. For more information on the format of transfer function model data, see the `tf` reference page.

For SISO transfer functions, the syntax

```
[num,den] = tfdata(sys, 'v')
```

forces `tfdata` to return the numerator and denominator directly as row vectors rather than as cell arrays (see example below).

`[num,den,Ts] = tfdata(sys)` also returns the sample time `Ts`.

`[num,den,Ts,sdnum,sdden]=tfdata(sys)` also returns the uncertainties in the numerator and denominator coefficients of identified system `sys`. `sdnum{i, j}(k)` is the 1 standard uncertainty

in the value  $\text{num}\{i,j\}(k)$  and  $\text{sdden}\{i,j\}(k)$  is the 1 standard uncertainty in the value  $\text{den}\{i,j\}(k)$ . If `sys` does not contain uncertainty information, `snum` and `sdden` are empty (`[]`).

`[num,den,Ts,...]=tfdata(sys,J1,...,Jn)` extracts the data for the (`J1`, ..., `JN`) entry in the model array `sys`.

You can access the remaining LTI properties of `sys` with `get` or by direct referencing, for example,

```
sys.Ts  
sys.variable
```

## Examples

### Example 1

Given the SISO transfer function

```
h = tf([1 1],[1 2 5])
```

you can extract the numerator and denominator coefficients by typing

```
[num,den] = tfdata(h,'v')  
num =  
    0    1    1  
  
den =  
    1    2    5
```

This syntax returns two row vectors.

If you turn `h` into a MIMO transfer function by typing

```
H = [h ; tf(1,[1 1])]
```

the command

```
[num,den] = tfdata(H)
```

now returns two cell arrays with the numerator/denominator data for each SISO entry. Use `celldisp` to visualize this data. Type

```
celldisp(num)
```

This command returns the numerator vectors of the entries of H.

```
num{1} =
    0    1    1
```

```
num{2} =
    0    1
```

Similarly, for the denominators, type

```
celldisp(den)
den{1} =
    1    2    5
```

```
den{2} =
    1    1
```

### Example 2

Extract the numerator, denominator and their standard deviations for a 2-input, 1 output identified transfer function.

```
load iddata7
```

transfer function model

```
sys1 = tfest(z7, 2, 1, 'InputDelay',[1 0]);
```

an equivalent process model

```
sys2 = procest(z7, {'P2UZ', 'P2UZ'}, 'InputDelay',[1 0]);
```

```
[num1, den1, ~, dnum1, dden1] = tfdata(sys1);
```

```
[num2, den2, ~, dnum2, dden2] = tfdata(sys2);
```

### See Also

get | ssdata | tf | zpkdata

**Purpose** Transfer function estimation

**Syntax**

```
sys = tfest(data,np)
sys = tfest(data,np,nz)
sys = tfest(data,np,nz,iodelay)
sys = tfest(__,Name,Value)
sys = tfest(data,init_sys)
sys = tfest(__,opt)
```

**Description**

`sys = tfest(data,np)` estimates a continuous-time transfer function, `sys`, using time- or frequency-domain data, `data`, and contains `np` poles. The number of zeros in the `sys` is `max(np-1,0)`.

`sys = tfest(data,np,nz)` estimates a transfer function containing `nz` zeros.

`sys = tfest(data,np,nz,iodelay)` estimates a transfer function with transport delay for input/output pairs `iodelay`.

`sys = tfest(__,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`sys = tfest(data,init_sys)` uses the dynamic system `init_sys` to configure the initial parameterization of `sys`.

`sys = tfest(__,opt)` specifies the estimation behavior using the option set `opt`.

## Input Arguments

### **data**

Estimation data.

For time domain estimation, `data` is an `iddata` object containing the input and output signal values.

Time series models, which are models that contain no measured inputs, cannot be estimated using `tfest`. Use `ar`, `arx` or `armax` for time series models instead.

For frequency domain estimation, `data` can be one of the following:

- Recorded frequency response data (frd or idfrd)
- iddata object with its properties specified as follows:
  - InputData — Fourier transform of the input signal
  - OutputData — Fourier transform of the output signal
  - Domain — 'Frequency'

For multi-experiment data, the sample times and inter-sample behavior of all the experiments must match.

### **np**

Number of poles in the estimated transfer function.

np is a nonnegative number.

For systems that are multiple-input, or multiple-output, or both:

- To use the same number of poles for all the input/output pairs, specify np as a scalar.
- To use different number of poles for the input/output pairs, specify np as an  $n_y$ -by- $n_u$  matrix. Here  $n_y$  is the number of outputs and  $n_u$  is the number of inputs.

### **nz**

Number of zeros in the estimated transfer function.

nz is a nonnegative number.

For systems that are multiple-input, or multiple-output, or both:

- To use the same number of zeros for all the input/output pairs, specify nz as a scalar.
- To use a different number of zeros for the input/output pairs, specify nz as an  $n_y$ -by- $n_u$  matrix. Here  $n_y$  is the number of outputs and  $n_u$  is the number of inputs.

For a continuous-time model, estimated using discrete-time data, set  $nz \leq np$ .

## **iodelay**

Transport delay.

For continuous-time systems, specify transport delays in the time unit stored in the `TimeUnit` property of `data`. For discrete-time systems, specify transport delays as integers denoting delay of a multiple of the sampling period `TS`.

For a MIMO system with  $n_y$  outputs and  $n_u$  inputs, set `iodelay` to an  $n_y$ -by- $n_u$  array, where each entry is a numerical value representing the transport delay for the corresponding input/output pair. You can also set `iodelay` to a scalar value to apply the same delay to all input/output pairs.

The specified values are treated as fixed delays.

`iodelay` must contain either nonnegative numbers or NaNs. Use NaN in the `iodelay` matrix to denote unknown transport delays.

Use `[]` or `0` to indicate that there is no transport delay.

## **opt**

Estimation options.

`opt` is an options set that specifies the following:

- Estimation objective
- Handling of initial conditions
- Numerical search method to be used in estimation

Use `tfestOptions` to create the options set.

## **init\_sys**

Dynamic system that configures the initial parameterization of `sys`.

If `init_sys` is an `idtf` model, `tfest` uses the parameters and constraints defined in `init_sys` as the initial guess for estimating `sys`.

Use the `Structure` property of `init_sys` to configure initial guesses and constraints for the numerator, denominator and transport lag.

To specify an initial guess for, say, the numerator of `init_sys`, set `init_sys.Structure.num.Value` to the initial guess.

To specify constraints for, say, the numerator of `init_sys`:

- Set `init_sys.Structure.num.Minimum` to the minimum numerator coefficient values
- Set `init_sys.Structure.num.Maximum` to the maximum numerator coefficient values
- Set `init_sys.Structure.num.Free` to indicate which numerator coefficients are free for estimation

You can similarly specify the initial guess and constraints for the denominator and transport lag.

If `init_sys` is not an `idtf` model, the software first converts `init_sys` to a transfer function. `tfest` uses the parameters of the resulting model as the initial guess for estimation.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

#### **Ts**

Sampling time.

Use the following values for `Ts`:

- 0 — Continuous-time model.
- `data.Ts` — Discrete-time model. In this case, `np` and `nz` refer to the number of roots of  $z^{-1}$  for the numerator and denominator polynomials.

**Default:** 0

## InputDelay

Input delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. For continuous-time systems, specify input delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify input delays in integer multiples of the sampling period `Ts`. For example, `InputDelay = 3` means a delay of three sampling periods.

For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector, where each entry is a numerical value representing the input delay for the corresponding input channel. You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0 for all input channels

## Feedthrough

Feedthrough for discrete-time transfer function. Must be a `Ny`-by-`Nu` logical matrix. Use a scalar to specify a common value across all channels.

A discrete-time model with 2 poles and 3 zeros takes the following form:

$$Hz^{-1} = \frac{b0 + b1z^{-1} + b2z^{-2} + b3z^{-3}}{1 + a1z^{-1} + a2z^{-2}}$$

When the model has direct feedthrough, `b0` is a free parameter whose value is estimated along with the rest of the model parameters `b1`, `b2`, `b3`, `a1`, `a2`. When the model has no feedthrough, `b0` is fixed to zero.

**Default:** false (`Ny,Nu`)

## Output Arguments

**sys**

Identified transfer function.



`sys` is an `idtf` model which encapsulates the identified transfer function.

## Examples

### Specify Number of Poles in Identified Transfer Function

Identify a transfer function containing a specified number of poles for given data.

Load time domain system response data and use it to identify a transfer function for the system.

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

`z1` is an `iddata` object that contains time domain input-output data.

`np` specifies the number of poles in the identified transfer function.

`sys` is an `idtf` model containing the identified transfer function.

To see the numerator and denominator coefficients of the resulting estimated model `sys`, enter:

```
sys.num
sys.den
```

### Specify Number of Poles and Zeros in Identified Transfer Function

Identify a transfer function containing a specified number of poles and zeros for given data.

Load time domain system response data and use it to identify a transfer function for the system.

```
load iddata2 z2;
np = 2;
nz = 1;
sys = tfest(z2,np,nz);
```

`z2` is an `iddata` object that contains time domain system response data.

`np` and `nz` specify the number of poles and zeros in the identified transfer function, respectively.

`sys` is an `idtf` model containing the identified transfer function.

## Known Transport Delay

Identify a transfer function containing a known transport delay for given data.

Load time domain system response data and use it to identify a transfer function for the system. Specify a known transport delay for the transfer function.

```
load iddata2 z2;
np = 2;
nz = 1;
iodelay = 0.2;
sys = tfest(z2,np,nz,iodelay);
```

`z2` is an `iddata` object that contains time domain system response data.

`np` and `nz` specify the number of poles and zeros in the identified transfer function, respectively.

`iodelay` specifies the transport delay for the identified transfer function as 0.2 seconds.

`sys` is an `idtf` model containing the identified transfer function, with `ioDelay` set to 0.2 seconds.

## Unknown Transport Delay

Identify a transfer function containing an unknown transport delay for given data.

Load time domain system response data and use it to identify a transfer function for the system. Specify an unknown transport delay for the transfer function.

```
load iddata2 z2;
np = 2;
nz = 1;
iodelay = NaN;
sys = tfest(z2,np,nz,iodelay);
```

`z2` is an `iddata` object that contains time domain system response data.

`np` and `nz` specify the number of poles and zeros in the identified transfer function, respectively.

`iodelay` specifies the transport delay for the identified transfer function. `iodelay = NaN` denotes the transport delay as an unknown parameter to be identified.

`sys` is an `idtf` model containing the identified transfer function, whose `ioDelay` is estimated using data.

### **Estimate Discrete-Time Transfer Function With No Feedthrough**

Identify a discrete-time transfer function for given data.

Load time-domain system response data.

```
load iddata2 z2;
```

`z2` is an `iddata` object that contains time domain system response data.

Identify a transfer function with a sample time and known transport delay

```
np = 2;
nz = 1;
iodelay = 2;
Ts = 0.1;
sysd = tfest(z2,np,nz,iodelay,'Ts',Ts);
```

By default, the model has no feedthrough.

### **Estimate Discrete-Time Transfer Function With Feedthrough**

Estimate a discrete-time transfer function whose numerator polynomial has a nonzero leading coefficient.

```
load iddata5 z5
np = 3;
nz = 1;
model = tfest(z5,np,nz,'ts',z5.ts,'Feedthrough',true);
```

### **Analyze the Origin of Delay in Measured Data**

This example compares two discrete-time models with and without feedthrough and transport delay.

If there is a delay in measured data, it can be because of lack of feedthrough or due to transport delay. The effect of absence of feedthrough can also be described using a transport delay of 1 sample. Estimating a model with `Feedthrough = true` and `ioDelay = 1` produces a discrete-time system that is equivalent to a system with `Feedthrough = false` and `ioDelay = 0`. Both systems show the same Bode response. However, you get different results in continuous-time representation. Therefore, you should check if the delay is due to transport delay or simply because of lack of feedthrough.

Estimate a discrete-time model with no feedthrough and 1 sample input-output delay.

```
load iddata1 z1
np = 2;
nz = 2;
model1 = tfest(z1, np, nz, 'Ts', z1.Ts);
```

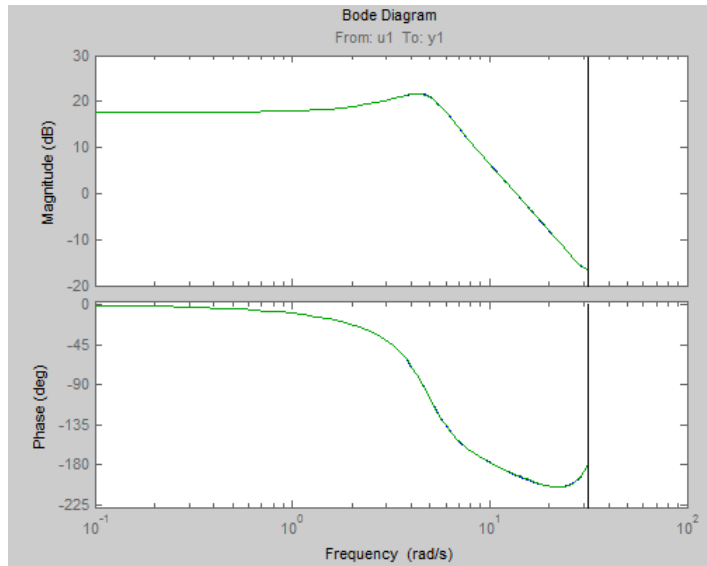
`nz` is 2 because number of zeros refers to the roots of numerator polynomial expressed in  $z^{-1}$ .

Estimate another discrete-time model with feedthrough and 1 sample input-output delay.

```
model2 = tfest(z1, np, nz-1, 1, 'Ts', z1.Ts, 'Feedthrough', true);
```

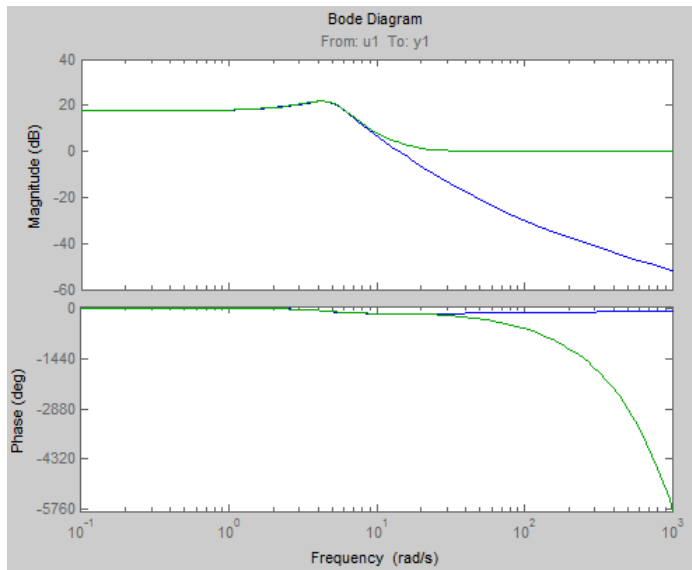
Compare the Bode response of the models.

```
bode(model1,model2);
```



Convert the models to continuous-time, and compare their Bode responses.

```
bode(d2c(model1),d2c(model2));
```



As the plot shows, the Bode responses of the two models do not match when you convert them to continuous time.

### Estimate MISO Discrete-Time Transfer Function with Feedthrough and Delay Specifications for Individual Channels

Estimate a 2-input, 1-output discrete-time transfer function with a delay of 2 samples on first input. There is no delay for the second input but there is an absence of feedthrough.

Split data into estimation and validation data sets.

```
load iddata7 z7
ze = z7(1:300);
zv = z7(200:400);
```

Estimate a 2-input, 1-output transfer function with 2 poles and 1 zero for each input-to-output transfer function.

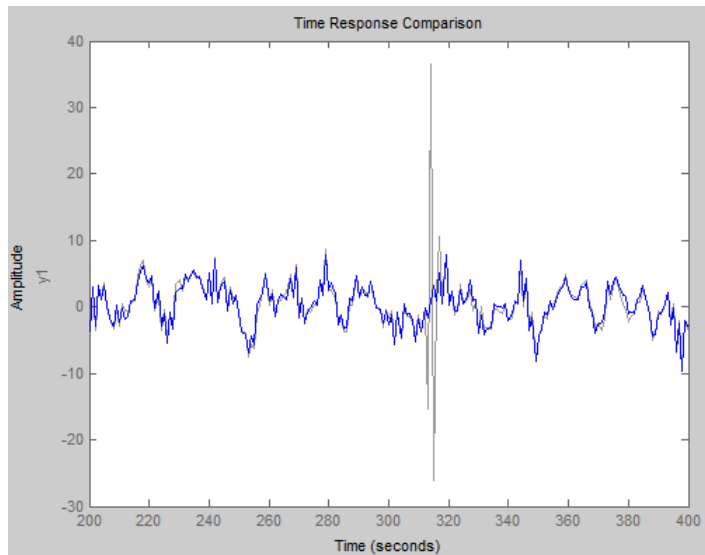
```
Lag = [2; 0];  
Ft = [false, false];  
model = tfest(ze, 2, 1, 'Ts', z7.Ts, 'Feedthrough', Ft, 'InputDelay',
```

Choice of `Feedthrough` dictates whether the leading numerator coefficient is zero (no feedthrough) or not (nonzero feedthrough). Delays are expressed separately using `InputDelay` or `ioDelay` property. This example uses `InputDelay` to express the delays.

Validate the estimated model.

```
I = 1:201; I(114:118) = [];  
opt = compareOptions('Samples',I);  
compare(zv, model, opt)
```

Exclude the data outliers for validation.



## Specify Estimation Options to Identify a Transfer Function

Identify a transfer function for given data and specify the estimation options.

Create the options set for `tfest`.

```
opt = tfestOptions('InitMethod', 'n4sid', 'Display', 'on', 'SearchMethod'
```

`opt` specifies that the initialization method as 'n4sid', the search method as 'lsqnonlin' and that the loss-function values for each iteration be shown.

Load time domain system response data and use it to identify a transfer function for the system. Specify the estimation options using `opt`.

```
load iddata2 z2;  
np = 2;  
nz = 1;  
iodelay = 0.2;  
sysc = tfest(z2,np,nz,iodelay,opt);
```

`z2` is an `iddata` object that contains time domain system response data.

`np` and `nz` specify the number of poles and zeros in the identified transfer function, respectively.

`iodelay` specifies the transport delay for the identified transfer function as 0.2 seconds.

`opt` specifies the estimation options for identifying the transfer function.

`sys` is an `idtf` model containing the identified transfer function.

## Specify Model Properties of the Identified Transfer Function

Identify a transfer function for given data and specify the model properties for the resulting `idtf` model.



Load time domain system response data and use it to identify a transfer function for the system. Specify the input delay for the identified transfer function.

```
load iddata2 z2;
np = 2;
nz = 1;
input_delay = 0.2;
sys = tfest(z2,np,nz,'InputDelay',input_delay)
```

z2 is an iddata object that contains time domain system response data.

np and nz specify the number of poles and zeros in the identified transfer function, respectively.

input\_delay specifies the input delay for the identified transfer function as 0.2 seconds.

sys is an idtf model containing the identified transfer function with an input delay of 0.2 seconds.

### **Convert Frequency Response Data (FRD) into Transfer Function**

Identify a transfer function to fit a given frequency response data (FRD).

Obtain frequency response data.

For example, use bode to obtain the magnitude and phase response data for the following system:

$$H(s) = \frac{s + 0.2}{s^3 + 2s^2 + s + 1}$$

Use 100 frequency points, ranging from 0.1 rad/s to 10 rad/s, to obtain the frequency response data. Use frd to create a frequency response data object.

```
freq = logspace(-1,1,100);
[mag, phase] = bode(tf([1 .2],[1 2 1 1]),freq);
data = frd(mag.*exp(1j*phase*pi/180),freq);
```

Estimate a transfer function using data.

```
np = 3;  
nz = 1;  
sys = tfest(data,np,nz);
```

np and nz specify the number of poles and zeros in the identified transfer function, respectively.

sys is an idtf model containing the identified transfer function.

## **Identify Transfer Function with Transport Delay to Fit Given Frequency Response Data**

Identify a transfer function to fit a given frequency response data (FRD) containing additional phase roll off induced by input delay.

Obtain frequency response data.

For this example, use bode to obtain the magnitude and phase response data for the following system:

$$H(s) = e^{-.5s} \frac{s + 0.2}{s^3 + 2s^2 + s + 1}$$

Use 100 frequency points, ranging from 0.1 rad/s to 10 rad/s, to obtain the frequency response data. Use frd to create a frequency response data object.

```
freq = logspace(-1,1,100);  
[mag, phase] = bode(tf([1 .2],[1 2 1 1],'InputDelay',.5),freq);  
data = frd(mag.*exp(1j*phase*pi/180),freq);
```

data is an iddata object that contains frequency response data for the described system.

Estimate a transfer function using data. Specify an unknown transport delay for the identified transfer function.

```
np = 3;  
nz = 1;
```

```
iodelay = NaN;  
sys = tfest(data,np,nz,iodelay)
```

`np` and `nz` specify the number of poles and zeros in the identified transfer function, respectively.

`iodelay` specifies an unknown transport delay for the identified transfer function.

`sys` is an `idtf` model containing the identified transfer function.

### **Specify Identified Transfer Function Model Structure and Constraints**

Identify a transfer function for given data by providing its expected structure and coefficient constraints.

Load time domain data.

```
load iddata1 z1;  
z1.y = cumsum(z1.y);
```

This command integrates the output data of `z1`. The identified transfer function should therefore contain an integrator.

Create a transfer function model with the expected structure.

```
init_sys = idtf([100 1500],[1 10 10 0]);
```

`int_sys` is an `idtf` model with three poles and one zero. The denominator coefficient for the  $s^0$  term is zero. Therefore, `int_sys` contains an integrator.

Specify constraints on the numerator and denominator coefficients of the transfer function model. To do so, configure fields in the `Structure` property of

```
init_sys.Structure.num.Minimum = eps;  
init_sys.Structure.den.Minimum = eps;  
init_sys.Structure.den.Free(end) = false;
```

The constraints specify that the numerator and denominator coefficients are nonnegative. Additionally, the last element of the denominator coefficients (associated with the  $s^0$  term) is not a free estimation variable. This constraint forces one of the estimated poles to be at  $s = 0$ .

Create an estimation option set that specifies using the Levenberg–Marquardt search method.

```
opt = tfestOptions('SearchMethod', 'lm');
```

Estimate a transfer function for `z1` using `init_sys` and the estimation option set.

```
sys = tfest(z1,init_sys,opt);
```

`tfest` uses the coefficients of `init_sys` to initialize the estimation of `sys`. Additionally, the estimation is constrained by the constraints you specify in the `Structure` property of `init_sys`. The resulting `idtf` model `sys` contains the parameter values that result from the estimation.

## Known Transport Delays for Multiple Inputs

Identify a transfer function with known transport delays for given data.

Load time domain system response data.

```
load co2data;  
Ts = 0.5;  
data = iddata(Output_exp1,Input_exp1,Ts);
```

`data` is an `iddata` object and has a sample rate of 0.5 seconds.

Specify the search method as `gna` and the maximum search iterations. Also specify the input and output offsets.

```
opt = tfestOptions('SearchMethod', 'gna');  
opt.InputOffset = [170; 50];  
opt.OutputOffset = mean(data.y(1:75));  
opt.SearchOption.MaxIter = 50;
```

`opt` is an estimation option set that specifies the search method as `gna`, with a maximum of 50 iterations. `opt` also specifies the input offset and the output offset.

Identify a transfer function using the measured data and the estimation option set. Specify the transport delays from the inputs to the output.

```
np = 3;  
nz = 1;  
iodelay = [2 5];  
sys = tfest(data,np,nz,iodelay,opt);
```

`iodelay` specifies the input to output delay from the first and second inputs to the output as 2 seconds and 5 seconds, respectively.

`sys` is an `idtf` model containing the identified transfer function.

### **Known and Unknown Transport Delays**

Identify a transfer function containing known and unknown input to output delays for given data.

Load time domain system response data and use it to identify a transfer function for the system. Specify the known and unknown transport delays.

```
load co2data;  
Ts = 0.5;  
data = iddata(Output_exp1,Input_exp1,Ts);
```

`data` is an `iddata` object and has a sample rate of 0.5 seconds.

Specify the search method as `gna` and the maximum search iterations. Also specify the input and output offsets.

```
opt = tfestOptions('Display','on','SearchMethod','gna');  
opt.InputOffset = [170; 50];  
opt.OutputOffset = mean(data.y(1:75));  
opt.SearchOption.MaxIter = 50;
```

`opt` is an estimation option set that specifies the search method as `gna`, with a maximum of 50 iterations. `opt` also specifies the input offset and the output offset.

Identify a transfer function using the measured data and the estimation option set. Specify the unknown and known transport delays.

```
np = 3;  
nz = 1;  
iodelay = [2 nan];  
sys = tfest(data,np,nz,iodelay,opt)
```

`iodelay` specifies the transport delay from the first input to the output as 2 seconds. Using `NaN` specifies the transport delay from the second input to the output as unknown.

`sys` is an `idtf` model containing the identified transfer function.

## Unknown Transport Delays with Constraints

Identify a transfer function containing unknown transport delays with constraints for given data.

Create a transfer function model with the expected numerator and denominator structure and delay constraints.

For this example, the experiment data consists of two inputs and one output. Both transport delays are unknown and have an identical upper bound. Additionally, the transfer functions from both inputs to the output are identical in structure.

```
init_sys = idtf(NaN(1,2),[1, NaN(1,3)],'ioDelay',NaN);  
init_sys.Structure(1).ioDelay.Free = true;  
init_sys.Structure(1).ioDelay.Maximum = 7;
```

`init_sys` is an `idtf` model describing the structure of the transfer function from one input to the output. The transfer function consists of one zero, three poles and a transport delay. The use of `NaN` indicates unknown coefficients.

`init_sys.Structure(1).ioDelay.Free = true` indicates that the transport delay is not fixed.

`init_sys.Structure(1).ioDelay.Maximum = 7` sets the upper bound for the transport delay to 7 seconds.

```
init_sys = [init_sys, init_sys];
```

`init_sys` now contains the transfer function from both inputs to the output.

Load time domain system response data and use it to identify a transfer function for the system.

```
load co2data;
Ts = 0.5;
data = iddata(Output_exp1,Input_exp1,Ts);
opt = tfestOptions('Display','on','SearchMethod','gna');
opt.InputOffset = [170; 50];
opt.OutputOffset = mean(data.y(1:75));
opt.SearchOption.MaxIter = 50;
sys = tfest(data,init_sys,opt)
```

`data` is an `iddata` object and has a sample rate of 0.5 seconds.

`opt` is an estimation option set that specifies the search method as `gna`, with a maximum of 50 iterations. `opt` also specifies the input offset and the output offset.

`sys` is an `idtf` model containing the identified transfer function.

Analyze the estimation result by comparison.

```
opt2 = compareOptions;
opt2.InputOffset = opt.InputOffset;
opt2.OutputOffset = opt.OutputOffset;
compare(data, sys, opt2)
```

## Different Number of Poles for Input/Output Pairs

Identify a multiple-input, single-output transfer function containing different number of poles for input/output pairs for given data.

Obtain frequency response data.

For example, use `frd` to frequency response data model for the following system:

$$G = \begin{bmatrix} e^{-4s} \frac{s+2}{s^3+2s^2+4s+5} \\ e^{-0.6s} \frac{5}{s^4+2s^3+s^2+s} \end{bmatrix}$$

Use 100 frequency points, ranging from 0.01 rad/s to 100 rad/s, to obtain the frequency response data.

```
G = tf([1 2],[5]},{[1 2 4 5],[1 2 1 1 0]},0,'ioDelay',[4 .6]);
data = frd(G,logspace(-2,2,100));
```

`data` is an `frd` object containing the continuous-time frequency response for `G`.

Identify a transfer function for `data`.

```
np = [3 4];
nz = [1 0];
iodelay = [4 .6];
sys = tfest(data,np,nz,iodelay);
```

`np` specifies the number of poles in the identified transfer function. The first element of `np` indicates that the transfer function from the first input to the output contains 3 poles. Similarly, the second element of `np` indicates that the transfer function from the second input to the output contains 4 poles.

`nz` specifies the number of zeros in the identified transfer function. The first element of `nz` indicates that the transfer function from the first



input to the output contains 1 zero. Similarly, the second element of `np` indicates that the transfer function from the second input to the output does not contain any zeros.

`idelay` specifies the transport delay from the first input to the output as 4 seconds. The transport delay from the second input to the output is specified as 0.6 seconds.

`sys` is an `idtf` model containing the identified transfer function.

### Identify Transfer Function for Unstable System

Identify a transfer function describing an unstable system for given data.

Obtain frequency response data.

For example, use `frd` to frequency response data model for the following system:

$$G = \begin{bmatrix} \frac{s+2}{s^3+2s^2+4s+5} \\ \frac{5}{s^4+2s^3+s^2+s+1} \end{bmatrix}$$

Use 100 frequency points, ranging from 0.01 rad/s to 100 rad/s, to obtain the frequency response data.

```
G = tf([1 2],[5]},{[1 2 4 5],[1 2 1 1 1]});
data = frd(G,logspace(-2,2,100));
```

`data` is an `frd` object containing the continuous-time frequency response for `G`.

Create estimation options set.

```
opt = tfestOptions('Focus','prediction');
```

Identify a transfer function for `data`, using the options set `opt`.

```
np = [3 4];
```

```
nz = [1 0];  
sys = tfest(data,np,nz,opt);
```

`np` specifies the number of poles in the identified transfer function. The first element of `np` indicates that the transfer function from the first input to the output contains 3 poles. Similarly, the second element of `np` indicates that the transfer function from the second input to the output contains 4 poles.

`nz` specifies the number of zeros in the identified transfer function. The first element of `nz` indicates that the transfer function from the first input to the output contains 1 zero. Similarly, the second element of `np` indicates that the transfer function from the second input to the output does not contain any zeros.

`opt` specifies the estimation options for identifying the transfer function.

`sys` is an `idtf` model containing the identified transfer function.

## Algorithms

`tfest` uses the prediction error minimization (PEM) approach to estimate transfer function coefficients. In general, the estimating algorithm performs two major tasks:

- 1 Initializing the estimable parameters.
- 2 Updating the estimable parameters.

The details of the algorithms used to perform these tasks vary depending on a variety of factors, including the sampling of the estimated model and the estimation data.

### Continuous-Time Transfer Function Estimation Using Time-Domain Data

#### Parameter Initialization

The estimation algorithm initializes the estimable parameters using the method specified by the `InitMethod` estimation option. The default method is the Instrument Variable (IV) method.

The State-Variable Filters (SVF) approach and the Generalized Poisson Moment Functions (GPMF) approach to continuous-time

parameter estimation use pre-filtered data [1] [2]. The constant  $\frac{1}{\tau}$  in [1] and [2] corresponds to the initialization option (`InitOption`) field `FilterTimeConstant`. IV is the simplified refined IV method and is called SRIVC in [1] and [2]. It has a prefilter that is the denominator of the current model, initialized with SVF, and iterated up to `MaxIter` times, until the model change is less than `Tolerance`. `MaxIter` and `Tolerance` are options that you can specify using the `InitOption` structure. The 'n4sid' initialization option estimates a discrete-time model using the 'n4sid' subspace estimation algorithm and then transforms it to continuous-time using `d2c`.

You use `tfestOptions` to create the option set used to estimate a transfer function.

### **Parameter Update**

The initialized parameters are updated using a nonlinear least-squares search method, specified by the `SearchMethod` estimation option. The objective of the search method is to minimize the weighted prediction error norm.

### **Discrete-Time Transfer Function Estimation**

When using discrete-time data, `tfest` uses the same algorithm as `oe` to determine the numerator and denominator polynomial coefficients. In this algorithm, the initialization is performed using `arx`, followed by nonlinear least-squares search based updates to minimize a weighted prediction error norm.

### **Continuous-Time Transfer Function Estimation Using Frequency-Domain Data**

When using continuous-time data and fixed delays, the Output-Error algorithm is used. When using continuous-time data and free delays, or using discrete-time data, the state-space estimation algorithm is used. In this algorithm, the model coefficients are initialized using the `N4SID`

estimation method, followed by nonlinear least-squares search based updates to minimize a weighted prediction error norm.

## Delay Estimation

- When delay values are specified as NaN, they are estimated separate from the model's numerator and denominator coefficients, using `delayest`. The delay values thus determined are treated as fixed values during the iterative update of the model using a nonlinear least-squares search method. That is, the delay values are not iteratively updated. The only exception is the estimation of continuous-time models using continuous-time data.
- When an initial model with finite but free values of `ioDelay` is used (`init_sys.Structure.ioDelay.Value` is a finite value and `init_sys.Structure.ioDelay.Free` is true), those values are updated during estimation only when `init_sys.Ts` is zero and when you are using continuous-time frequency domain data. In all other cases, the initial delay values are left unchanged.

Estimation of delays is often a difficult problem. You should assess the presence and the value of a delay using physical insight of the process being modeled and using functions such as `arxstruc`, `delayest`, and `impulseest`. For an example of determining input delay, see Model Structure Selection: Determining Model Order and Input Delay.

## References

- [1] Garnier, H., M. Mensler, and A. Richard. "Continuous-time Model Identification From Sampled Data: Implementation Issues and Performance Evaluation" *International Journal of Control*. Vol. 76, Issue 13, 2003, pp 1337–1357.
- [2] Ljung, L. "Experiments With Identification of Continuous-Time Models." *Proceedings of the 15th IFAC Symposium on System Identification*. 2009.

## See Also

`tfestOptions` | `idtf` | `ssest` | `procest` | `ar` | `arx` | `oe` | `bj`  
| `polyest` | `greyest`

**Related  
Examples**

- “How to Estimate Transfer Function Models at the Command Line”

**Concepts**

- “What are Transfer Function Models?”

# tfestOptions

---

**Purpose** Options set for tfest

**Syntax**  
`opt = tfestOptions`  
`opt = tfestOptions(Name,Value)`

**Description** `opt = tfestOptions` creates the default options set for tfest.  
`opt = tfestOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### InitMethod

Algorithm used to initialize the values of the numerator and denominator of the output of tfest.

Applies only for estimation of continuous-time transfer functions using time domain data.

`InitMethod` is a string that takes the following values:

- 'iv' — Instrument Variable approach.
- 'svf' — State Variable Filters approach.
- 'gpmf' — Generalized Poisson Moment Functions approach.
- 'n4sid' — Subspace state-space estimation approach.
- 'all' — Combination of all of the above approaches. The software tries all the above methods and selects the method that yields the smallest value of prediction error norm.

**Default:** 'iv'

## InitOption

Options associated with the method used to initialize the values of the numerator and denominator of the output of `tfest`.

`InitOption` is a structure with the following fields:

- `N4Weight` — Calculates the weighting matrices used in the singular-value decomposition step of the `'n4sid'` algorithm. Applicable when `InitMethod` is `'n4sid'`.

`N4Weight` is a string that takes the following values:

- `'MOESP'` — Uses the MOESP algorithm by Verhaegen.
- `'CVA'` — Uses the canonical variable algorithm (CVA) by Larimore.
- `'auto'` — The software automatically determines if the MOESP algorithm or the CVA algorithm should be used in the singular-value decomposition step.

**Default:** `'auto'`

- `N4Horizon` — Determines the forward and backward prediction horizons used by the `'n4sid'` algorithm. Applicable when `InitMethod` is `'n4sid'`.

`N4Horizon` is a row vector with three elements:  $[r \ sy \ su]$ , 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 [1] 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.  $k$  is the number of guesses of  $[r \ sy \ su]$  combinations.

If `N4Horizon = 'auto'`, the software uses an Akaike Information Criterion (AIC) for the selection of  $sy$  and  $su$ .

**Default:** `'auto'`

- **FilterTimeConstant** — Time constant of the differentiating filter in the `svf` and `gpmf` initialization methods.

Specify `FilterTimeConstant` as a positive number, typically greater than 1.

**Default:** 10

- **MaxIter** — Maximum number of iterations. Applicable when `InitMethod` is `'iv'`.

**Default:** 30

- **Tolerance** — Convergence tolerance. Applicable when `InitMethod` is `'iv'`.

**Default:** 0.01

## **InitialCondition**

Specifies how initial conditions are handled during estimation.

- `'zero'` — All initial conditions are taken as zero.
- `'estimate'` — The necessary initial conditions are treated as estimation parameters.
- `'backcast'` — The necessary initial conditions are estimated by a backcasting (backward filtering) process, described in [2].
- `'auto'` — An automatic choice among the above is made, guided by the data.

**Default:** `'auto'`

## **Focus**

Defines how the errors  $e$  between the measured and the modeled outputs are weighed at specific frequencies during the minimization of the prediction error.

Higher weighting at specific frequencies emphasizes the requirement for a good fit at these frequencies.



Focus can take the following values:

- '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.
- prediction — Same as simulation, except that it does not enforce the stability of the resulting model.
- Passbands — Row vector or matrix containing frequency values that define desired passbands. For example:

```
[w1,wh]
[w11,w1h;w21,w2h;w31,w3h;...]
```

where w1 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.

- SISO filter — Enter any SISO linear filter in any of the following ways:
  - A single-input-single-output (SISO) linear system.
  - The {A,B,C,D} format, which specifies the state-space matrices of the filter.
  - The {numerator, denominator} format, 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. To obtain a good model fit for a specific frequency range, you must choose the filter with a passband in this range. The estimation result is the same if you first prefilter the data using `idfilt`.

- Weighting vector — 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.

**Default:** 'simulation'

## **EstCovar**

Controls whether parameter covariance data is generated or not.

If `EstCovar` is true, then use `getcov` to fetch the covariance matrix from the estimated model.

**Default:** true

## **Display**

Specifies whether estimation progress should be displayed.

`Display` requires one of the following strings:

- 'on' — Information on model structure and estimation results are displayed in a progress viewer window
- 'off' — No progress or results information is displayed

**Default:** 'off'

## **InputOffset**

Removes offset from time domain input data during estimation.

Specify as a column vector of length  $Nu$ , where  $Nu$  is the number of inputs.

Use `[]` to indicate no offset.

For multiexperiment data, specify `InputOffset` as a  $Nu$ -by- $Ne$  matrix.  $Nu$  is the number of inputs, and  $Ne$  is the number of experiments.

Each entry specified by `InputOffset` is subtracted from the corresponding input data.

**Default:** []

### **OutputOffset**

Removes offset from time domain output data during estimation.

Specify as a column vector of length  $N_y$ , where  $N_y$  is the number of outputs.

Use [] to indicate no offset.

For multiexperiment data, specify **OutputOffset** as a  $N_y$ -by- $N_e$  matrix.  $N_y$  is the number of outputs, and  $N_e$  is the number of experiments. Each entry specified by **OutputOffset** is subtracted from the corresponding output data.

**Default:** []

### **SearchMethod**

Search method used for iterative parameter estimation.

**SearchMethod** is a string that can take the following values:

- **gn** — The subspace Gauss-Newton direction. Singular values of the Jacobian matrix less than  $\text{GnPinvConst} * \text{eps} * \max(\text{size}(\mathbf{J})) * \text{norm}(\mathbf{J})$  are discarded when computing the search direction.  $\mathbf{J}$  is the Jacobian matrix. The Hessian matrix is approximated by  $\mathbf{J}^T \mathbf{J}$ . If there is no improvement in this direction, the function tries the gradient direction.
- **gna** — An adaptive version of subspace Gauss-Newton approach, suggested by Wills and Ninness [3]. Eigenvalues less than  $\text{gamma} * \max(\text{sv})$  of the Hessian are neglected, where  $\text{sv}$  are the singular values of the Hessian. The Gauss-Newton direction is computed in the remaining subspace. *gamma* has the initial value **InitGnaTol** (see **Advanced** for more information) 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  $2 * \text{LMStep}$  each time a search is successful without any bisections.

- `lm` — Uses the Levenberg-Marquardt method. This means that the next parameter value is  $-\text{pinv}(H+d*I) * \text{grad}$  from the previous one, where  $H$  is the Hessian,  $I$  is the identity matrix, and  $\text{grad}$  is the gradient.  $d$  is a number that is increased until a lower value of the criterion is found.
- `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.
- `grad` — The steepest descent gradient search method.
- `auto` — A choice among the above is made in the algorithm. The descent direction is calculated using `gn`, `gna`, `lm` and `grad` successively, in that order, at each iteration until a sufficient reduction in error is achieved.

**Default:** 'auto'

## SearchOption

SearchOption is an options set for the search algorithm with the fields:

SearchMethod	SearchOption
gn,gna,lm,grad and auto	<ul style="list-style-type: none"> <li>• <b>Tolerance</b> — 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 <b>Tolerance</b>, the iterations are stopped. 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. <b>Default:</b> 0.01</li> <li>• <b>MaxIter</b> — Maximum number of iterations during loss-function minimization. The iterations stop when <b>MaxIter</b> is reached or another stopping criterion is satisfied, such as <b>Tolerance</b>.</li> </ul>

## SearchMethodOption

Setting `MaxIter = 0` returns the result of the startup procedure.

Use `sys.Report.Termination.Iterations` to get the actual number of iterations during an estimation, where `sys` is an `idtf` model.

**Default:** 20

- **Advanced** — Search settings:
  - `GnPinvConst` — Singular values of the Jacobian that are smaller than `GnPinvConst*max(size(J)*norm(J)*eps)` are discarded when computing the search direction and `SearchMethod` is `'gn'`.  
  
`GnPinvConst` must be a positive real value.  
**Default:** 10000
  - `InitGnaTol` — Initial value of *gamma* when `SearchMethod` is `'gna'`.  
**Default:** .0001
  - `LMStartValue` — Starting value of search-direction length *d* in the Levenberg-Marquardt method. Applicable when `SearchMethod` is `'lm'`.  
**Default:** .001
  - `LMStep` — 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. Applicable when `SearchMethod` is `'lm'`.

**Default:** 2

SearchMethod	SearchOption
	<ul style="list-style-type: none"> <li data-bbox="583 361 1233 418"> <p>▪ <b>MaxBisections</b> — Maximum number of bisections used by the line search along the search direction.</p> <p><b>Default:</b> 25</p> </li> <li data-bbox="583 487 1267 595"> <p>▪ <b>MaxFunEvals</b> — Iterations are stopped if the number of calls to the model file exceeds this value.</p> <p>MaxFunEvals must be a positive integer value.</p> <p><b>Default:</b> Inf</p> </li> <li data-bbox="583 664 1270 817"> <p>▪ <b>MinParChange</b> — Smallest parameter update allowed per iteration</p> <p>MinParChange must be a positive, real value.</p> <p><b>Default:</b> 0</p> </li> <li data-bbox="583 843 1262 1029"> <p>▪ <b>RelImprovement</b> — Iterations are stopped if the relative improvement of the criterion function is less than RelImprovement.</p> <p>RelImprovement must be a positive integer value.</p> <p><b>Default:</b> 0</p> </li> <li data-bbox="583 1055 1262 1298"> <p>▪ <b>StepReduction</b> — 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.</p> <p>StepReduction must be a positive, real value that is greater than 1.</p> <p><b>Default:</b> 2</p> </li> </ul>
lsqnonlin	<ul style="list-style-type: none"> <li data-bbox="583 1321 1270 1413"> <p>▪ <b>TolFun</b> — Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values.</p> </li> </ul>

**SearchMethodOption**

The value of TolFun is the same as that of `sys.SearchOption.Advanced.TolFun`.

**Default:** 1e-5

- TolX — Termination tolerance on the estimated parameter values.

The value of TolX is the same as that of `sys.SearchOption.Advanced.TolX`.

**Default:** 1e-6

- MaxIter — Maximum number of iterations during loss-function minimization. The iterations stop when MaxIter is reached or another stopping criterion is satisfied, such as TolFun etc..

The value of MaxIter is the same as that of `sys.SearchOption.Advanced.MaxIter`.

**Default:** 20

- Advanced — Options set for `lsqnonlin`.

For more information, see “Optimization Options”.

Use `optimset('lsqnonlin')` to create an options set for `lsqnonlin` and then modify it to specify its various options.

**Advanced**

Advanced is a structure with the following fields:

- ErrorThreshold — Specifies when to adjust the weight of large errors from quadratic to linear.

Errors larger than ErrorThreshold times the estimated standard deviation have a linear weight in the 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 the chapter “Computing the Estimate” of [1].) `ErrorThreshold = 0` disables robustification and leads to a purely quadratic criterion. When estimating with frequency-domain data, `ErrorThreshold` is set to zero.

**Default:** 0

- `MaxSize` — Specifies the maximum number of elements in a segment when input-output data is split into segments.

`MaxSize` must be a positive integer.

**Default:** 250000

- `StabilityThreshold` — Specifies thresholds for stability tests.

`StabilityThreshold` is a structure with the following fields:

- `s` — Specifies the location of the right-most pole to test the stability of continuous-time models. A model is considered stable when its right-most pole is to the left of `s`.

**Default:** 0

- `z` — 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 `z` from the origin.

**Default:**  $1 + \sqrt{\text{eps}}$

- `AutoInitThreshold` — Specifies when to automatically estimate the initial conditions.

When `InitialCondition = 'Auto'`, the initial condition is estimated when the ratio of the prediction-error norm with a zero initial condition to the norm with an estimated initial condition exceeds `AutoInitialState`.

**Default:** 1.05

## **OutputWeight**

Specifies criterion used during minimization.



OutputWeight can have the following values:

- 'noise' — 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.
- positive semi-definite symmetric matrix (W) — Minimize the trace of the weighted prediction error matrix  $\text{trace}(E^*E*W)$ , where  $E$  is the matrix of prediction errors, with one column for each output, and  $W$  is the positive semi-definite symmetric matrix of size equal to the number of outputs. Use  $W$  to specify the relative importance of outputs in multiple-input multiple-output models, or the reliability of corresponding data.

This option is relevant only for multi-input, multi-output models.

- [] — The software chooses between the 'noise' or using the identity matrix for  $W$ .

## Output Arguments

### opt

Option set containing the specified options for tfest.

## Examples

### Create Default Options Set for Transfer Function Estimation

Create a default options set for tfest.

```
opt = tfestOptions;
```

### Specify Options for Transfer Function Estimation

Create an options set for tfest using the 'n4sid' initialization algorithm and set the Display to 'on'.

```
opt = tfestOptions('InitMethod','n4sid','Display','on');
```

Alternatively, use dot notation to set the values of opt.

```
opt = tfestOptions;
```

```
opt.InitMethod = 'n4sid';  
opt.Display = 'on';
```

## References

- [1] Ljung, L. *System Identification: Theory for the User*, Upper Saddle River, NJ, Prentice-Hal PTR, 1999.
- [2] Knudsen, T. "New method for estimating ARMAX models," *In Proceedings of 10th IFAC Symposium on System Identification, SYSID'94*, Copenhagen, Denmark, July 1994, Vol. 2, pp. 611–617.
- [3] Wills, Adrian, B. Ninness, and S. Gibson. "On Gradient-Based Search for Multivariable System Estimates". *Proceedings of the 16th IFAC World Congress, Prague, Czech Republic, July 3–8, 2005*. Oxford, UK: Elsevier Ltd., 2005.

**See Also** `tfest`

**Purpose** Create list of time plot options

**Syntax**  
`P = timeoptions`  
`P = timeoptions('cstprefs')`

**Description** `P = timeoptions` returns a list of available options for time plots with default values set. You can use these options to customize the time value plot appearance from the command line.

`P = timeoptions('cstprefs')` initializes the plot options you selected in the Control System Toolbox Preferences Editor. For more information about the editor, see “Toolbox Preferences Editor” in the User’s Guide documentation.

This table summarizes the available time plot options.

Option	Description
Title, XLabel, YLabel	Label text and style
TickLabel	Tick label style
Grid	Show or hide the grid Specified as one of the following strings: 'off'   'on' <b>Default:</b> 'off'
XlimMode, YlimMode	Limit modes
Xlim, Ylim	Axes limits
IOGrouping	Grouping of input-output pairs Specified as one of the following strings: 'none'   'inputs'   'output'   'all' <b>Default:</b> 'none'
InputLabel, OutputLabel	Input and output label styles
InputVisible, OutputVisible	Visibility of input and output channels

# timeoptions

---

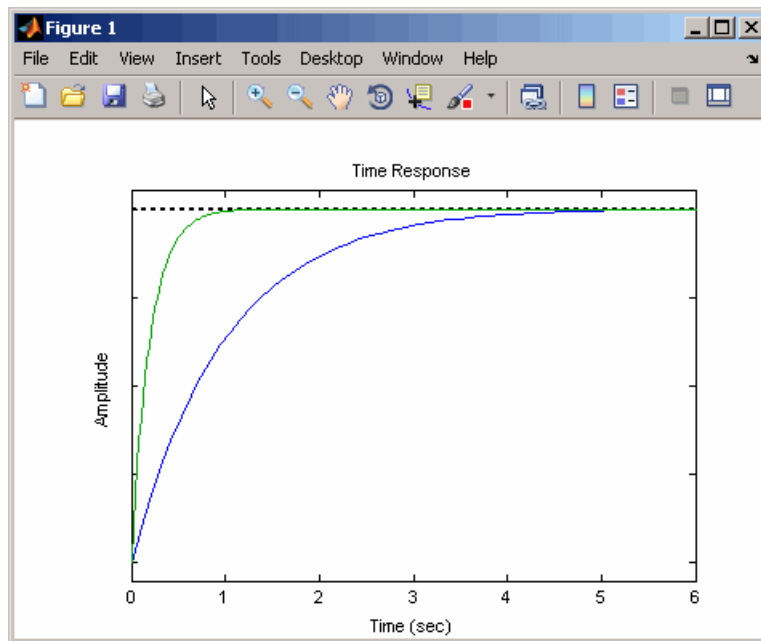
Option	Description
Normalize	Normalize responses Specified as one of the following strings: 'on'   'off' <b>Default:</b> 'off'
SettleTimeThreshold	Settling time threshold
RiseTimeLimits	Rise time limits
TimeUnits	Time units, specified as one of the following strings: <ul style="list-style-type: none"><li>• 'nanoseconds'</li><li>• 'microseconds'</li><li>• 'milliseconds'</li><li>• 'seconds'</li><li>• 'minutes'</li><li>• 'hours'</li><li>• 'days'</li><li>• 'weeks'</li><li>• 'months'</li><li>• 'years'</li></ul> <b>Default:</b> 'seconds'  You can also specify 'auto' which uses time units specified in the TimeUnit property of the input system. For multiple systems with different time units, the units of the first system is used.

## Examples

In this example, enable the normalized response option before creating a plot.

```
P = timeoptions;  
% Set normalize response to on in options  
P.Normalize = 'on';  
% Create plot with the options specified by P  
h = stepplot(tf(10,[1,1]),tf(5,[1,5]),P);
```

The following step plot is created with the responses normalized.



## See Also

[getoptions](#) | [impzplot](#) | [initialplot](#) | [lsimplot](#) | [setoptions](#)  
| [stepplot](#)

# totaldelay

---

**Purpose** Total combined I/O delays for LTI model

**Syntax** `td = totaldelay(sys)`

**Description** `td = totaldelay(sys)` returns the total combined I/O delays for an LTI model `sys`. The matrix `td` combines contributions from the `InputDelay`, `OutputDelay`, and `ioDelayMatrix` properties.

Delays are expressed in seconds for continuous-time models, and as integer multiples of the sample period for discrete-time models. To obtain the delay times in seconds, multiply `td` by the sample time `sys.Ts`.

**Examples**

```
sys = tf(1,[1 0]); % TF of 1/s
sys.inputd = 2; % 2 sec input delay
sys.outputd = 1.5; % 1.5 sec output delay
td = totaldelay(sys)
td =
    3.5000
```

The resulting I/O map is

$$e^{-2s} \times \frac{1}{s} e^{-1.5s} = e^{-3.5s} \frac{1}{s}$$

This is equivalent to assigning an I/O delay of 3.5 seconds to the original model `sys`.

**See Also** `absorbDelay` | `hasdelay`

**Purpose** Translate parameter covariance across model operations

**Syntax**

```
sys_new = translatecov(fcn,sys)
sys_new = translatecov(fcn,sys1,sys2,...sysn)
```

**Description** `sys_new = translatecov(fcn,sys)` translates parameter covariance in the model `sys` during the transformation operation specified in `fcn`. Parameter covariance is computed by applying Gauss Approximation formula on the parameter covariance of `sys`.

`sys_new = translatecov(fcn,sys1,sys2,...sysn)` translates parameter covariance in the multiple models `sys1,sys2,...sysn`. The parameters of the systems are assumed to be uncorrelated.

## Tips

- `translatecov` transforms the model in the same way that directly calling the transformation command does. For example, `translatecov(@(x)d2c(x),sys)` produces a system that has the same coefficients as `d2c(sys)`. The resulting model also has the parameter covariance of `sys`. Using `d2c(sys)` directly does not translate the parameter covariance.
- If you obtained `sys` through estimation and have access to the estimation data, you can use zero-iteration update to recompute the parameter covariance. For example:

```
load iddata1
m = ssest(z1,4);
opt = ssestOptions
opt.SearchOption.MaxIter = 0;
m_new = ssest(z1,m2,opt)
```

You cannot run a zero-iteration update in the following cases:

- If `MaxIter` option, which depends on the `SearchMethod` option, is not available.
- For some model and data types. For example, a continuous-time `idpoly` model using time-domain data.

## Input Arguments

### **fcn**

Function for a model transformation operation, specified as a function handle. The function handle describes the transformation such that:

- For single-model operations, `sys_new = fcn(sys)`. Examples of single-model operations are model-type conversion (`idpoly`, `idss`, ...) and sample time conversion (`c2d`, ...). For example, `fcn = @(x)c2d(x,Ts)`, or `fcn = @idpoly`.
- For multi-model operations, `sys_new = fcn(sys1,sys2,...)`. Examples of multimodel operations are merging and concatenation. For example, `fcn = @(x,y)[x,y]` such that `fcn(sys1,sys2)` performs horizontal concatenation of the models `sys1` and `sys2`.

### **sys**

Linear model, specified as one of the following model types:

- `idtf`
- `idproc`
- `idss`
- `idpoly`
- `idgrey`

The model must contain parameter covariance information (`getcov(sys)` is nonempty).

### **sys1,sys2,...sysn**

Multiple linear models. Models must be of the same type.

## Output Arguments

### **sys\_new**

Model resulting from a transformation operation and includes parameter covariance.



## Definitions

### Examples

#### Translate Parameter Covariance During Model Conversion

Convert an estimated transfer function model into state-space model while also translating the estimated parameter covariance.

Estimate a transfer function model.

```
load iddata1
sys1 = tfest(z1,2);
```

Convert the estimated model to state-space form while also translating the estimated parameter covariance.

```
sys2 = translatecov(@(x)idss(x),sys1);
```

If you convert the transfer function model to state-space form directly, the estimated parameter covariance is lost (the output of `getcov` is empty).

```
sys3 = idss(sys1);
getcov(sys3)
```

```
ans =
```

```
    []
```

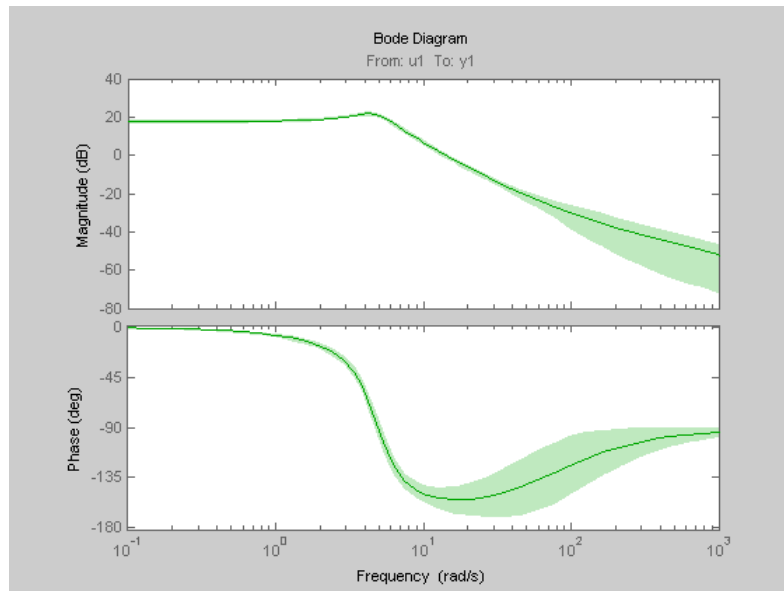
View the parameter covariance in the estimated and converted models.

```
covsys1 = getcov(sys1);
covsys2 = getcov(sys2);
```

Compare the confidence regions.

```
h = bodeplot(sys1,sys2);
showConfidence(h,2);
```

The confidence bounds for `sys1` overlaps with `sys2`.



## Translate Parameter Covariance During Model Concatenation

Concatenate 3 single-output models such that the covariance data from the 3 models combine to produce the covariance data for the resulting model.

Construct a state-space model.

```
a = [-1.1008 0.3733;0.3733 -0.9561];  
b = [0.7254 0.7147;-0.0631 -0.2050];  
c = [-0.1241 0; 1.4897 0.6715; 1.4090 -1.2075];  
d = [0 1.0347; 1.6302 0; 0.4889 0];  
sys = idss(a,b,c,d,'Ts',0);
```

Generate multi-output estimation data.

```
t = (0:0.01:0.99)';  
u = randn(100,2);  
y = lsim(sys,u,t,'zoh');
```

```
y = y + rand(size(y))/10;
data = iddata(y,u,0.01);
```

Estimate a separate model for each output signal.

```
m1 = ssest(data(:,1,:),2,'feedthrough',true(1,2), 'DisturbanceModel',
m2 = ssest(data(:,2,:),2,'feedthrough',true(1,2), 'DisturbanceModel',
m3 = ssest(data(:,3,:),2,'feedthrough',true(1,2), 'DisturbanceModel',
```

Combine the estimated models while also translating the covariance information.

```
f = @(x,y,z)[x;y;z];
M2 = translatecov(f, m1, m2, m3);
getcov(M2, 'factors')
```

The parameter covariance is not empty.

```
getcov(M2, 'factors')
```

```
ans =
```

```
    R: [36x36 double]
    T: [24x36 double]
  Free: [90x1 logical]
```

If you combine the estimated models into one 3-output model directly, the covariance information is lost (the output of `getcov` is empty).

```
M1 = [m1;m2;m3];
getcov(M1)
```

```
ans
```

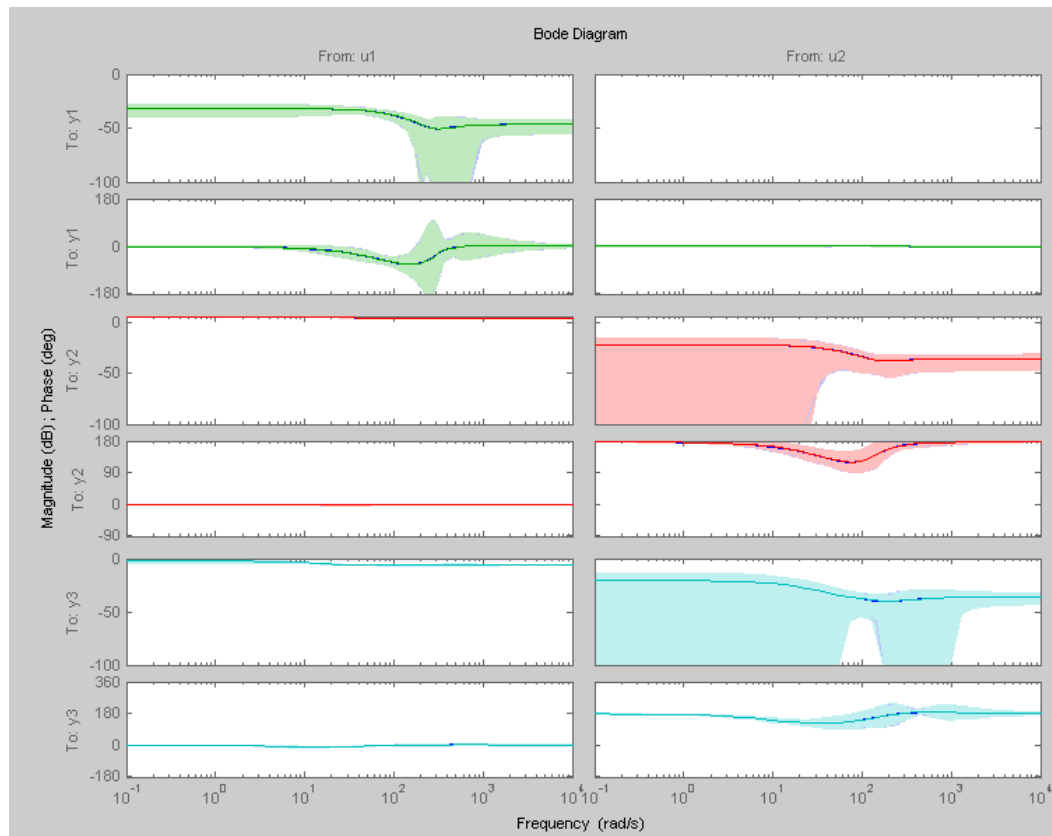
```
[]
```

Compare the confidence bounds.

# translatecov

```
h = bodeplot(M2, m1, m2, m3)  
showConfidence(h);
```

The confidence bounds for M2 overlap with those of m1, m2 and m3 models on their respective plot axes.



## Algorithms

translatecov uses numerical perturbations of individual parameters of sys to compute the Jacobian of fcn(sys) parameters with respect to parameters of sys. translatecov then applies Gauss Approximation

formula  $cov\_new = J \times cov \times J^T$  to translate the covariance, where J is the Jacobian matrix. This operation can be slow for models containing a large number of free parameters.

## See Also

[getcov](#) | [setcov](#) | [getpvec](#) | [resample](#)

## Concepts

- “What Is Model Covariance?”
- “Types of Model Uncertainty Information”

# 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. The object defines 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. Compute the value of  $F$  using `evaluate(t,x)`, where  $t$  is the `treepartition` object at  $x$ .

**Construction** `t=treepartition(Property1,Value1,...PropertyN,ValueN)` creates a binary tree nonlinearity estimator object specified by properties in “`treepartition Properties`” on page 2-972. The tree has the number of leaves equal to  $2^{(J+1)} - 1$ , where  $J$  is the number of nodes in the tree and set by the property `NumberOfUnits`. The default value of `NumberOfUnits` is computed automatically and sets an upper limit on the actual number of tree nodes used by the estimator.

`t=treepartition('NumberOfUnits',N)` creates a binary tree nonlinearity estimator object with  $N$  terms in the binary tree expansion (the number of nodes in the tree). When you estimate a model containing `t`, the value of the `NumberOfUnits` property,  $N$ , 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  $N$ .

**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	<p>Integer specifies the number of nodes in the tree. Default='auto' selects the number of nodes from the data using the pruning algorithm.</p> <p>When you estimate a model containing a <code>treepartition</code> nonlinearity, the value of <code>NumberOfUnits</code> is automatically changed to show the actual number of leaves used—which is the largest integer of the form <math>2^n - 1</math> and less than or equal to <math>N</math> (the integer value of units you specify).</p> <p>For example:</p> <pre>treepartition('NumberOfUnits',5)</pre>
Parameters	<p>Structure containing the following fields:</p> <ul style="list-style-type: none"> <li>• <b>RegressorMean</b>: 1-by-<math>m</math> vector containing the means of <math>x</math> in estimation data, <math>r</math>.</li> <li>• <b>RegressorMinMax</b>: <math>m</math>-by-2 matrix containing the maximum and minimum estimation-data regressor values.</li> <li>• <b>OutputOffset</b>: scalar <math>d</math>.</li> <li>• <b>LinearCoef</b>: <math>m</math>-by-1 vector <math>L</math>.</li> <li>• <b>SampleLength</b>: Length of estimation data.</li> <li>• <b>NoiseVariance</b>: Estimated variance of the noise in estimation data.</li> <li>• <b>Tree</b>: A structure containing the following tree parameters:</li> </ul>

# treepartition

Property Name	Description
	<ul style="list-style-type: none"> <li>▪ <b>TreeLevelPtr:</b> <math>N \times b_y - 1</math> vector containing the levels <math>j</math> of each node.</li> <li>▪ <b>AncestorDescendantPtr:</b> <math>N \times b_y - 3</math> matrix, such that the entry <math>(k, 1)</math> is the ancestor of node <math>k</math>, and entries <math>(k, 2)</math> and <math>(k, 3)</math> are the left and right descendants, respectively.</li> <li>▪ <b>LocalizingVectors:</b> <math>N \times b_y \times (m+1)</math> matrix, such that the <math>r</math>th row is <math>B_r</math>.</li> <li>▪ <b>LocalParVector:</b> <math>N \times b_y \times (m+1)</math> matrix, such that the <math>k</math>th row is <math>C_k</math>.</li> <li>▪ <b>LocalCovMatrix:</b> <math>N \times b_y \times ((m+1)m/2)</math> matrix such that the <math>k</math>th row is the covariance matrix of <math>C_k</math>. <math>C_k</math> is reshaped as a row vector.</li> </ul>
Options	<p>Structure containing the following fields that affect the initial model:</p> <ul style="list-style-type: none"> <li>• <b>FinestCell:</b> Integer or string specifying the minimum number of data points in the smallest partition. Default: 'auto', which computes the value from the data.</li> <li>• <b>Threshold:</b> Threshold parameter used by the adaptive pruning algorithm. Smaller threshold value corresponds to a shorter branch that is terminated by the active partition <math>D_a</math>. Higher threshold value results in a longer branch. Default: 1.0.</li> <li>• <b>Stabilizer:</b> Penalty parameter of the penalized least-squares algorithm used to compute local parameter vectors <math>C_k</math>. Higher stabilizer value improves stability, but may deteriorate the accuracy of the least-square estimate. Default: <math>1e-6</math>.</li> </ul>



## Algorithms

The mapping  $F$  is defined by a dyadic partition  $P$  of the  $x$ -space, such that on each partition element  $P_k$ ,  $F$  is a linear mapping. When  $x$  belongs to  $P_k$ ,  $F(x)$  is given by:

$$F(x) = d + xL + (1, x)C_k,$$

where  $L$  is 1-by- $m$  vector and  $d$  is a scalar common for all elements of partition.  $C_k$  is a 1-by- $(m+1)$  vector.

The mapping  $F$  and associated partition  $P$  of the  $x$ -space are computed as follows:

- 1 Given the value of  $J$ , a dyadic tree with  $J$  levels and  $N = 2^{J-1}$  nodes is initialized.
- 2 Each node at level  $1 < j < J$  has two descendants at level  $j + 1$  and one parent at level  $j - 1$ .
  - The root node at level 1 has two descendants.
  - Nodes at level  $J$  are terminating leaves of the tree and have one parent.
- 3 One partition element is associated to each node  $k$  of the tree.
  - The vector of coefficients  $C_k$  is computed using the observations on the corresponding partition element  $P_k$  by the penalized least-squares algorithm.
  - When the node  $k$  is not a terminating leaf, the partition element  $P_k$  is cut into two to obtain the partition elements of descendant nodes. The cut is defined by the half-spaces  $(1, x)B_k > 0$  or  $\leq 0$  (move to left or right descendant), where  $B_k$  is chosen to improve the stability of least-square computation on the partitions at the descendant nodes.
- 4 When the value of the mapping  $F$ , defined by the treepartition object, is computed at  $x$ , an adaptive algorithm selects the *active node*  $k$  of the tree on the branch of partitions which contain  $x$ .

# treepartition

---

When the `idnlarx` property `Focus` is 'Prediction', `treepartition` uses a noniterative technique for estimating parameters. Iterative refinements are not possible for models containing this nonlinearity estimator.

You cannot use `treepartition` when `Focus` is 'Simulation' because this nonlinearity estimator is not differentiable. Minimization of simulation error requires differentiable nonlinear functions.

## 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 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`

## Purpose

Offset and linear trend slope values for detrending data

## Description

TrendInfo class represents offset and linear trend information of input and output data. Constructing the corresponding object lets you:

- Compute and store mean values or best-fit linear trends of input and output data signals.
- Define specific offsets and trends to be removed from input-output data.

By storing offset and trend information, you can apply it to multiple data sets.

After estimating a linear model from detrended data, you can simulate the model at original operation conditions by adding the saved trend to the simulated output using `retrend`.

## Construction

For transient data, if you want to define a specific offset or trend to be removed from this data, create the TrendInfo object using `getTrend`. For example:

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

where `data` is the `iddata` object from which you will be removing the offset or linear trend, and `T` is the TrendInfo object. You must then assign specific offset and slope values as properties of this object before passing the object as an argument to `detrend`.

For steady-state data, if you want to detrend the data and store the trend information, use the `detrend` command with the output argument for storing trend information.

## Properties

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

# TrendInfo

---

Property Name	Default	Description
DataName	Empty string	Name of the iddata object from which trend information is derived (if any)
InputOffset	<code>zeros(1,nu)</code> , where <code>nu</code> is the number of inputs	<ul style="list-style-type: none"><li>• For transient data, the physical equilibrium offset you specify for each input signal.</li><li>• For steady-state data, the mean of input values. Computed automatically when detrending the data.</li><li>• If removing a linear trend from the input-output data, the value of the line at <code>t0</code>, where <code>t0</code> is the start time.</li></ul> <p>For multiple experiment data, this is a cell array of size equal to the number of experiments in the data set.</p>
InputSlope	<code>zeros(1,nu)</code> , where <code>nu</code> is the number of inputs	<p>Slope of linear trend in input data, computed automatically when using the <code>detrend</code> command to remove the linear trend in the data.</p> <p>For multiple experiment data, this is a cell array of size equal to the number of experiments in the data set.</p>

Property Name	Default	Description
OutputOffset	zeros(1,ny), where ny is the number of outputs	<ul style="list-style-type: none"> <li>• For transient data, the physical equilibrium offset you specify for each output signal</li> <li>• For steady-state data, the mean of output values. Computed automatically when detrending the data.</li> <li>• If removing a linear trend from the input-output data, the value of the line at t0, where t0 is the start time.</li> </ul> <p>For multiple experiment data, this is a cell array of size equal to the number of experiments in the data set.</p>
OutputSlope	zeros(1,ny), where ny is the number of outputs	<p>Slope of linear trend in output data, computed automatically when using the detrend command to remove the linear trend in the data.</p> <p>For multiple experiment data, this is a cell array of size equal to the number of experiments in the data set.</p>

## Examples

Construct the object that stores trend information as part of data detrending:

```
% Load SISO data containing vectors u2 and y2
load dryer2
% Create data object with sampling time of 0.08 sec
data=iddata(y2,u2,0.08)
% Plot data on a time plot - it has a nonzero mean
plot(data)
% Detrend the mean from the data
% Store the mean as TrendInfo object T
[data_d,T] = detrend(data,0)
% View mean value removed from the data
```

`get(T)`

Construct the object that stores input and output offsets to be removed from transient data:

```
% Load SISO data containing vectors u2 and y2
load dryer2
% Create data object with sampling time of 0.08 sec
data=iddata(y2,u2,0.08)
% Plot data on a time plot - it has a nonzero mean
plot(data)
% Create a TrendInfo object for storing offsets and trends
T = getTrend(data)
% Assign offset values to the TrendInfo object
T.InputOffset=5;
T.OutputOffset=5;
% Subtract specific offset from the data
data_d = detrend(data,T)
% View mean value removed from the data
get(T)
```

## See Also

`detrend` | `getTrend` | `retrend`

## How To

- “Handling Offsets and Trends in Data”

**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, use 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.

**Tips** 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, use the following syntax:

```
m = nlhw(Data,Orders,'saturation','unitgain')
```

In this case, the input has a saturation nonlinearity.

If nonlinearities are absent in input or output channels, you can replace `unitgain` with an empty matrix. For example, to specify a Wiener

# unitgain

---

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**

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 2-984.

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

**Tips**

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:

$$\begin{aligned}
 F(x) = & (x-r)PL + a_{s\_1}f(b_{s\_1}((x-r)Q - c_{s\_1})) + \dots \\
 & + a_{s\_ns}f(b_{s\_ns}((x-r)Q - c_{s\_ns})) \\
 & + a_{w\_1}g(b_{w\_1}((x-r)Q - c_{w\_1})) + \dots \\
 & + a_{w\_nw}g(b_{w\_nw}((x-r)Q - c_{w\_nw})) + d
 \end{aligned}$$

where:

- $f$  is a scaling function.
- $g$  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 a  $p$ -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:

$$f(x) = e^{-0.5xx'}$$
$$g(x) = (N_r - xx')e^{-0.5xx'}$$

where  $N_r$  is the length of  $x$  (number of regressors).

## 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	<p>Integer specifies the number of nonlinearity units in the expansion. Default='auto'.</p> <p>For example:</p> <pre>wavenet('NumberOfUnits',5)</pre>
LinearTerm	<p>Can have the following values:</p> <ul style="list-style-type: none"> <li>• 'on' — (Default) Estimates the vector <math>L</math> in the expansion.</li> <li>• 'off' — Fixes the vector <math>L</math> to zero and omits the term <math>(x-r)PL</math>.</li> </ul> <p>For example:</p> <pre>wavenet(H, 'LinearTerm', 'on')</pre>

Property Name	Description
Parameters	<p>Structure containing the parameters in the nonlinear expansion, as follows:</p> <ul style="list-style-type: none"> <li>• <b>RegressorMean</b>: 1-by-m vector containing the means of <math>x</math> in estimation data, <math>r</math>.</li> <li>• <b>NonLinearSubspace</b>: m-by-q matrix containing <math>Q</math>.</li> <li>• <b>LinearSubspace</b>: m-by-p matrix containing <math>P</math>.</li> <li>• <b>LinearCoef</b>: p-by-1 vector <math>L</math>.</li> <li>• <b>ScalingDilation</b>: ns-by-1 matrix containing the values <math>b_{s\_ns}</math>.</li> <li>• <b>WaveletDilation</b>: nw-by-1 matrix containing the values <math>b_{w\_nw}</math>.</li> <li>• <b>ScalingTranslation</b>: ns-by-q matrix containing the values <math>c_{s\_ns}</math>.</li> <li>• <b>WaveletTranslation</b>: nw-by-q matrix containing the values <math>c_{w\_nw}</math>.</li> <li>• <b>ScalingCoef</b>: ns-by-1 vector containing the values <math>a_{s\_ns}</math>.</li> <li>• <b>WaveletCoef</b>: nw-by-1 vector containing the values <math>a_{w\_nw}</math>.</li> <li>• <b>OutputOffset</b>: scalar <math>d</math>.</li> </ul>

Property Name	Description
Options	<p>Structure containing the following fields that affect the initial model:</p> <ul style="list-style-type: none"> <li>• <b>FinestCell</b>: Integer or string specifying the minimum number of data points in the smallest cell. A <i>cell</i> is the area covered by the significantly nonzero portion of a wavelet. Default: 'auto', which computes the value from the data.</li> <li>• <b>MinCells</b>: Integer specifying the minimum number of cells in the partition. Default: 16.</li> <li>• <b>MaxCells</b>: Integer specifying the maximum number of cells in the partition. Default: 128.</li> <li>• <b>MaxLevels</b>: Integer specifying the maximum number of wavelet levels. Default: 10.</li> <li>• <b>DilationStep</b>: Real scalar specifying the dilation step size. Default: 2.</li> <li>• <b>TranslationStep</b>: Real scalar specifying the translation step size. Default: 1.</li> </ul>

## Algorithms

When the `idnlarx` property `Focus` is 'Prediction', `wavenet` uses a fast, noniterative technique for estimating parameters. Successive refinements after the first estimation use an iterative algorithm.

When the `idnlarx` property `Focus`='Simulation', `wavenet` uses an iterative technique for estimating parameters.

To always use noniterative or iterative algorithm, specify the `IterWavenet` algorithm property of the `idnlarx` class.

## Examples

Use `wavenet` to specify the nonlinear estimator in nonlinear ARX and Hammerstein-Wiener models. For example:

```
m=nlarx(Data,Orders,wavenet);
```

# wavenet

---

## See Also

[nlarx](#) | [nlhw](#)

**Purpose** Reorder states in state-space models

**Syntax** `sys = xperm(sys,P)`

**Description** `sys = xperm(sys,P)` reorders the states of the state-space model `sys` according to the permutation `P`. The vector `P` is a permutation of `1:NX`, where `NX` is the number of states in `sys`. For information about creating state-space models, see `ss` and `dss`.

**Examples** Order the states in the `ssF8` model in alphabetical order.

**1** Load the `ssF8` model by typing the following commands:

```
load ltiexamples
ssF8
```

These commands return:

```
a =
      PitchRate  Velocity      AOA  PitchAngle
PitchRate      -0.7    -0.0458   -12.2      0
Velocity        0     -0.014   -0.2904   -0.562
AOA              1     -0.0057   -1.4      0
PitchAngle      1      0          0          0

b =
      Elevator  Flaperon
PitchRate     -19.1    -3.1
Velocity     -0.0119  -0.0096
AOA          -0.14   -0.72
PitchAngle    0      0

c =
      PitchRate  Velocity      AOA  PitchAngle
FlightPath      0      0        -1      1
Acceleration    0      0     0.733      0
```

```
d =
      Elevator  Flaperon
FlightPath      0      0
Acceleration  0.0768  0.1134
```

Continuous-time model.

- 2** Order the states in alphabetical order by typing the following commands:

```
[y,P]=sort(ssF8.StateName);
sys=xperm(ssF8,P)
```

These commands return:

```
a =
      AOA  PitchAngle  PitchRate  Velocity
AOA      -1.4         0         1     -0.0057
PitchAngle  0         0         1         0
PitchRate  -12.2        0        -0.7     -0.0458
Velocity   -0.2904     -0.562        0     -0.014
```

```
b =
      Elevator  Flaperon
AOA      -0.14     -0.72
PitchAngle  0         0
PitchRate  -19.1    -3.1
Velocity   -0.0119  -0.0096
```

```
c =
      AOA  PitchAngle  PitchRate  Velocity
FlightPath  -1         1         0         0
Acceleration 0.733        0         0         0
```

```
d =
      Elevator  Flaperon
FlightPath      0      0
Acceleration  0.0768  0.1134
```



Continuous-time model.

The states in ssF8 now appear in alphabetical order.

**See Also**

ss | dss

# zero

---

**Purpose** Zeros and gain of SISO dynamic system

**Syntax**  
`z = zero(sys)`  
`[z,gain] = zero(sys)`  
`[z,gain] = zero(sysarr,J1,...,JN)`

**Description** `z = zero(sys)` returns the zeros of the single-input, single-output (SISO) dynamic system model, `sys`.  
`[z,gain] = zero(sys)` also returns the overall gain of `sys`.  
`[z,gain] = zero(sysarr,J1,...,JN)` returns the zeros and gain of the model with subscripts `J1, ..., JN` in the model array `sysarr`.

**Input Arguments** **sys**  
SISO dynamic system model.

If `sys` has internal delays, `zero` sets all internal delays to zero, creating a zero-order Padé approximation. This approximation ensures that the system has a finite number of zeros. `zero` returns an error if setting internal delays to zero creates singular algebraic loops.

**sysarr**  
Array of dynamic system models.

**J1,...,JN**  
Indices identifying the model `sysarr(J1, ..., JN)` in the array `sysarr`.

**Output Arguments** **z**  
Column vector containing the locations of zeros in `sys`. The zero locations are expressed in the reciprocal of the time units of `sys`. For example, the zeros are in units of 1/minutes if the `TimeUnit` property of `sys` is minutes.

**gain**

Gain of sys (in the zero-pole-gain sense).

## Examples

Calculate the zero locations and overall gain of the transfer function

$$H(s) = \frac{4.2s^2 + 0.25s - 0.004}{s^2 + 9.6s + 17}$$

```
H = tf([4.2,0.25,-0.004],[1,9.6,17]);
[z,gain] = zero(H)
```

```
z =
    -0.0726
     0.0131
```

```
gain =
    4.2000
```

The zero locations are expressed in radians per second, because the time unit of the transfer function (`H.TimeUnit`) is seconds. Change the model time units, and `zero` returns pole locations relative to the new unit.

```
H = chgTimeUnit(H,'minutes');
[z,gain] = zero(H)
```

```
z =
   -4.3581
    0.7867
```

```
gain =
    4.2000
```

# zero

---

**Alternatives** To calculate the transmission zeros of a multi-input, multi-output system, use `tzero`.

**See Also** `pole` | `pzmap` | `tzero`

<b>Purpose</b>	Generate z-plane grid of constant damping factors and natural frequencies
<b>Syntax</b>	<pre>zgrid zgrid(z,wn) zgrid([],[])</pre>
<b>Description</b>	<p><code>zgrid</code> generates, for root locus and pole-zero maps, a grid of constant damping factors from zero to one in steps of 0.1 and natural frequencies from zero to <math>\pi</math> in steps of <math>\pi/10</math>, and plots the grid over the current axis. If the current axis contains a discrete z-plane root locus diagram or pole-zero map, <code>zgrid</code> draws the grid over the plot without altering the current axis limits.</p> <p><code>zgrid(z,wn)</code> plots a grid of constant damping factor and natural frequency lines for the damping factors and normalized natural frequencies in the vectors <code>z</code> and <code>wn</code>, respectively. If the current axis contains a discrete z-plane root locus diagram or pole-zero map, <code>zgrid(z,wn)</code> draws the grid over the plot. The frequency lines for unnormalized (true) frequencies can be plotted using</p> <pre>zgrid(z,wn/Ts)</pre> <p>where <code>Ts</code> is the sample time.</p> <p><code>zgrid([],[])</code> draws the unit circle.</p> <p>Alternatively, you can select <b>Grid</b> from the right-click menu to generate the same z-plane grid.</p>

**Examples** Plot z-plane grid lines on the root locus for the system

$$H(z) = \frac{2z^2 - 3.4z + 1.5}{z^2 - 1.6z + 0.8}$$

by typing

```
H = tf([2 -3.4 1.5],[1 -1.6 0.8],-1)
```

# zgrid

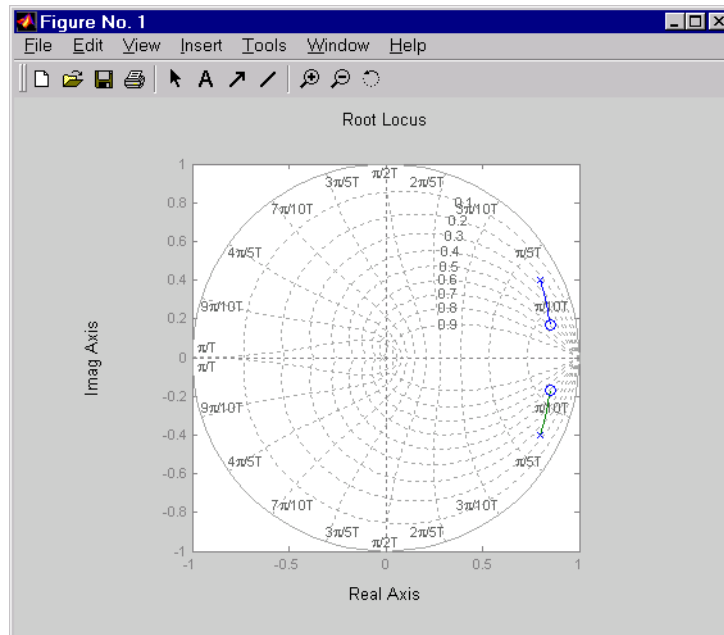
Transfer function:  
 $2z^2 - 3.4z + 1.5$

-----  
 $z^2 - 1.6z + 0.8$

Sampling time: unspecified

To see the z-plane grid on the root locus plot, type

```
rlocus(H)
zgrid
axis('square')
```



**See Also**

[pzmap](#) | [rlocus](#) | [sgrid](#)

**Purpose**

Access zero-pole-gain data

**Syntax**

```
[z,p,k] = zpkdata(sys)
[z,p,k,Ts,Td] = zpkdata(sys)
[z,p,k,Ts,covp,covk] = zpkdata(sys)
```

**Description**

`[z,p,k] = zpkdata(sys)` returns the zeros  $z$ , poles  $p$ , and gain(s)  $k$  of the zero-pole-gain model `sys`. The outputs  $z$  and  $p$  are cell arrays with the following characteristics:

- $z$  and  $p$  have as many rows as outputs and as many columns as inputs.
- The  $(i, j)$  entries  $z\{i, j\}$  and  $p\{i, j\}$  are the (column) vectors of zeros and poles of the transfer function from input  $j$  to output  $i$ .

The output  $k$  is a matrix with as many rows as outputs and as many columns as inputs such that  $k(i, j)$  is the gain of the transfer function from input  $j$  to output  $i$ . If `sys` is a transfer function or state-space model, it is first converted to zero-pole-gain form using `zpk`.

For SISO zero-pole-gain models, the syntax

```
[z,p,k] = zpkdata(sys, 'v')
```

forces `zpkdata` to return the zeros and poles directly as column vectors rather than as cell arrays (see example below).

`[z,p,k,Ts,Td] = zpkdata(sys)` also returns the sample time  $T_s$ .

`[z,p,k,Ts,covp,covk] = zpkdata(sys)` also returns the covariances of the zeros, poles and gain of the identified model `sys`. `covz` is a cell array such that `covz{ky,ku}` contains the covariance information about the zeros in the vector `z{ky,ku}`. `covz{ky,ku}` is a 3-D array of dimension 2-by-2-by- $N_z$ , where  $N_z$  is the length of `z{ky,ku}`, 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. `covp` has a similar relationship to `p`. `covk` is a matrix containing the variances of the elements of `k`.

You can access the remaining LTI properties of `sys` with `get` or by direct referencing, for example,

```
sys.Ts
sys.inputname
```

## Examples

### Example 1

Given a zero-pole-gain model with two outputs and one input

```
H = zpk([0];[-0.5]},{[0.3];[0.1+i 0.1-i]],[1;2],-1)
Zero/pole/gain from input to output...
```

```
          z
#1:  -----
      (z-0.3)

          2 (z+0.5)
#2:  -----
      (z^2 - 0.2z + 1.01)
```

Sampling time: unspecified

you can extract the zero/pole/gain data embedded in `H` with

```
[z,p,k] = zpkdata(H)
z =
     [      0]
     [-0.5000]
p =
     [ 0.3000]
     [2x1 double]
k =
     1
     2
```

To access the zeros and poles of the second output channel of `H`, get the content of the second cell in `z` and `p` by typing



```
z{2,1}
ans =
    -0.5000
p{2,1}
ans =
    0.1000+ 1.0000i
    0.1000- 1.0000i
```

## Example 2

Extract the ZPK matrices and their standard deviations for a 2-input, 1 output identified transfer function.

```
load iddata7
```

```
transfer function model
```

```
sys1 = tfest(z7, 2, 1, 'InputDelay',[1 0]);
```

```
an equivalent process model
```

```
sys2 = procest(z7, {'P2UZ', 'P2UZ'}, 'InputDelay',[1 0]);
```

```
1, p1, k1, ~, dz1, dp1, dk1] = zpkdata(sys1);
[z2, p2, k2, ~, dz2, dp2, dk2] = zpkdata(sys2);
```

Use `iopzplot` to visualize the pole-zero locations and their covariances

```
h = iopzplot(sys1, sys2);
showConfidence(h)
```

## See Also

```
get | ssdata | tfdata | zpk
```

**zpkdata**

---

# Block Reference

---

Data Import and Processing (p. 3-2)	Blocks for transferring data between System Identification Toolbox and Simulink software
Linear Model Identification (p. 3-3)	Blocks for estimating linear models during simulation in Simulink software
Simulation (p. 3-4)	Blocks for simulating and predicting linear and nonlinear model output in Simulink software

## **Data Import and Processing**

IDDATA Sink

Export iddata object to MATLAB workspace

IDDATA Source

Import iddata object from MATLAB workspace

## Linear Model Identification

AR Estimator	Estimate parameters of AR model from scalar time series in Simulink software returning <code>idpoly</code> object
ARMAX Estimator	Estimate parameters of ARMAX model from SISO data in Simulink software returning <code>idpoly</code> object
ARX Estimator	Estimate parameters of ARX model from SISO data in Simulink software returning <code>idpoly</code> object
BJ Estimator	Estimate parameters of Box-Jenkins model from SISO data in Simulink software returning <code>idpoly</code> object
OE Estimator	Estimate parameters of Output-Error model from SISO data in Simulink software returning <code>idpoly</code> object
PEM Estimator	Estimate generic input-output polynomial model parameters from SISO data using iterative prediction-error minimization method

## Simulation

IDMODEL Model	Simulate <code>idmodel</code> 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

# Blocks — Alphabetical List

---

# AR Estimator

---

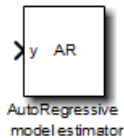
## Purpose

Estimate parameters of AR model from scalar time series in Simulink software returning `idpoly` object

## Library

System Identification Toolbox

## Description



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 `arOptions`.

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 measured output and the model output.

## Model Definition

The AR model is defined, as follows:

$$y(t) + a_1 y(t-1) + \dots + a_{n_a} y(t-n_a) = e(t)$$

where

- $y(t)$  is the output at time  $t$ .
- $a_1 \dots a_n$  are the parameters to be estimated from the data.
- $n_a$  is the number of poles of the system.
- $y(t-1) \dots y(t-n_a)$  are the previous outputs on which the current output depends.
- $e(t)$  is white-noise disturbance.

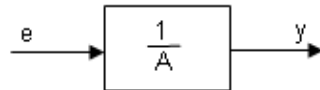


The AR model can be written compactly for a single output  $y(t)$  using the following notation:

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

where  $A(q) = 1 + a_1q^{-1} + \dots + a_{n_a}q^{-n_a}$  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

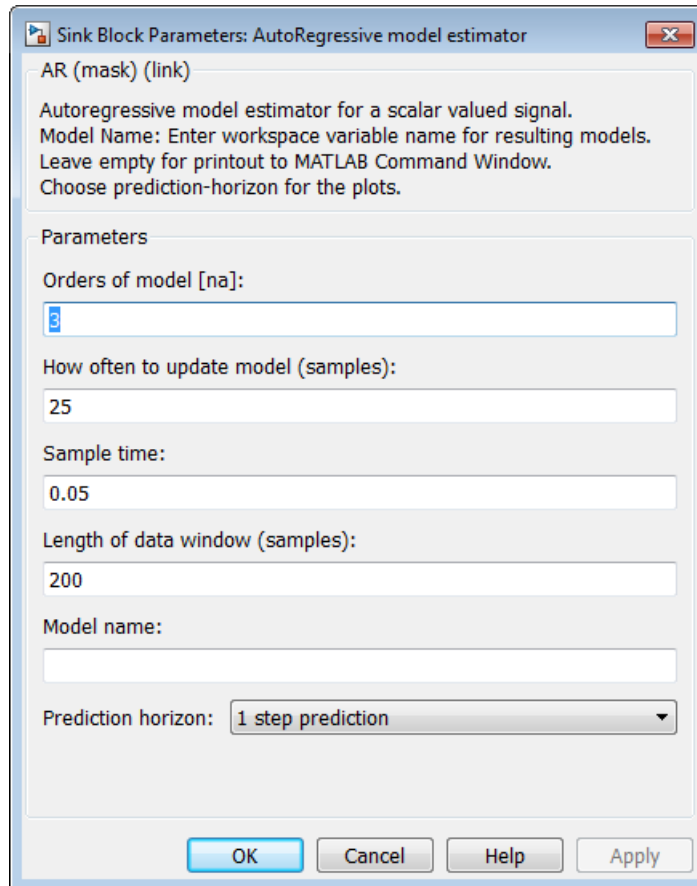
Time-series signal.

## 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.

# AR Estimator



## 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.

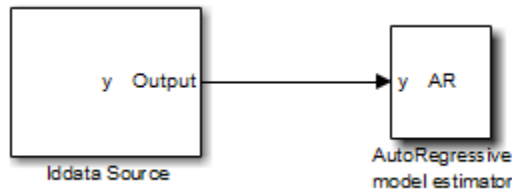
# AR Estimator

---

```
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 appear in the MATLAB Command Window every 25 samples.

## See Also

## Related Commands

ar  
idpoly

## Topics in the System Identification Toolbox User's Guide

“Estimating AR and ARMA Models”

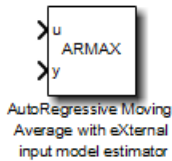
## Purpose

Estimate parameters of ARMAX model from SISO data in Simulink software returning `idpoly` object

## Library

System Identification Toolbox

## Description



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 `armaxOptions`.

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 measured output and the model output.

## Model Definition

The ARMAX model is defined, as follows:

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

where

- $y(t)$  is the output at time  $t$ .
- $a_1 \dots a_n$ ,  $b_1 \dots b_n$ , and  $c_1 \dots c_n$  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.

# ARMAX Estimator

---

- $n_k$  is the number of input samples that occur before the inputs affecting the current output.
- $y(t-1)\dots y(t-n_a)$  are the previous outputs on which the current output depends.
- $u(t-n_k)\dots u(t-n_k-n_b+1)$  are the previous inputs on which the current output depends.
- $e(t), e(t-1), \dots, e(t-n_c)$  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

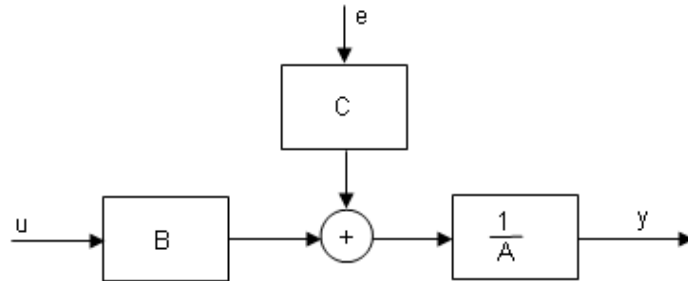
$$A(q) = 1 + a_1q^{-1} + \dots + a_{n_a}q^{-n_a}$$

$$B(q) = b_1 + b_2q^{-1} + \dots + b_{n_b}q^{-n_b+1}$$

$$C(q) = 1 + c_1q^{-1} + \dots + c_{n_c}q^{-n_c}$$

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.

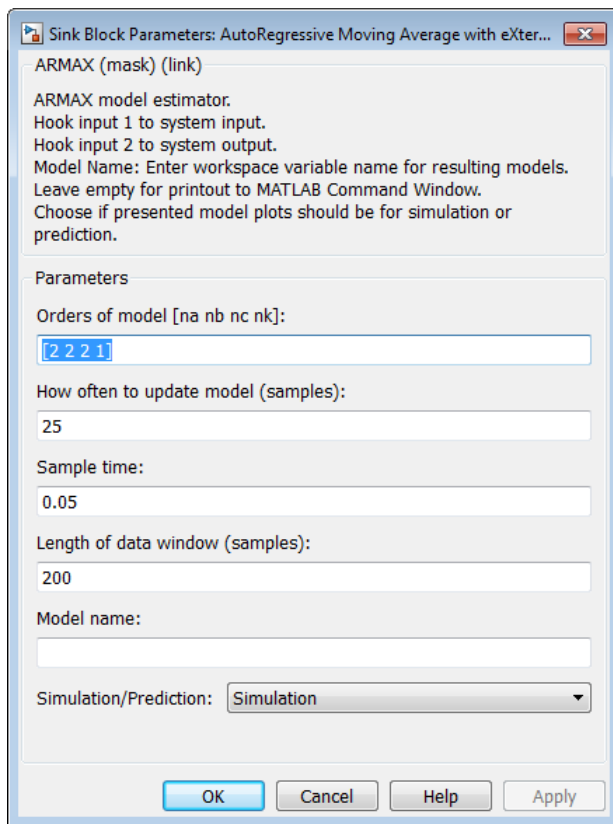
## Output

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.

# ARMAX Estimator



## 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  $n_k$  is 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 to use the ARMAX Estimator block in a Simulink model.

# ARMAX Estimator

---

- 1 Generate sample input and output data.

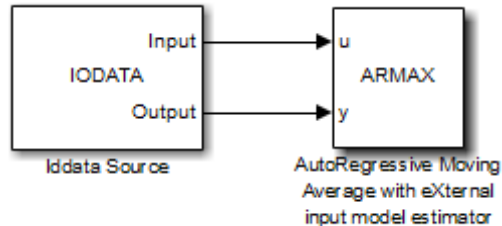
```
u = sin([1:300]') + 0.6*(rand(300,1)-0.5);  
y = cos(u) + 0.1*rand(300,1);  
IODEATA = iddata(y,u,1);
```

- 2 Create a new Simulink model, as follows.

Add the IODEATA Source block and specify IODEATA in the **Iddata object** field of the IODEATA Source block parameters dialog box.

Add the ARMAX Estimator block to the model and set the model orders to [4 4 4 0] and set the sample time to 1.

Connect the Input and Output ports of the IODEATA Source block to the u and y ports of the ARMAX Estimator block, respectively. Set the simulation end time to 300 seconds.



- 3 Run the simulation.

The estimated models display in the MATLAB Command Window every 25 samples.

## See Also

## Related Commands

armax  
idpoly

## **Topics in the System Identification Toolbox User's Guide**

“Identifying Input-Output Polynomial Models”

# ARX Estimator

---

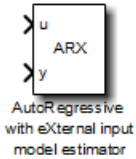
## Purpose

Estimate parameters of ARX model from SISO data in Simulink software returning `idpoly` object

## Library

System Identification Toolbox

## Description



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 `arxOptions`.

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 measured output and the model output.

## Model Definition

The ARX model is defined, as follows:

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

where

- $y(t)$  is the output at time  $t$ .
- $a_1 \dots a_n$  and  $b_1 \dots b_n$  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.
- $y(t-1) \dots y(t-n_a)$  are the previous outputs on which the current output depends.

- $u(t - n_k) \dots u(t - n_k - n_b + 1)$  are the previous inputs on which the current output depends.
- $e(t)$  is a white-noise disturbance value.

The ARX model can also be written in a compact way using the following notation:

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

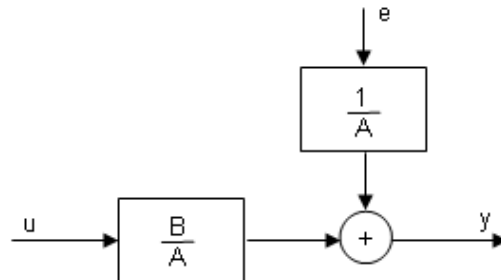
where

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

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

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.

# ARX Estimator

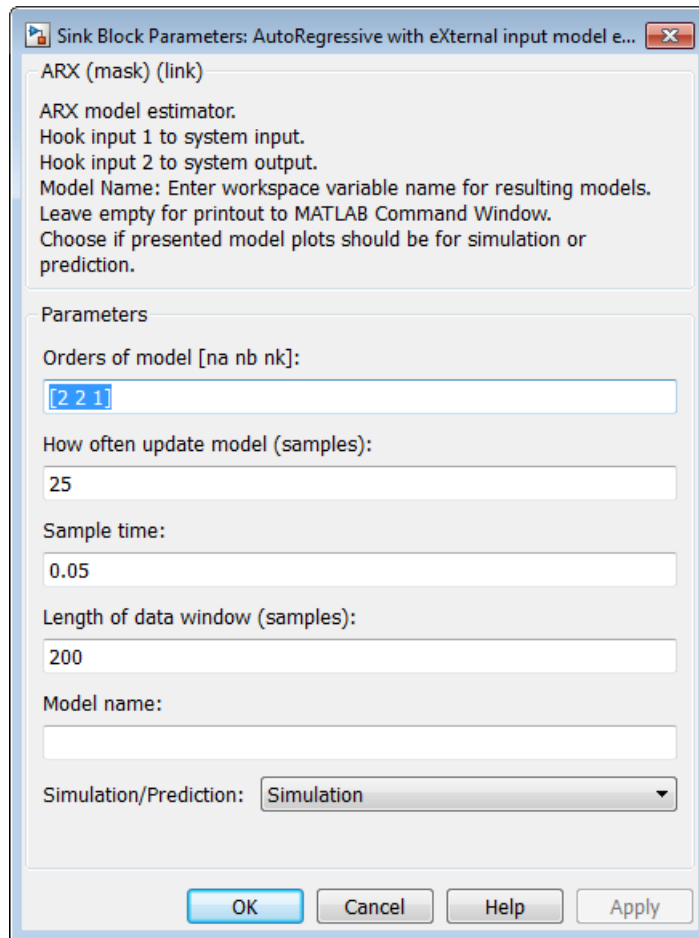
---

## 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  $n_k$  is input-output delay, respectively.

### 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.

## **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

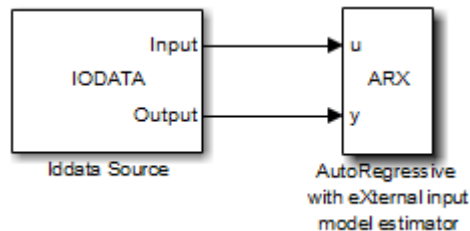
This example shows how you can use the ARX Estimator block in a Simulink model.

- 1 Specify the data from `iddata1.mat` for estimation:

```
load iddata1;  
IODEATA = z1;
```

- 2 Create a new Simulink model, as follows:

- Add the IDDATA Source block and specify IODEATA in the **Iddata object** field of the IDDATA Source block parameters dialog box.
- Add the ARX Estimator block to the model. Set the sample time in the block to 0.1 seconds and the simulation end time to 30 seconds.
- 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.

## See Also

### Related Commands

`arx`  
`idpoly`

## **Topics in the System Identification Toolbox User's Guide**

“Identifying Input-Output Polynomial Models”

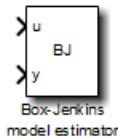
## Purpose

Estimate parameters of Box-Jenkins model from SISO data in Simulink software returning `idpoly` object

## Library

System Identification Toolbox

## Description



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 `bjOptions`.

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 measured 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

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

$$F(q) = 1 + f_1 q^{-1} + \dots + f_{n_f} q^{-n_f}$$

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

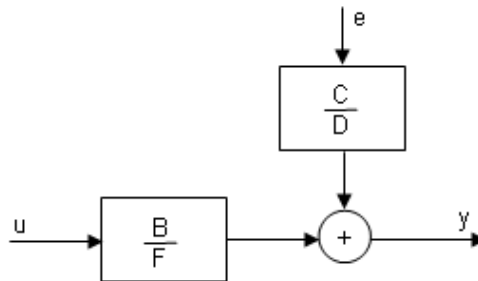
$$D(q) = 1 + d_1 q^{-1} + \dots + d_{n_d} q^{-n_d}$$

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.

# BJ Estimator

---



## Input

The block accepts two inputs, corresponding to the measured input-output data for estimating the model.

First input: Input signal.

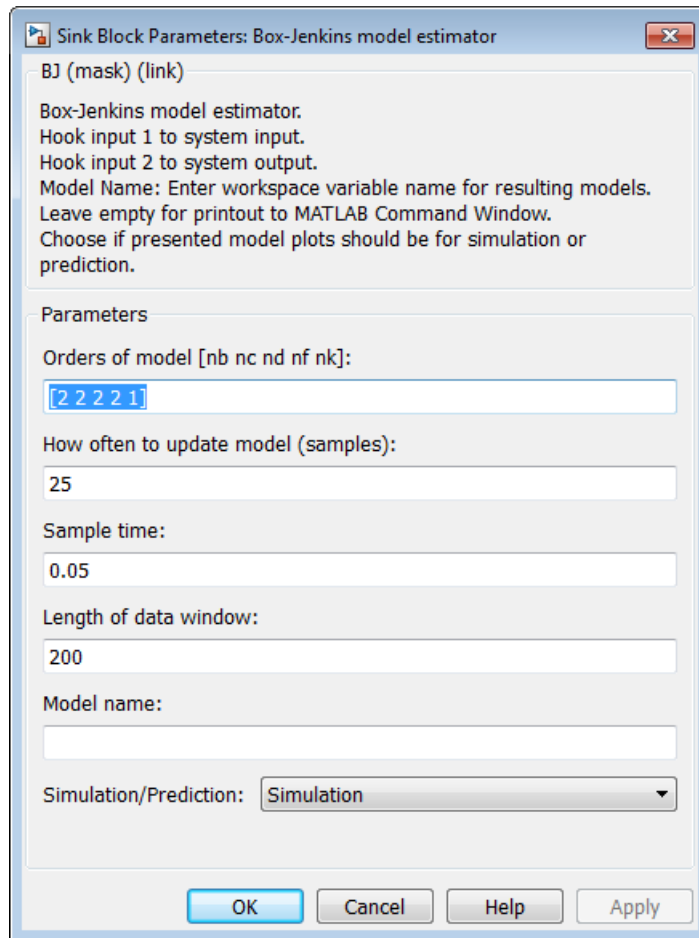
Second input: Output signal.

## 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 [nb nc nd nf nk]

Integers  $n_b$ ,  $n_c$ ,  $n_d$ , and  $n_f$  specify the number of  $B$ ,  $C$ ,  $D$ , 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

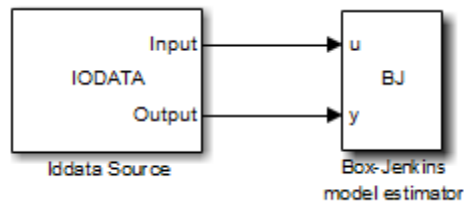
This example shows how you can use the BJ Estimator block in a Simulink model.

- 1 Specify the data in `iddata1.mat` for estimation:

```
load iddata1;  
IODATA = z1;
```

- 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. Set the sample time in the block to 0.1 seconds and the simulation end time to 30 seconds.
- 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 appear in the MATLAB Command Window every 25 samples.

# BJ Estimator

---

## See Also

## Related Commands

`bj`

`idpoly`

## Topics in the System Identification Toolbox User's Guide

“Identifying Input-Output Polynomial Models”



**Purpose**

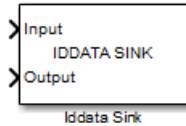
Export iddata object to MATLAB workspace

**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.

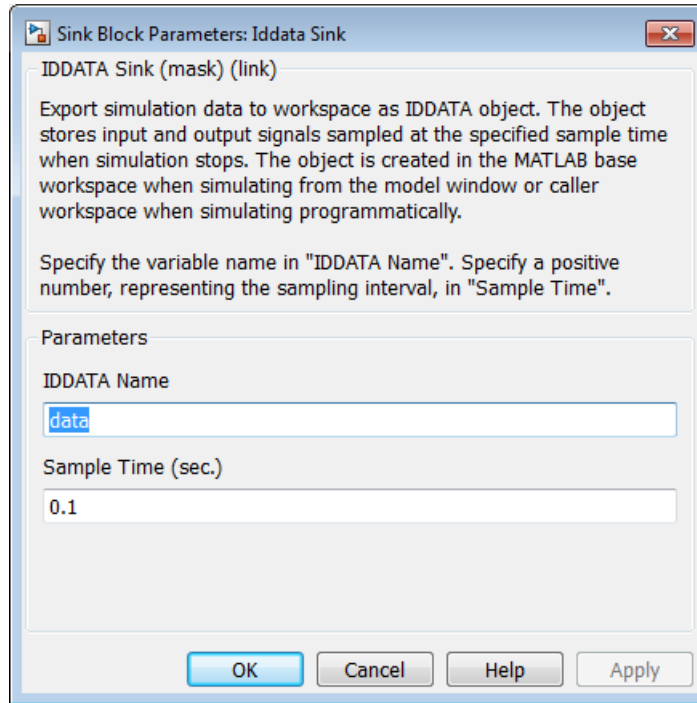
**Output**

None.

# IDDATA Sink

---

## 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**

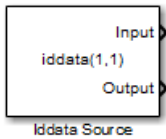
IDDATA Source

**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.

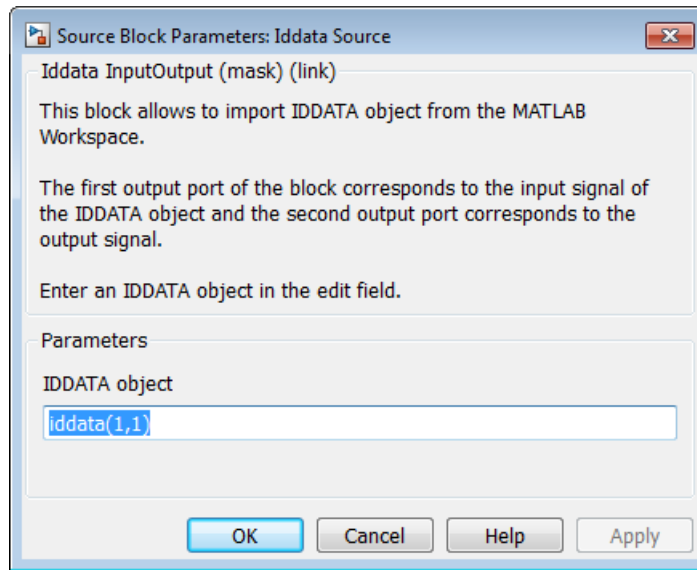
**Output**

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.

# IDDATA Source

---



## Dialog Box

### Iddata object

Name of the `iddata` object imported from 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

**Purpose**

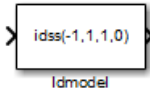
Simulate `idmodel` object in Simulink software

**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.

---

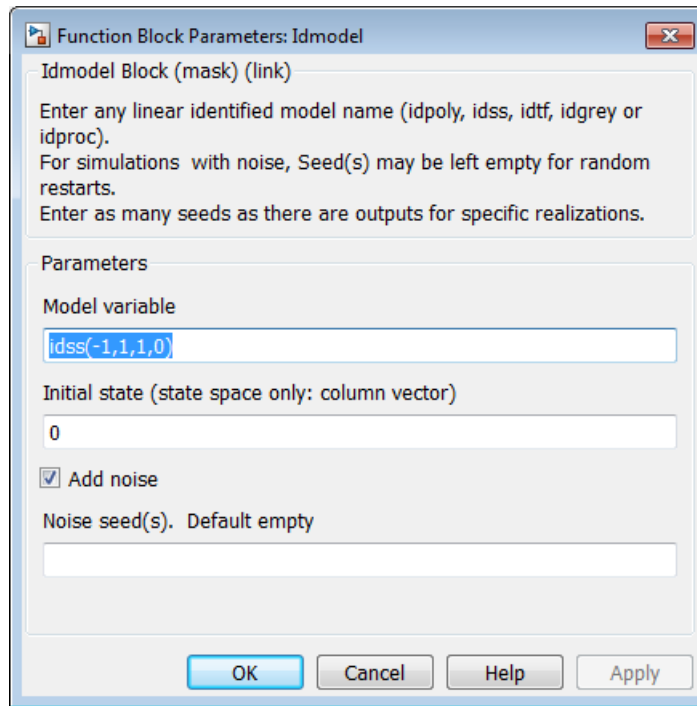
**Input**

Input signal to the model.

**Output**

Simulated output from the model.

# IDMODEL 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(idParametric)` command to get the initial state values that produce the best fit between the model output and the measured output signal.

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

# IDMODEL Model

---

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(idParametric)`

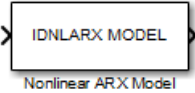
`idmodel`



**Purpose** Simulate nonlinear ARX model in Simulink software

**Library** System Identification Toolbox

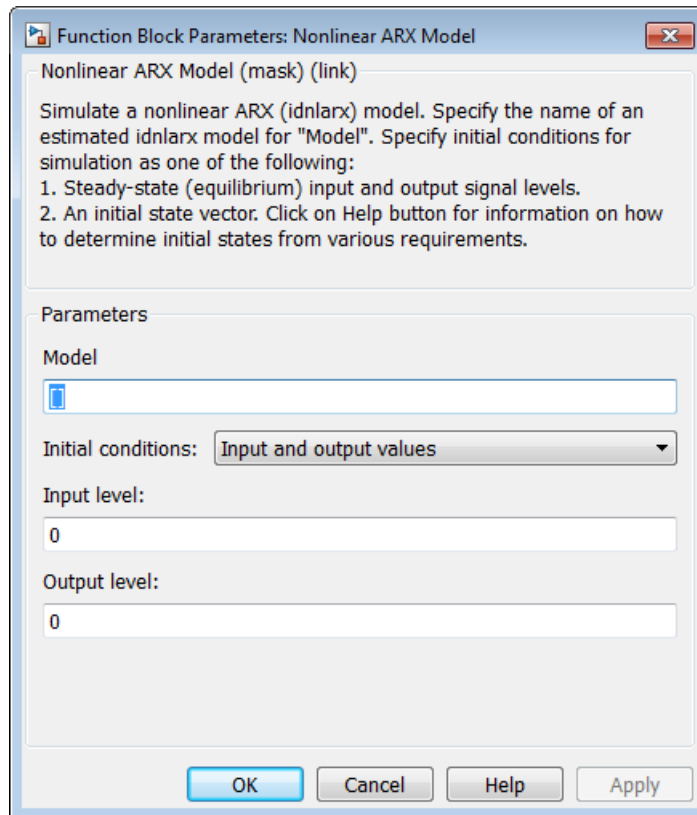
**Description** The IDNLARX Model block simulates a nonlinear ARX (idnlarx) model for time-domain input and output data.



**Input** Input signal to the model.

**Output** Simulated output from the model.

# IDNLARX 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 around 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 point 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.

# IDNLARX Model

---

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 to the measured output of the system.

- 1 Load the sample data.

```
load twotankdata
```

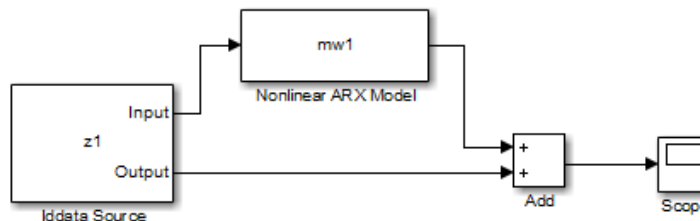
- 2 Create a data object from sample data.

```
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.

## Example 2

In this example, you reduce the difference between the measured and simulated responses. 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.

## See Also

## Related Commands

```
findop(idnlarx)  
findstates(idnlarx)  
idnlarx
```

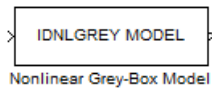
## **Topics in the System Identification Toolbox User's Guide**

“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.

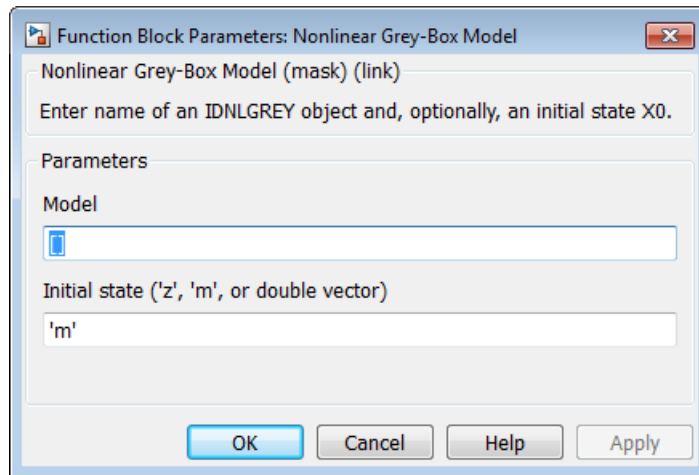


### Input

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.

# IDNLGREY 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

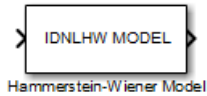
“Estimating Nonlinear Grey-Box Models”



**Purpose** Simulate Hammerstein-Wiener model in Simulink software

**Library** System Identification Toolbox

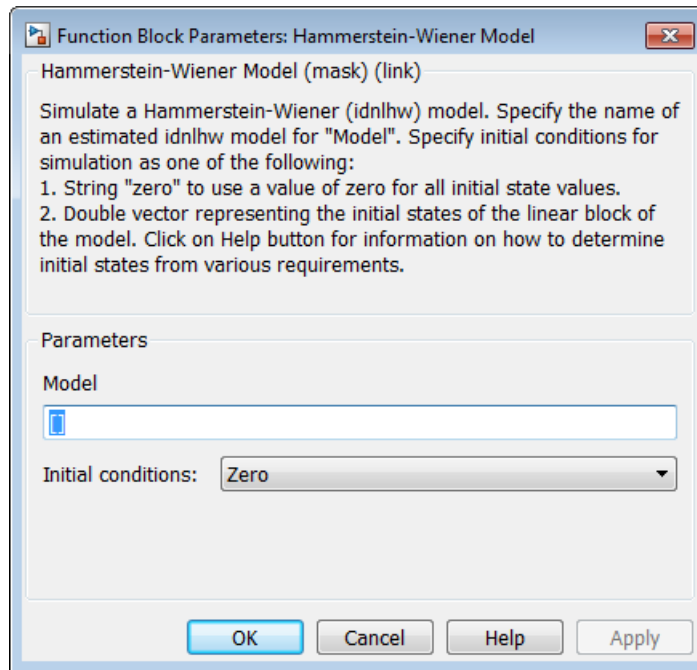
**Description** The IDNLHW Model block simulates a Hammerstein-Wiener (idn1hw) model for time-domain input and output data.



**Input** Input signal to the model.

**Output** Simulated output from the model.

# IDNLHW 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 around 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 point 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);
```

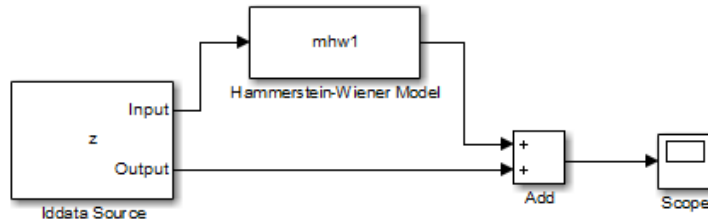
- 3 Estimate a Hammerstein-Wiener model.

```
mhw1 = nlhw(z,[1 5 3],pwlinear,pwlinear);
```

# IDNLHW Model

---

- 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.

## Example 2

In this example, you reduce the difference between the measured and simulated responses using suitable initial state values. 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.

## See Also

## Related Commands

```
findop(idnlhw)
```

```
findstates(idnlhw)
```

```
idnlhw
```

## Topics in the System Identification Toolbox User's Guide

“Identifying Hammerstein-Wiener Models”

# OE Estimator

---

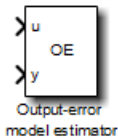
## Purpose

Estimate parameters of Output-Error model from SISO data in Simulink software returning `idpoly` object

## Library

System Identification Toolbox

## Description



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 `oeOptions`.

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 measured output and the model output.

## Model Definition

The output-error model is defined, as follows:

$$w(t) + f_1 w(t-1) + \dots + f_{n_f} w(t-n_f) = b_1 u(t-1) + \dots + b_{n_b} u(t-n_b - n_b + 1)$$
$$y(t) = w(t) + e(t)$$

where

- $w$  is the undisturbed output.
- $y(t)$  is the output at time  $t$ .
- $f_1 \dots f_{n_f}$  and  $b_1 \dots b_{n_b}$  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.
- $u(t - n_k) \dots u(t - n_k - n_b + 1)$  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:

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

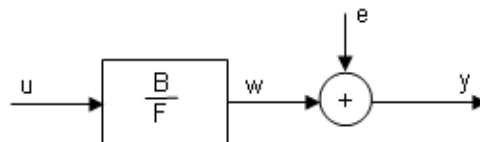
where

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

$$F(q) = 1 + f_1 q^{-1} + \dots + f_{n_f} q^{-n_f}$$

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.

# OE Estimator

---

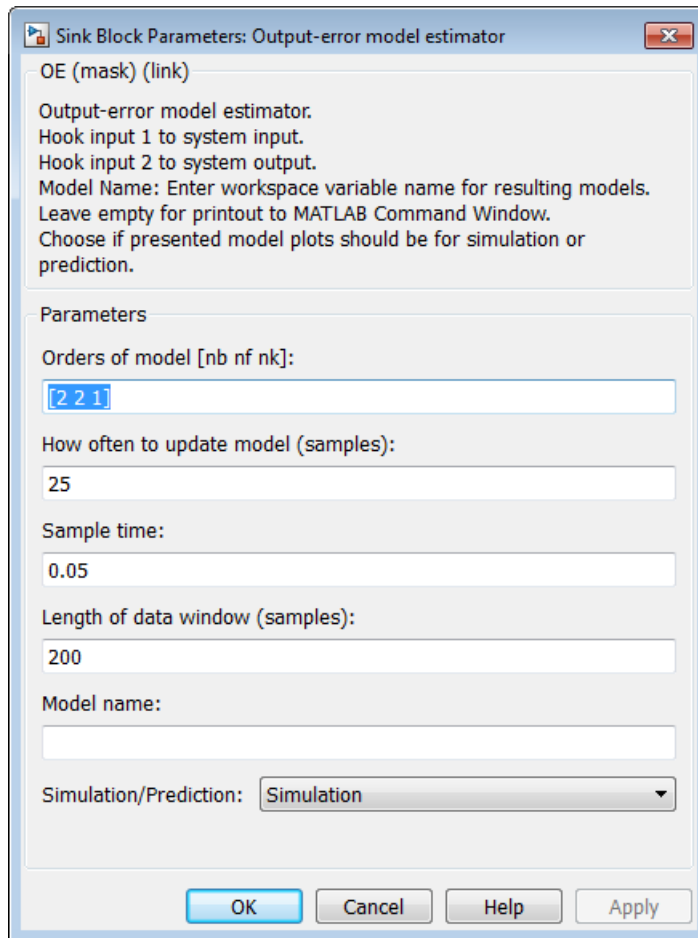
## Output

The OE Estimator block outputs a sequence of multiple models (idpoly), estimated at regular intervals during the simulation.

The **Length of 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 [nb nf nk]

Integers  $n_b$ ,  $n_f$ , and  $n_k$  specify the number of  $B$  and  $F$  model parameters and  $n_k$  is the input-output delay, respectively.

### How often to 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

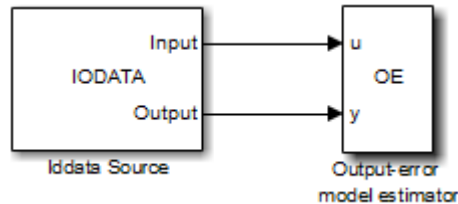
This example shows how you can use the OE Estimator block in a Simulink model.

- 1 Specify the data from `iddata1.mat` for estimation:

```
load iddata1;
IODEATA = z1;
```

- 2 Create a new Simulink model, as follows:

- Add the IDDATA Source block and specify IODEATA in the **Iddata object** field of the IDDATA Source block parameters dialog box.
- Add the OE Estimator block to the model. Set sample time in the block to 0.1 seconds and the simulation end time to 30 seconds.
- 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 appear in the MATLAB Command Window every 25 samples.

## See Also

### Related Commands

`oe`  
`idpoly`

## **Topics in the System Identification Toolbox User's Guide**

“Identifying Input-Output Polynomial Models”

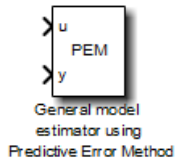
## Purpose

Estimate generic input-output polynomial model parameters from SISO data using iterative prediction-error minimization method

## Library

System Identification Toolbox

## Description



The PEM Estimator block estimates linear input-output polynomial models in Simulink software.

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 measured 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  $a_1 \dots a_{n_a}$ ,  $b_1 \dots b_{n_b}$ ,  $f_1 \dots f_{n_f}$ ,  $c_1 \dots c_{n_c}$  and  $d_1 \dots d_{n_d}$  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.

# PEM Estimator

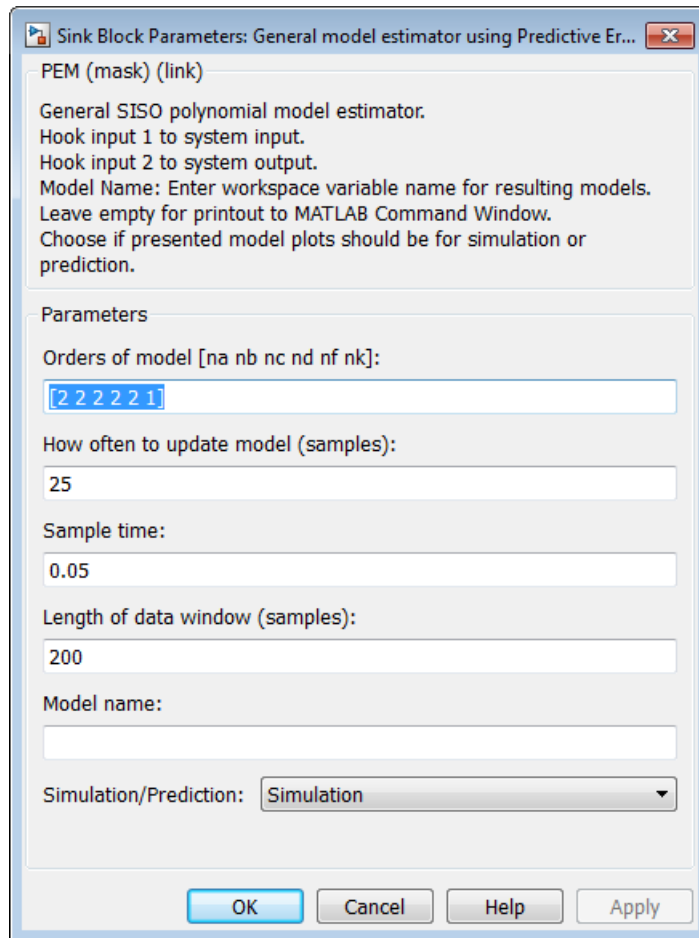
---

## 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  $n_k$  is 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 Specify data in `iddata1.mat` for estimation:

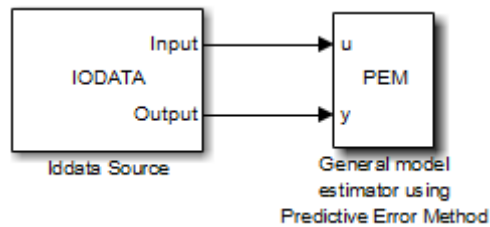
```
load iddata1;  
IODATA = z1;
```

- 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 block to the model. Set the sample time in the block to 0.1 seconds and the simulation end time to 30 seconds.

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.

## See Also

## Related Commands

`idpoly`

`pem`

## **Topics in the System Identification Toolbox User's Guide**

“Identifying Input-Output Polynomial Models”

## A

adaptive noise canceling 2-756  
append 2-10

## B

bodemag (Bode magnitude plots) 2-92

## C

c2d 2-103  
cell array 2-249  
continuous-time  
    conversion to.. *See* conversion, model  
conversion, model  
    discrete to continuous (d2c) 2-158  
        with negative real poles 2-162  
    resampling  
        discrete models 2-167

## D

d2c 2-158  
d2d 2-167  
dB to magnitude 2-178  
db2mag 2-178 2-570  
dcgain 2-179  
dead time. *See* delays  
delays  
    combining 2-964  
    existence of, test for 2-287  
    hasdelay 2-287  
Dirac impulse 2-486  
discretization  
    available methods 2-113 2-169

## E

evalfr 2-189

## F

filt 2-239  
first-order hold (FOH) 2-113  
FRD (frequency response data) objects  
    data 2-239  
    frdata 2-239  
frdata 2-239  
freqresp 2-241  
frequency response  
    at single frequency (evalfr) 2-189  
frequency response function 2-844

## G

gain  
    low frequency (DC) 2-179  
get 2-248

## H

hasdelay 2-287

## I

impulse 2-486  
impulse response 2-486  
input  
    Dirac impulse 2-486  
isempty 2-520  
isproper 2-521  
issiso 2-526

## L

lsim 2-560  
LTI properties  
    accessing property values (get) 2-248  
    displaying properties 2-248  
    property names 2-248 2-784  
    property values 2-248 2-784  
    setting 2-784

**M**

- magnitude to dB 2-570
- matched pole-zero 2-113
- MIMO 2-486
- model building
  - appending LTI models 2-10

**N**

- numerator
  - value 2-249
- nyquist 2-627

**O**

- operations on LTI models
  - append 2-10
  - diagonal building 2-10

**P**

- plotting
  - s-plane grid (sgrid) 2-801
  - z-plane grid (zgrid) 2-995
- pole 2-682
- pole-zero
  - map (pzmap) 2-742
- poles
  - computing 2-682
  - multiple 2-682
  - pole-zero map 2-742
  - s-plane grid (sgrid) 2-801
  - z-plane grid (zgrid) 2-995
- pzmap 2-742

**R**

- realization
  - state coordinate transformation 2-866
- resampling (d2d) 2-167

**S**

- sample time
  - resampling 2-167
- set 2-784
- simulation of linear systems.. *See* time response
- stability margins
  - pole 2-682
  - pzmap 2-742
- state
  - transformation 2-866
- state-space models
  - state order 2-989
- step response 2-901

**T**

- time response
  - impulse response (impulse) 2-486
  - MIMO 2-486
  - response to arbitrary inputs (lsim) 2-560
  - step response (step) 2-901
- totaldelay 2-964
- transfer functions
  - quick data retrieval (tfdata) 2-919
- transmission zeros.. *See* zeros
- triangle approximation 2-113
- Tustin approximation 2-113 2-169
  - with frequency prewarping 2-113 2-169
- tzero. . *See* zero

**Z**

- zero 2-992
- zero-order hold (ZOH) 2-113 2-169
- zero-pole-gain (ZPK) models
  - quick data retrieval (zpkdata) 2-997
- zeros
  - computing 2-992
  - pole-zero map 2-742
  - transmission 2-992