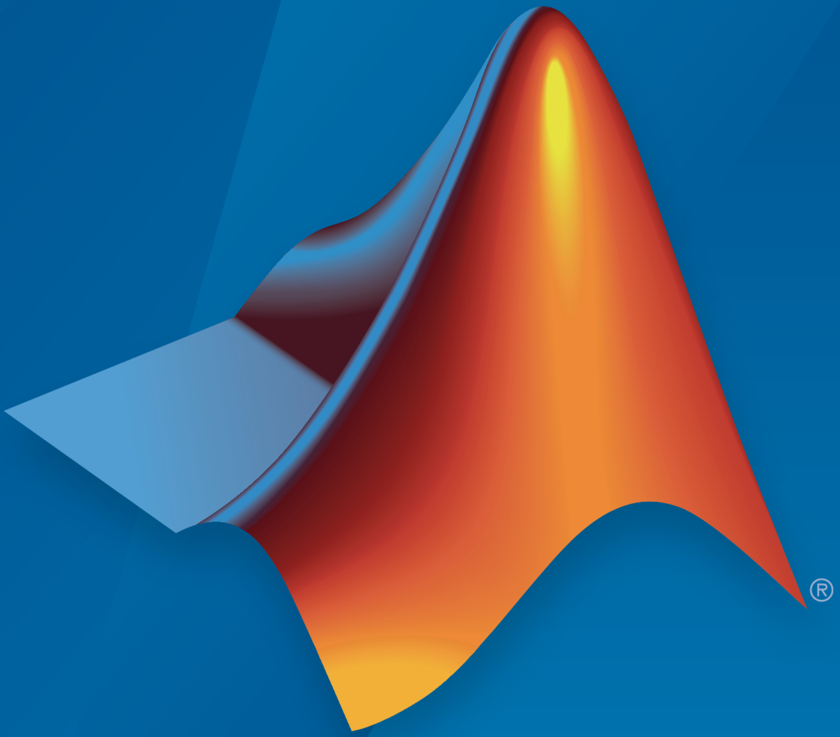
System Identification Toolbox™

Reference

Lennart Ljung



MATLAB® & SIMULINK®

R2016b

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

System Identification Toolbox™ Reference

© COPYRIGHT 1988–2016 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

September 2007	Online only	Revised for Version 7.1 (Release 2007b)
March 2008	Online only	Revised for Version 7.2 (Release 2008a)
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)
March 2013	Online only	Revised for Version 8.2 (Release 2013a)
September 2013	Online only	Revised for Version 8.3 (Release 2013b)
March 2014	Online only	Revised for Version 9.0 (Release 2014a)
October 2014	Online only	Revised for Version 9.1 (Release 2014b)
March 2015	Online only	Revised for Version 9.2 (Release 2015a)
September 2015	Online only	Revised for Version 9.3 (Release 2015b)
March 2016	Online only	Revised for Version 9.4 (Release 2016a)
September 2016	Online only	Revised for Version 9.5 (Release 2016b)

Contents

1 Functions – Alphabetical List

2 Blocks — Alphabetical List

1

Functions – Alphabetical List

absorbDelay

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

Absorb Time Delay into System Dynamics

Create a discrete-time transfer function that has a time delay.

```
z = tf('z',-1);
sysd = (-0.4*z -0.1)/(z^2 + 1.05*z + 0.08);
sysd.InputDelay = 3
```

```
sysd =
```

```
              -0.4 z - 0.1
  z^(-3) * -------------------
           z^2 + 1.05 z + 0.08

Sample time: unspecified
Discrete-time transfer function.
```

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 time delay into the system dynamics as poles at `z`= 0.

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

```
sysnd =

        -0.4 z - 0.1
  -------------------------
  z^5 + 1.05 z^4 + 0.08 z^3

Sample time: unspecified
Discrete-time transfer function.
```

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

Verify that `sysnd` has no input delay.

```
sysnd.InputDelay
```

```
ans =

     0
```

Convert Leading Structural Zeros of Polynomial Model to Regular Coefficients

Create a discrete-time polynomial model.

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

Convert `m` to a transfer function model.

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

```
sys =

  z^(-2) * (2 z^-1 + 3 z^-2)

Sample time: unspecified
Discrete-time transfer function.
```

The numerator of the transfer function, `sys`, is `[0 2 3]` and the transport delay, `sys.IODelay`, is 2. This is because the value of the B polynomial, `m.B`, has 3 leading zeros. The first fixed zero shows lack of feedthrough in the model. The two zeros after that are treated as input-output delays.

Use `absorbDelay` to treat the leading zeros as regular `B` coefficients.

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

```
sys2 =

  2 z^-3 + 3 z^-4

Sample time: unspecified
Discrete-time transfer function.
```

The numerator of `sys2` is `[0 0 0 2 3]` and transport delay is `0`. The model `m2` treats the leading zeros as regular coefficients by freeing their values. `m2.Structure.B.Free(2:3)` is `TRUE` while `m.Structure.B.Free(2:3)` is `FALSE`.

See Also

`hasdelay` | `pade` | `totaldelay`

Introduced in R2012a

advice

Analysis and recommendations for data or estimated linear models

Syntax

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

Description

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

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

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

- Does the model capture essential dynamics of the system and the disturbance characteristics?
- Is the model order higher than necessary?
- Is there potential output feedback in the validation data?

Input Arguments

`data`

Specify `data` as an `iddata` object.

`model`

Specify `model` as an `idtf`, `idgrey`, `idpoly`, `idproc`, or `idss` model object.

See Also

`detrend` | `feedback` | `iddata` | `pexcit`

Introduced before R2006a

addreg

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 `idnalrx` objects. For single-output models, *`regressors`* is an object array of regressors you create using `customreg` or `polyreg`, or a cell array of character vectors. For multiple-output models, *`regressors`* is 1-by-`ny` cell array of `customreg` objects or 1-by-`ny` cell array of cell arrays of character vectors. `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

Create nonlinear ARX model with standard regressors.

```
m1 = idnlarx([4 2 1],'wavenet','nlr',[1:3]);
```

Create model with additional custom regressors, specifed as a cell array of character vectors.

```
m2 = addreg(m1,{'y1(t-2)^2';'u1(t)*y1(t-7)'});
```

List all standard and custom regressors of `m2`.

```
getreg(m2)
```

```
Regressors:
    y1(t-1)
    y1(t-2)
    y1(t-3)
    y1(t-4)
    u1(t-1)
    u1(t-2)
    y1(t-2)^2
    u1(t)*y1(t-7)
```

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 `customreg` objects.

```
r1 = customreg(@(x)x^2,{'y1'},2)
```

```
Custom Regressor:
Expression: y1(t-2)^2
        Function: @(x)x^2
       Arguments: {'y1'}
          Delays: 2
      Vectorized: 0
    TimeVariable: 't'
```

```
r2 = customreg(@(x,y)x*y,{'u1','y1'},[0 7])
```

```
Custom Regressor:
Expression: u1(t)*y1(t-7)
        Function: @(x,y)x*y
       Arguments: {'u1'  'y1'}
          Delays: [0 7]
      Vectorized: 0
    TimeVariable: 't'
```

Create a model based on `m1` with custom regressors.

```
m2 = addreg(m1,[r1 r2]);
```

List all standard and custom regressors of `m2`.

```
getreg(m2)
```

```
Regressors:
    y1(t-1)
    y1(t-2)
    y1(t-3)
    y1(t-4)
    u1(t-1)
    u1(t-2)
    y1(t-2)^2
    u1(t)*y1(t-7)
```

More About

- "Identifying Nonlinear ARX Models"

See Also

`customreg` | `getreg` | `nlarx` | `polyreg`

Introduced in R2007a

aic

Akaike’s Information Criterion for estimated model

Syntax

```
value = aic(model)
value = aic(model1,...,modeln)
value = aic( ___ ,measure)
```

Description

`value = aic(model)` returns the normalized “Akaike's Information Criterion (AIC)” on page 1-15 value for the estimated model.

`value = aic(model1,...,modeln)` returns the normalized AIC values for multiple estimated models.

`value = aic( ___ ,measure)` specifies the type of AIC.

Examples

Compute Normalized Akaike's Information Criterion of Estimated Model

Estimate a transfer function model.

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

Compute the normalized Akaike's Information Criterion value.

```
value = aic(sys)
```

```
value =

    0.5453
```

The value is also computed during model estimation. Alternatively, use the `Report` property of the model to access this value.

```
sys.Report.Fit.nAIC
```

```
ans =

    0.5453
```

Compute Akaike's Information Criterion Metrics of Estimated Model

Estimate a transfer function model.

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

Compute the normalized Akaike's Information Criterion (AIC) value. This syntax is equivalent to `aic_raw = aic(sys)`.

```
aic_raw = aic(sys,'nAIC')
```

```
aic_raw =

    0.5453
```

Compute the raw AIC value.

```
aic_raw = aic(sys,'aic')
```

```
aic_raw =

   1.0150e+03
```

Compute the sample-size corrected AIC value.

```
aic_c = aic(sys,'AICc')
```

```
aic_c =

   1.0153e+03
```

Compute the Bayesian Information Criteria (BIC) value.

```
bic = aic(sys,'BIC')
```

```
bic =

   1.0372e+03
```

These values are also computed during model estimation. Alternatively, use the `Report.Fit` property of the model to access these values.

```
sys.Report.Fit
```

```
ans =

  struct with fields:

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

Pick Model with Optimal Tradeoff Between Accuracy and Complexity Using AICc Criterion

Estimate multiple Output-Error (OE) models and use the small sample-size corrected Akaike's Information Criterion (AICc) value to pick the one with optimal tradeoff between accuracy and complexity.

Load the estimation data.

```
load iddata2
```

Specify model orders varying in 1:4 range.

```
nf = 1:4;
nb = 1:4;
nk = 0:4;
```

Estimate OE models with all possible combinations of chosen order ranges.

```
NN = struc(nf,nb,nk);
models = cell(size(NN,1),1);
for ct = 1:size(NN,1)
   models{ct} = oe(z2, NN(ct,:));
end
```

Compute the small sample-size corrected AIC values for the models, and return the smallest value.

```
V = aic(models{:},'AICc');
[Vmin,I] = min(V);
```

Return the optimal model that has the smallest AICc value.

```
models{I}
```

```
ans =
Discrete-time OE model:  y(t) = [B(z)/F(z)]u(t) + e(t)
  B(z) = 1.067 z^-2

  F(z) = 1 - 1.824 z^-1 + 1.195 z^-2 - 0.2307 z^-3

Sample time: 0.1 seconds

Parameterization:
   Polynomial orders:   nb=1   nf=3   nk=2
   Number of free coefficients: 4
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using OE on time domain data "z2".
Fit to estimation data: 86.53%
```

```
FPE: 0.9809, MSE: 0.9615
```

Input Arguments

`model` — Identified model

`idtf` | `idgrey` | `idpoly` | `idproc` | `idss` | `idnlarx`, | `idnlhw` | `idnlgrey`

Identified model, specified as one of the following model objects:

- `idtf`
- `idgrey`
- `idpoly`
- `idproc`
- `idss`
- `idnlarx`, except nonlinear ARX model that includes a binary-tree or neural network nonlinearity estimator
- `idnlhw`
- `idnlgrey`

`measure` — Type of AIC

`'nAIC'` (default) | `'aic'` | `'AICc'` | `'BIC'`

Type of AIC, specified as one of the following values:

- `'nAIC'` — Normalized AIC
- `'aic'` — Raw AIC
- `'AICc'` — Small sample-size corrected AIC
- `'BIC'` — Bayesian Information Criteria

See “Akaike's Information Criterion (AIC)” on page 1-15 for more information.

Output Arguments

`value` — Value of quality metric

scalar | vector

Value of the quality measure, returned as a scalar or vector. For multiple models, `value` is a row vector where `value(k)` corresponds to the `k`th estimated model `modelk`.

More About

Akaike's Information Criterion (AIC)

Akaike's Information Criterion (AIC) provides a measure of model quality obtained 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. 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) includes the following quality metrics:

- Raw AIC, defined as:

  $$AIC = N * \log\left(\det\left(\frac{1}{N}\sum_{1}^{N}\varepsilon\left(t,\theta_N\right)\left(\varepsilon\left(t,\theta_N\right)\right)^T\right)\right) + 2n_p + N*\left(n_y*\left(\log(2\pi)+1\right)\right)$$

  where:

  - N is the number of values in the estimation data set
  - $\varepsilon(t)$ is a n_y-by-1 vector of prediction errors
  - θ_N represents the estimated parameters
  - n_p is the number of estimated parameters
  - n_y is the number of model outputs
- Small sample-size corrected AIC, defined as:

  $$AICc = AIC + 2n_p * \frac{n_p + 1}{N - n_p - 1}$$

- Normalized AIC, defined as:

  $$nAIC = \log\left(\det\left(\frac{1}{N}\sum_{1}^{N}\varepsilon\left(t,\theta_N\right)\left(\varepsilon\left(t,\theta_N\right)\right)^T\right)\right) + \frac{2n_p}{N}$$

- Bayesian Information Criteria, defined as:

$$BIC = N * \log\left(\det\left(\frac{1}{N}\sum_{1}^{N}\varepsilon\left(t,\theta_N\right)\left(\varepsilon\left(t,\theta_N\right)\right)^T\right)\right) + N * (n_y * \log(2\pi) + 1) + n_p * \log(N)$$

Tips

- The software computes and stores all types of Akaike's Information Criterion metrics during model estimation. If you want to access these values, see the `Report.Fit` property of the model.
- “Loss Function and Model Quality Metrics”

References

[1] 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` | `goodnessOfFit`

Introduced before R2006a

append

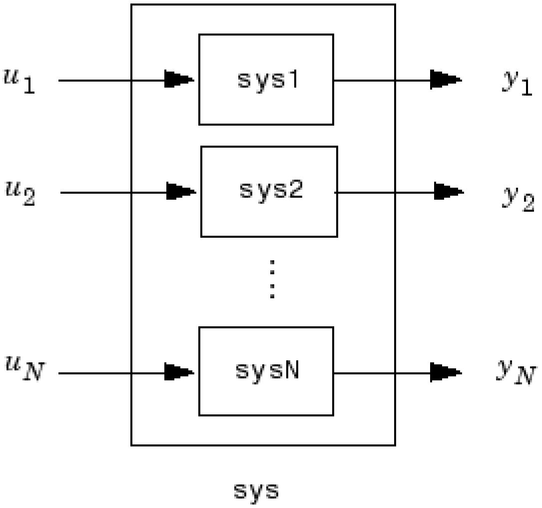
Group models by appending their inputs and outputs

Syntax

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

Description

`sys = append(sys1,sys2,...,sysN)` appends the inputs and outputs of the models `sys1`,...,`sysN` to form the augmented model `sys` depicted below.



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

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

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 "Rules That Determine Model Type" for details).

There is no limitation on the number of inputs.

Examples

Append Inputs and Outputs of Models

Create a SISO transfer function.

```
sys1 = tf(1,[1 0]);
size(sys1)
```

```
Transfer function with 1 outputs and 1 inputs.
```

Create a SISO continuous-time state-space model.

```
sys2 = ss(1,2,3,4);
size(sys2)
```

```
State-space model with 1 outputs, 1 inputs, and 1 states.
```

Append the inputs and outputs of `sys1`, a SISO static gain system, and `sys2`. The resulting model should be a 3-input, 3-output state-space model.

```
sys = append(sys1,10,sys2)
size(sys)
```

```
sys =

  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

  D =
       u1  u2  u3
   y1   0   0   0
   y2   0  10   0
   y3   0   0   4

Continuous-time state-space model.

State-space model with 3 outputs, 3 inputs, and 2 states.
```

See Also

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

Introduced in R2012a

ar

Estimate parameters of AR model for scalar time series

Syntax

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

`m= 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).

Default:

`n`

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

`approach`

Algorithm for computing the least squares AR model, specified as one of the following values:

- `'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`

Use of information about the data outside the measured time interval (past and future values), specified as one of the following 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. `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.

Default:

`'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.

`refl`

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 `refl` is the zeroth-order model, and the (2,1) element of `refl` 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);
```

More About

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.

Method	Approach and Windowing
Modified Covariance Method	(Default) Forward-backward approach and no windowing.

Method	Approach and Windowing
Correlation Method	Yule-Walker approach, which corresponds to least squares plus pre- and postwindowing.
Covariance Method	Least squares approach with no windowing. `arx` uses this routine.

References

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

See Also

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

Introduced before R2006a

armax

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

`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 a discrete-time linear model `init_sys` to configure the initial parameterization.

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

Input Arguments

`data` — Time-domain estimation data

`iddata` object

Time-domain estimation data, specified as an `iddata` object. You cannot use frequency-domain data for estimating ARMAX models.

`[na nb nc nk]` — Polynomial orders

1-by-4 vector of positive integers | 1-by-4 vector of matrices

Polynomial orders of an "ARMAX Model" on page 1-38, specified as a 1-by-4 vector, `[na nb nc nk]`.

For a model with *Ny* outputs and *Nu* inputs:

- `na` is the order of the polynomial *A*(*q*), specified as an *Ny*-by-*Ny* matrix of nonnegative integers.
- `nb` is the order of the polynomial *B*(*q*) + 1, specified as an *Ny*-by-*Nu* matrix of nonnegative integers.
- `nc` is the order of the polynomial *C*(*q*), specified as a column vector of nonnegative integers of length *Ny*.
- `nk` is the input-output delay expressed as fixed leading zeros of the *B* polynomial.

  Specify `nk` as an *Ny*-by-*Nu* matrix of nonnegative integers.

`init_sys` — System for configuring initial parametrization

discrete-time linear model

System for configuring initial parametrization of `sys`, specified as a discrete-time linear model. You obtain `init_sys` by either performing an estimation using measured data or by direct construction using commands such as `idpoly` and `idss`.

If `init_sys` is an ARMAX model, `armax` uses the parameter values of `init_sys` as the initial guess for estimation. To configure initial guesses and constraints for *A*(*q*), *B*(*q*), and *C*(*q*), use the `Structure` property of `init_sys`. For example:

- To specify an initial guess for the *A*(*q*) term of `init_sys`, set `init_sys.Structure.A.Value` as the initial guess.
- To specify constraints for 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 `init_sys` is not a polynomial model of ARMAX structure, the software first converts `init_sys` to an ARMAX model. `armax` uses the parameters of the resulting model as the initial guess for estimating `sys`.

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

`opt` — Estimation options

`armaxOptions` option set

Estimation options for ARMAX model identification, specified as an `armaxOptions` option set.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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

`0` (default) | scalar | vector of positive integers

Input delays, specified as the comma-separated pair consisting of `'InputDelay'` and one of the following:

- `Nu`-by-1 vector, where `Nu` is the number of inputs — Each entry is a numerical value representing the input delay for the corresponding input channel. Specify input delays as integer multiples of the sample time `Ts`. For example, `InputDelay = 3` means a delay of three sampling periods.
- Scalar value — The same delay is applied to all input channels.

`'IODelay'` — Transport delays

`0` (default) | scalar | matrix

Transport delays for each input/output pair, specified as the comma-separated pair consisting of `'IODelay'` and one of the following:

- `Ny`-by-`Nu` matrix, where `Ny` is the number of outputs and `Nu` is the number of inputs — Each entry is an integer value representing the transport delay for the corresponding input/output pair. Specify transport delays as integers denoting delay of a multiple of the sample time `Ts`.
- Scalar value — The same delay is applied to all input/output pairs.

`'IODelay'` is useful as a replacement for the `nk` order. You can factor out `max(nk-1,0)` lags as the `IODelay` value. For `nk`>1, `armax(na,nb,nk)` is equivalent to `armax(na,nb,1,'IODelay',nk-1)`.

`'IntegrateNoise'` — Addition of integrators in noise channel

`false(Ny,1)`, where Ny is the number of outputs (default) | logical vector

Addition of integrators in noise channel, specified as the comma-separated pair consisting of `'IntegrateNoise'` and a logical vector of length Ny, where Ny 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 ARIMA or ARIMAX models.

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
```

Output Arguments

`sys` — ARMAX model

`idpoly` object

ARMAX model that fits the given estimation data, returned as a discrete-time `idpoly` object. This model is created using the specified model orders, delays, and estimation options.

Information about the estimation results and options used is stored in the `Report` property of the model. `Report` has the following fields:

Report Field	Description
`Status`	Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Method</code></td><td>Estimation command used.</td></tr>
<tr><td><code>InitialCo</code></td><td>Handling of initial conditions during model estimation, returned as one of the following values:
<code>'zero'</code> — The initial conditions were set to zero.
<code>'estimate'</code> — The initial conditions were treated as independent estimation parameters.
<code>'backcast'</code> — The initial conditions were estimated using the best least squares fit.
This field is especially useful to view how the initial conditions were handled when the <code>InitialCondition</code> option in the estimation option set is <code>'auto'</code>.</td></tr>
<tr><td><code>Fit</code></td><td>Quantitative assessment of the estimation, returned as a structure. See “Loss Function and Model Quality Metrics” for more information on these quality metrics. The structure has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>FitPerce</code></td><td>Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as a percentage.</td></tr>
<tr><td><code>LossFcn</code></td><td>Value of the loss function when the estimation completes.</td></tr>
<tr><td><code>MSE</code></td><td>Mean squared error (MSE) measure of how well the response of the model fits the estimation data.</td></tr>
<tr><td><code>FPE</code></td><td>Final prediction error for the model.</td></tr>
<tr><td><code>AIC</code></td><td>Raw Akaike Information Criteria (AIC) measure of model quality.</td></tr>
<tr><td><code>AICc</code></td><td>Small sample-size corrected AIC.</td></tr>
<tr><td><code>nAIC</code></td><td>Normalized AIC.</td></tr>
<tr><td><code>BIC</code></td><td>Bayesian Information Criteria (BIC).</td></tr>
</table></td></tr>
<tr><td><code>Parameter</code></td><td>Estimated values of model parameters.</td></tr>
<tr><td><code>OptionsUs</code></td><td>Option set used for estimation. If no custom options were configured, this is a set of default options. See <code>armaxOptions</code> for more information.</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>RandState</code></td><td>State of the random number stream at the start of estimation. Empty, <code>[]</code>, if randomization was not used during estimation. For more information, see <code>rng</code> in the MATLAB documentation.</td></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation, returned as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
<tr><td><code>Type</code></td><td>Data type.</td></tr>
<tr><td><code>Length</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time.</td></tr>
<tr><td><code>InterSam</code></td><td>Input intersample behavior, returned as one of the following values:
• <code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
• <code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
• <code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.</td></tr>
<tr><td><code>InputOff</code></td><td>Offset removed from time-domain input data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
<tr><td><code>OutputOf</code></td><td>Offset removed from time-domain output data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
</table>
</td></tr>
</table>

Report Field	Description
`Terminati`	Termination conditions for the iterative search used for prediction error minimization. Structure with the following fields: <table><tr><th>Field</th><th>Description</th></tr><tr><td>WhyStop</td><td>Reason for terminating the numerical search.</td></tr><tr><td>Iteratio</td><td>Number of search iterations performed by the estimation algorithm.</td></tr><tr><td>FirstOrd</td><td>∞ -norm of the gradient search vector when the search algorithm terminates.</td></tr><tr><td>FcnCount</td><td>Number of times the objective function was called.</td></tr><tr><td>UpdateNo</td><td>Norm of the gradient search vector in the last iteration. Omitted when the search method is 'lsqnonlin'.</td></tr><tr><td>LastImpr</td><td>Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is 'lsqnonlin'.</td></tr><tr><td>Algorith</td><td>Algorithm used by 'lsqnonlin' search method. Omitted when other search methods are used.</td></tr></table> For estimation methods that do not require numerical search optimization, the `Termination` field is omitted.

For more information on using `Report`, see "Estimation Report".

Examples

Estimate ARMAX Model Using Regularization

Estimate a regularized ARMAX model by converting a regularized ARX model.

Load data.

```
load regularizationExampleData.mat m0simdata;
```

Estimate an unregularized ARMAX model of order 15.

```
m1 = armax(m0simdata(1:150),[30 30 30 1]);
```

Estimate a regularized ARMAX model by determining Lambda value by trial and error.

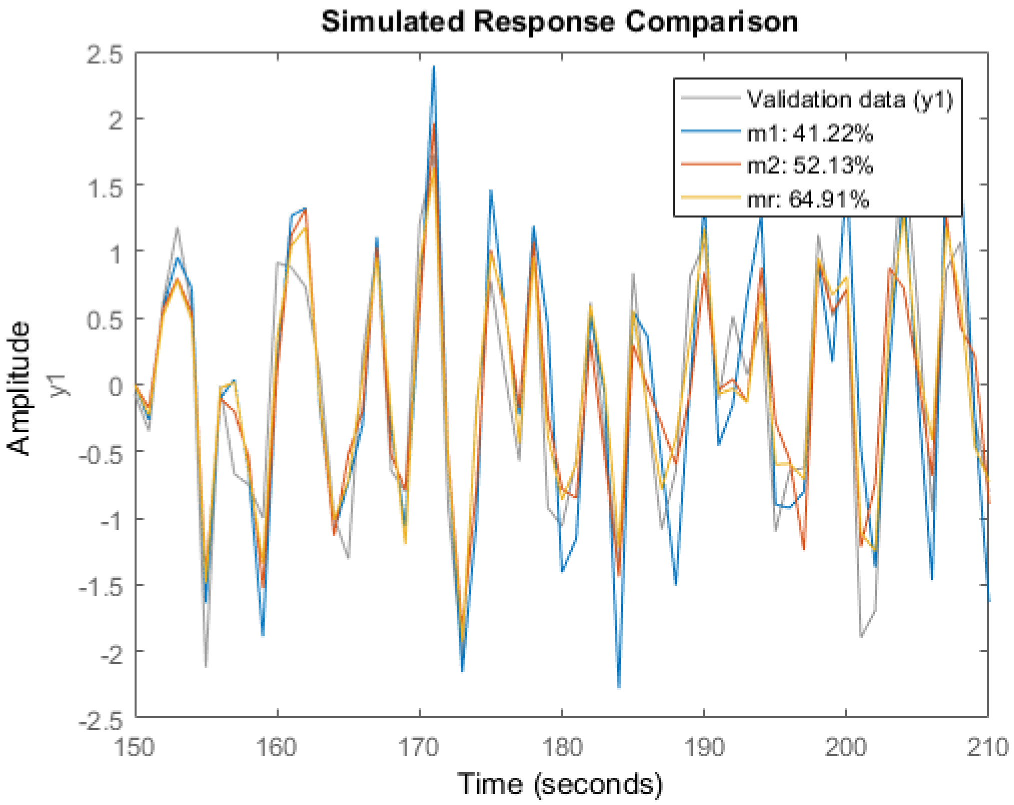
```
opt = armaxOptions;
opt.Regularization.Lambda = 1;
m2 = armax(m0simdata(1:150),[30 30 30 1],opt);
```

Obtain a lower-order ARMAX model by converting a regularized ARX model followed by order reduction.

```
opt1 = arxOptions;
[L,R] = arxRegul(m0simdata(1:150),[30 30 1]);
opt1.Regularization.Lambda = L;
opt1.Regularization.R = R;
m0 = arx(m0simdata(1:150),[30 30 1],opt1);
mr = idpoly(balred(idss(m0),7));
```

Compare the model outputs against data.

```
opt2 = compareOptions('InitialCondition','z');
compare(m0simdata(150:end),m1,m2,mr,opt2);
```



Specify 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.

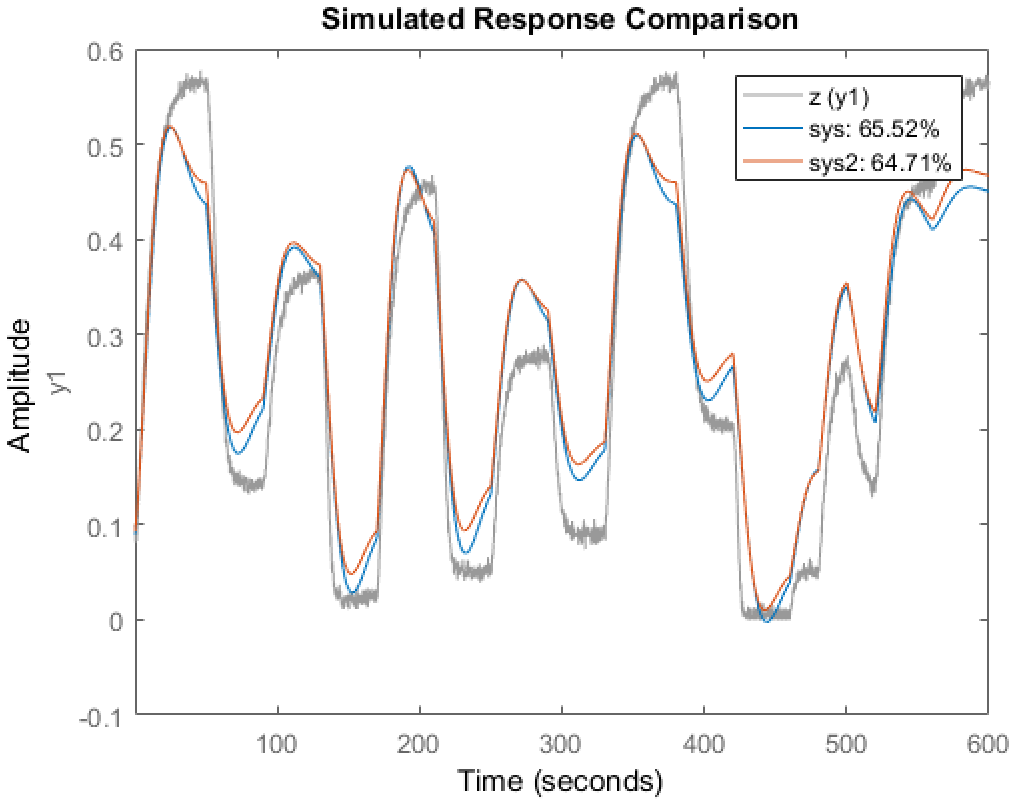
```
load twotankdata;
z = iddata(y,u,0.2);
opt = armaxOptions;
opt.Focus = 'simulation';
opt.SearchMethod = 'lm';
opt.SearchOption.MaxIter = 10;
opt.Display = 'on';
```

```
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 continue iterations on the previously estimated model as follows:

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

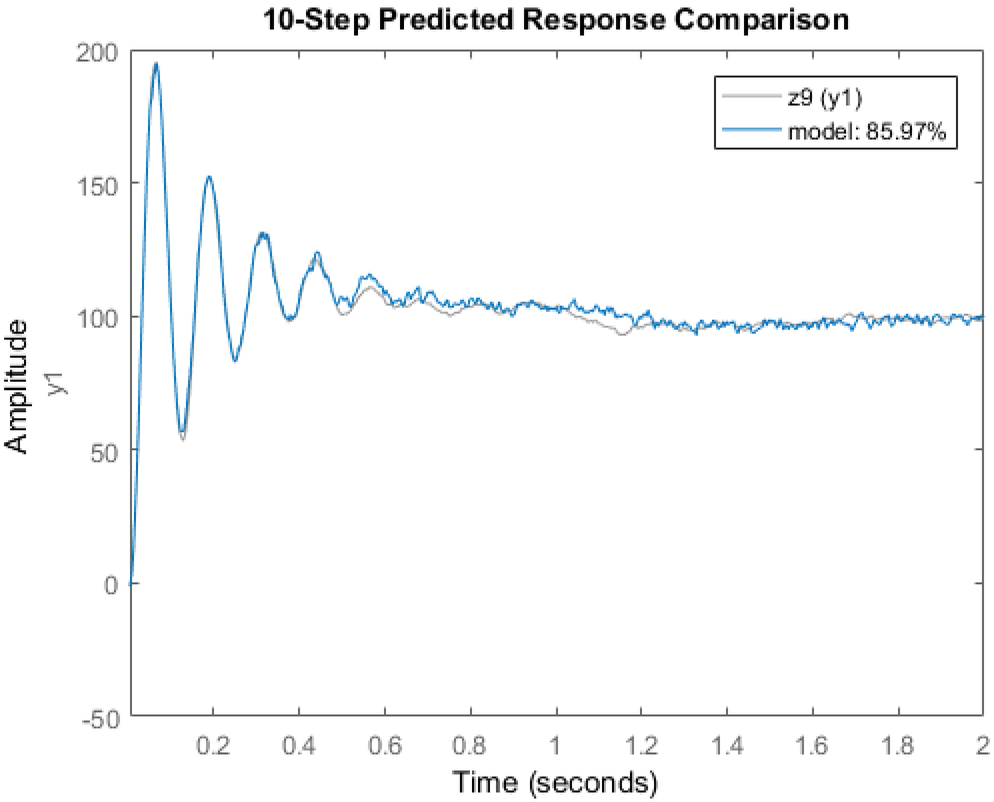


where `sys2` refines the parameters of `sys` to improve the fit to data.

Estimate ARIMA Model

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

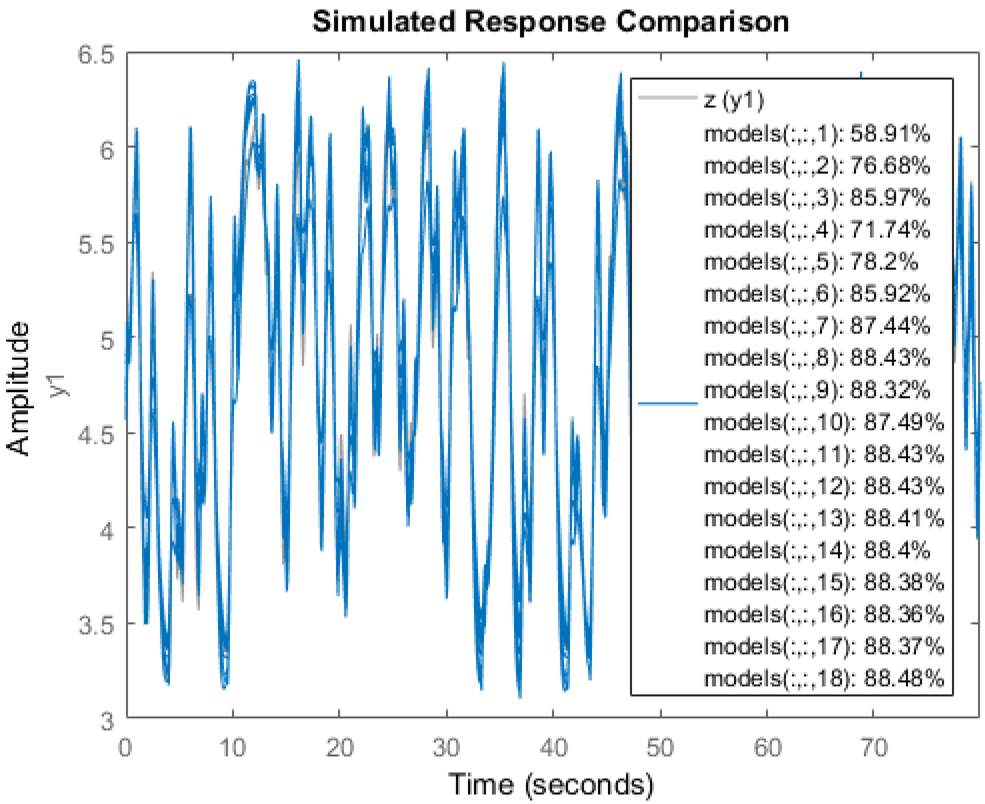
Estimate ARMAX models of varying orders iteratively from measured data.

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)
```



Estimate ARMAX Model Using State-Space Model as Inital Guess

Load the estimation data.

```
load iddata2 z2
```

Estimate a state-space model of order 3 from the estimation data.

```
sys0 = n4sid(z2,3);
```

Estimate an ARMAX model using the previously estimated state-space model as an initial guess.

```
sys = armax(z2,sys0);
```

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.

`armax` supports only time-domain data. For frequency-domain data, use `oe` to estimate an Output-Error (OE) model.

More About

ARMAX Model

The ARMAX model structure is:

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

A more compact way to write the difference equation is:

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

where,

- $y(t)$ — Output at time t.
- n_a — Number of poles.
- n_b — Number of zeroes plus 1.
- n_c — Number of C coefficients.
- n_k — Number of input samples that occur before the input affects the output, also called the *dead time* in the system.

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

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

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

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

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

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

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

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)*:

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

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

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

Tips

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

Algorithms

An iterative search algorithm minimizes a robustified quadratic prediction error criterion. The iterations are terminated when any of the following is true:

- Maximum number of iterations is reached.
- Expected improvement is less than the specified tolerance.
- 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.

- “Loss Function and Model Quality Metrics”
- “Regularized Estimates of Model Parameters”
- “Estimation Report”

References

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

See Also

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

Introduced before R2006a

armaxOptions

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. `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'` — Handling of initial conditions
`'auto'` (default) | `'zero'` | `'estimate'` | `'backcast'`

Handling of initial conditions during estimation, specified as 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.

`'Focus'` — Error to be minimized

`'prediction'` (default) | `'simulation'`

Error to be minimized in the loss function during estimation, specified as the comma-separated pair consisting of `'Focus'` and one of the following values:

- `'prediction'` — The one-step ahead prediction error between measured and predicted outputs is minimized during estimation. As a result, the estimation focuses on producing a good predictor model.
- `'simulation'` — The simulation error between measured and simulated outputs is minimized during estimation. As a result, the estimation focuses on making a good fit for simulation of model response with the current inputs.

The `Focus` option can be interpreted as a weighting filter in the loss function. For more information, see “Loss Function and Model Quality Metrics”.

`'WeightingFilter'` — Weighting prefilter

`[]` (default) | vector | matrix | cell array | linear system

Weighting prefilter applied to the loss function to be minimized during estimation. To understand the effect of `WeightingFilter` on the loss function, see “Loss Function and Model Quality Metrics”.

Specify `WeightingFilter` as one of the following values:

- `[]` — No weighting prefilter is used.
- Passbands — Specify a row vector or matrix containing frequency values that define desired passbands. You select a frequency band where the fit between estimated model and estimation data is optimized. For example, `[wl,wh]`, where `wl` and `wh` represent lower and upper limits of a passband. For a matrix with several rows defining frequency passbands, `[w1l,w1h;w2l,w2h;w3l,w3h;...]`, the estimation algorithm uses the union of the frequency ranges to define the estimation passband.

  Passbands are expressed in `rad/TimeUnit` for time-domain data and in `FrequencyUnit` for frequency-domain data, where `TimeUnit` and `FrequencyUnit` are the time and frequency units of the estimation data.
- SISO filter — Specify a single-input-single-output (SISO) linear filter in one of the following ways:
  - A SISO LTI model

  - `{A,B,C,D}` format, which specifies the state-space matrices of a filter with the same sample time as estimation data.
  - `{numerator,denominator}` format, which specifies the numerator and denominator of the filter as a transfer function with same sample time as estimation data.

    This option calculates the weighting function as a product of the filter and the input spectrum to estimate the transfer function.

`'EnforceStability'` — Control whether to enforce stability of model

`false` (default) | `true`

Control whether to enforce stability of estimated model, specified as the comma-separated pair consisting of `'EnforceStability'` and either `true` or `false`.

This option is not available for multi-output models with a non-diagonal A polynomial array.

Data Types: `logical`

`'EstCovar'` — Control whether to generate parameter covariance data

`true` (default) | `false`

Controls whether parameter covariance data is generated, specified as `true` or `false`.

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

`'Display'` — Specify whether to display the estimation progress

`'off'` (default) | `'on'`

Specify whether to display the estimation progress, specified as one of the following values:

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

`'InputOffset'` — Removal of offset from time-domain input data during estimation

`[ ]` (default) | vector of positive integers | matrix

Removal of offset from time-domain input data during estimation, specified as the comma-separated pair consisting of `'InputOffset'` and one of the following:

- A column vector of positive integers of length *Nu*, where *Nu* is the number of inputs.
- `[]` — Indicates no offset.
- *Nu*-by-*Ne* matrix — For multi-experiment data, specify `InputOffset` as an *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.

`'OutputOffset'` — Removal of offset from time-domain output data during estimation

[ ] (default) | vector | matrix

Removal of offset from time-domain output data during estimation, specified as the comma-separated pair consisting of `'OutputOffset'` and one of the following:

- A column vector of length *Ny*, where *Ny* is the number of outputs.
- `[]` — Indicates no offset.
- *Ny*-by-*Ne* matrix — For multi-experiment 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.

`'Regularization'` — Options for regularized estimation of model parameters

structure

Options for regularized estimation of model parameters. For more information on regularization, see “Regularized Estimates of Model Parameters”.

`Regularization` is a structure with the following fields:

- `Lambda` — Constant that determines the bias versus variance tradeoff.

  Specify a positive scalar to add the regularization term to the estimation cost.

  The default value of zero implies no regularization.

  Default: 0
- `R` — Weighting matrix.

  Specify a vector of nonnegative numbers or a square positive semi-definite matrix. The length must be equal to the number of free parameters of the model.

For black-box models, using the default value is recommended. For structured and grey-box models, you can also specify a vector of `np` positive numbers such that each entry denotes the confidence in the value of the associated parameter.

The default value of 1 implies a value of `eye(npfree)`, where `npfree` is the number of free parameters.

Default: 1

- `Nominal` — The nominal value towards which the free parameters are pulled during estimation.

  The default value of zero implies that the parameter values are pulled towards zero. If you are refining a model, you can set the value to `'model'` to pull the parameters towards the parameter values of the initial model. The initial parameter values must be finite for this setting to work.

  Default: 0

`'SearchMethod'` — Search method used for iterative parameter estimation

`'auto'` (default) | `'gn'` | `'gna'` | `'lm'` | `'lsqnonlin'` | `'grad'`

Search method used for iterative parameter estimation, specified as one of the following values:

- `'gn'` — The subspace Gauss-Newton direction. Singular values of the Jacobian matrix less than `GnPinvConst*eps*max(size(J))*norm(J)` are discarded when computing the search direction. J is the Jacobian matrix. The Hessian matrix is approximated by J^TJ. 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 `gamma*max(sv)` of the Hessian are ignored, 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*LMStep` each time a search is successful without any bisections.
- `'lm'` — Uses the Levenberg-Marquardt method so that the next parameter value is `-pinv(H+d*I)*grad` from the previous one. 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 handle only the Trace criterion.
- `'grad'` — The steepest descent gradient search method.
- `'auto'` — The algorithm chooses one of the preceding options. The descent direction is calculated using `'gn'`, `'gna'`, `'lm'`, and `'grad'` successively at each iteration. The iterations continue until a sufficient reduction in error is achieved.

`'SearchOption'` — Option set for the search algorithm

search option set

Option set for the search algorithm with fields that depend on the value of `SearchMethod`.

SearchOption structure when SearchMethod is specified as 'gn', 'gna', 'lm', 'grad', or 'auto'

Field Name	Description
`Toleran`	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 stop. The estimate of the expected loss-function improvement at the next iteration is based on the Gauss-Newton vector computed for the current parameter value. **Default:** `0.01`
`MaxIter`	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 start-up procedure. Use `sys.Report.Termination.Iterations` to get the actual number of iterations during an estimation, where *sys* is an `idtf` model. **Default:** `20`
`Advance`	Advanced search settings.

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>Specified as a structure with the following fields:
<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td>`GnPinvCon`</td><td>Singular values of the Jacobian matrix that are smaller than `GnPinvConst*max(size(J)*norm(J)*eps)` are discarded when computing the search direction. Applicable when `SearchMethod` is `'gn'`.
`GnPinvConst` must be a positive, real value.
Default: `10000`</td></tr>
<tr><td>`InitGnaTo`</td><td>Initial value of *gamma*. Applicable when `SearchMethod` is `'gna'`.
Default: `0.0001`</td></tr>
<tr><td>`LMStartVa`</td><td>Starting value of search-direction length *d* in the Levenberg-Marquardt method. Applicable when `SearchMethod` is `'lm'`.
Default: `0.001`</td></tr>
<tr><td>`LMStep`</td><td>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`</td></tr>
<tr><td>`MaxBisect`</td><td>Maximum number of bisections used by the line search along the search direction.
Default: `25`</td></tr>
<tr><td>`MaxFunEva`</td><td>Iterations stop if the number of calls to the model file exceeds this value.
`MaxFunEvals` must be a positive, integer value.
Default: `Inf`</td></tr>
<tr><td>`MinParCha`</td><td>Smallest parameter update allowed per iteration.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td><code>MinParChange</code> must be a positive, real value.

Default: <code>0</code></td></tr>
<tr><td><code>RelImprov</code></td><td>Iterations stop if the relative improvement of the criterion function is less than <code>RelImprovement</code>.

<code>RelImprovement</code> must be a positive, integer value.

Default: <code>0</code></td></tr>
<tr><td><code>StepReduc</code></td><td>Suggested parameter update is reduced by the factor <code>StepReduction</code> after each try. This reduction continues until either <code>MaxBisections</code> tries are completed or a lower value of the criterion function is obtained.

<code>StepReduction</code> must be a positive, real value that is greater than 1.

Default: <code>2</code></td></tr>
</table>
</td></tr>
</table>

SearchOption structure when SearchMethod is specified as 'lsqnonlin'

Field Name	Description
`TolFun`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values. The value of `TolFun` is the same as that of `opt.SearchOption.Advanced.TolFun`. **Default:** `1e-5`
`TolX`	Termination tolerance on the estimated parameter values. The value of `TolX` is the same as that of `opt.SearchOption.Advanced.TolX`. **Default:** `1e-6`

Field Name	Description
`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 `opt.SearchOption.Advanced.MaxIter`. **Default:** `20`
`Advance`	Options set for `lsqnonlin`. For more information, see the Optimization Options table in “Optimization Options”. Use `optimset('lsqnonlin')` to create an options set for `lsqnonlin`, and then modify it to specify its various options.

`'Advanced'` — Additional advanced options

structure

Additional advanced options, specified as 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 loss function. The standard deviation is estimated robustly as the median of the absolute deviations from the median of the prediction errors and divided by `0.7`. For more information on robust norm choices, see section 15.2 of [2].

  `ErrorThreshold = 0` disables robustification and leads to a purely quadratic loss function. When estimating with frequency-domain data, the software sets `ErrorThreshold` to zero. For time-domain data that contains outliers, try setting `ErrorThreshold` to `1.6`.

  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 condition.

  The initial condition is estimated when

  $$\frac{\left\| y_{p,z} - y_{meas} \right\|}{\left\| y_{p,e} - y_{meas} \right\|} > \text{AutoInitThreshold}$$

  - y_{meas} is the measured output.
  - $y_{p,z}$ is the predicted output of a model estimated using zero initial conditions.
  - $y_{p,e}$ is the predicted output of a model estimated using estimated initial conditions.

  Applicable when `InitialCondition` is `'auto'`.

  Default: `1.05`

Output Arguments

`opt` — Options set for `armax`
`armaxOptions` option set

Option set for `armax`, returned as an `armaxOptions` option set.

Examples

Create Default Options Set for ARMAX Estimation

```
opt = armaxOptions;
```

Specify Options for ARMAX Estimation

Create an option set for `armax` to use the 'simulation' Focus and to set the Display to 'on'.

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

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

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

More About

- “Loss Function and Model Quality Metrics”

References

[1] 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.

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

See Also

`armax` | `idfilt`

Introduced in R2012a

arOptions

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. `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` is specified as one of the following values:

- `'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) is handled.

`Window` is specified as one of the following values:

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

This option is ignored when you use the Yule-Walker approach.

Default: `'now'`

`'DataOffset'`

Data offset level that is removed before estimation of parameters.

Specify `DataOffset` as a double scalar. For multiexperiment data, specify `DataOffset` as a vector of length *Ne*, where *Ne* 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 uses 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

```
opt = arOptions;
```

Specify Options for AR Estimation

Create an options set for `ar` using the least squares algorithm for estimation. Set `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`

Introduced in R2012a

arx

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 Ny-by-Ny matrix of nonnegative integers. Ny is the number of outputs.
- `nb` — Order of the polynomial $B(q) + 1$.

  `nb` is an Ny-by-Nu matrix of nonnegative integers. Ny is the number of outputs and Nu is the number of inputs.
- `nk` — Input-output delay expressed as fixed leading zeros of the B polynomial.

  Specify `nk` as an Ny-by-Nu matrix of nonnegative integers. Ny is the number of outputs and Nu 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. `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 sample time `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.

Specify transport delays as integers denoting delay of a multiple of the sample time, `Ts`.

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. 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'`

Specify integrators in the noise channels.

Adding an integrator creates an ARIX model represented by:

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

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

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

Default: `false(Ny,1)`, where `Ny` is the number of outputs

Output Arguments

`sys`

ARX model that fits the estimation data, returned as a discrete-time `idpoly` object. This model is created using the specified model orders, delays, and estimation options.

Information about the estimation results and options used is stored in the `Report` property of the model. `Report` has the following fields:

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Status</code></td><td>Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.</td></tr>
<tr><td><code>Method</code></td><td>Estimation command used.</td></tr>
<tr><td><code>InitialC</code></td><td>Handling of initial conditions during model estimation, returned as one of the following values:
• <code>'zero'</code> — The initial conditions were set to zero.
• <code>'estimate'</code> — The initial conditions were treated as independent estimation parameters.
This field is especially useful to view how the initial conditions were handled when the <code>InitialCondition</code> option in the estimation option set is <code>'auto'</code>.</td></tr>
<tr><td><code>Fit</code></td><td>Quantitative assessment of the estimation, returned as a structure. See “Loss Function and Model Quality Metrics” for more information on these quality metrics. The structure has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>FitPerc</code></td><td>Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as a percentage.</td></tr>
<tr><td><code>LossFcn</code></td><td>Value of the loss function when the estimation completes.</td></tr>
<tr><td><code>MSE</code></td><td>Mean squared error (MSE) measure of how well the response of the model fits the estimation data.</td></tr>
<tr><td><code>FPE</code></td><td>Final prediction error for the model.</td></tr>
<tr><td><code>AIC</code></td><td>Raw Akaike Information Criteria (AIC) measure of model quality.</td></tr>
<tr><td><code>AICc</code></td><td>Small sample-size corrected AIC.</td></tr>
<tr><td><code>nAIC</code></td><td>Normalized AIC.</td></tr>
<tr><td><code>BIC</code></td><td>Bayesian Information Criteria (BIC).</td></tr>
</table></td></tr>
<tr><td><code>Parameter</code></td><td>Estimated values of model parameters.</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>OptionsUs</code></td><td>Option set used for estimation. If no custom options were configured, this is a set of default options. See <code>arxOptions</code> for more information.</td></tr>
<tr><td><code>RandState</code></td><td>State of the random number stream at the start of estimation. Empty, <code>[]</code>, if randomization was not used during estimation. For more information, see <code>rng</code> in the MATLAB documentation.</td></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation, returned as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
<tr><td><code>Type</code></td><td>Data type.</td></tr>
<tr><td><code>Length</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time.</td></tr>
<tr><td><code>InterSam</code></td><td>Input intersample behavior, returned as one of the following values:

<code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
<code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
<code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.
</td></tr>
<tr><td><code>InputOff</code></td><td>Offset removed from time-domain input data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
<tr><td><code>OutputOf</code></td><td>Offset removed from time-domain output data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
</table></td></tr>
</table>

For more information on using `Report`, see “Estimation Report”.

Examples

Estimate ARX model

Generate input data based on a specified ARX model, and then use 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]);
```

Estimate ARX Model Using Regularization

Use `arxRegul` to automatically determine regularization constants and use the values for estimating an FIR model of order 50.

Obtain `L` and `R` values.

```
load regularizationExampleData eData;
orders = [0 50 0];
[L,R] = arxRegul(eData,orders);
```

By default, the `TC` kernel is used.

Use the returned `Lambda` and `R` values for regularized ARX model estimation.

```
opt = arxOptions;
opt.Regularization.Lambda = L;
opt.Regularization.R = R;
model = arx(eData,orders,opt);
```

More About

ARX structure

The ARX model structure is :

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

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

- $y(t)$ — Output at time t.
- n_a — Number of poles.
- n_b — Number of zeroes plus 1.
- n_k — Number of input samples that occur before the input affects the output, also called the *dead time* in the system.
- $y(t-1)\ldots y(t-n_a)$ — Previous outputs on which the current output depends.
- $u(t-n_k)\ldots u(t-n_k-n_b+1)$ — Previous and delayed inputs on which the current output depends.
- $e(t)$ — White-noise disturbance value.

A more compact way to write the difference equation is

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

q is the delay operator. Specifically,

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

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

Time Series Models

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

The AR model structure is

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

Multiple Inputs and Single-Output Models

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

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

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 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.

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.

Without regularization, the ARX model parameters vector θ is estimated by solving the normal equation:

$$\left(J^T J\right)\theta = J^T y$$

where J is the regressor matrix and y is the measured output. Therefore,

$$\theta = \left(J^T J\right)^{-1} J^T y\,.$$

Using regularization adds a regularization term:

$$\theta = \left(J^T J + \lambda R\right)^{-1} J^T y$$

where, λ and R are the regularization constants. See `arxOptions` for more information on the regularization constants.

- “Using Linear Model for Nonlinear ARX Estimation”
- “Regularized Estimates of Model Parameters”

See Also

`arxOptions` | `arxRegul` | `arxstruc` | `ar` | `armax` | `bj` | `iv4` | `n4sid` | `oe` | `nlarx` | `impulseest`

Introduced before R2006a

arxOptions

Option set for `arx`

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. `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'` — Handling of initial conditions
`'auto'` (default) | `'zero'` | `'estimate'`

Handling of initial conditions during estimation using frequency-domain data, specified as the comma-separated pair consisting of `'InitialCondition'` and one of the following values:

- `'zero'` — The initial conditions are set to zero.
- `'estimate'` — The initial conditions are treated as independent estimation parameters.

- `'auto'` — The software chooses the method to handle initial conditions based on the estimation data.

`'Focus'` — Error to be minimized

`'prediction'` (default) | `'simulation'`

Error to be minimized in the loss function during estimation, specified as the comma-separated pair consisting of `'Focus'` and one of the following values:

- `'prediction'` — The one-step ahead prediction error between measured and predicted outputs is minimized during estimation. As a result, the estimation focuses on producing a good predictor model.
- `'simulation'` — The simulation error between measured and simulated outputs is minimized during estimation. As a result, the estimation focuses on making a good fit for simulation of model response with the current inputs.

The `Focus` option can be interpreted as a weighting filter in the loss function. For more information, see “Loss Function and Model Quality Metrics”.

`'WeightingFilter'` — Weighting prefilter

[ ] (default) | vector | matrix | cell array | linear system

Weighting prefilter applied to the loss function to be minimized during estimation. To understand the effect of `WeightingFilter` on the loss function, see “Loss Function and Model Quality Metrics”.

Specify `WeightingFilter` as one of the following values:

- `[]` — No weighting prefilter is used.
- Passbands — Specify a row vector or matrix containing frequency values that define desired passbands. You select a frequency band where the fit between estimated model and estimation data is optimized. For example, `[wl,wh]` where `wl` and `wh` represent lower and upper limits of a passband. For a matrix with several rows defining frequency passbands, `[w1l,w1h;w2l,w2h;w3l,w3h;...]`, the estimation algorithm uses the union of the frequency ranges to define the estimation passband.

  Passbands are expressed in `rad/TimeUnit` for time-domain data and in `FrequencyUnit` for frequency-domain data, where `TimeUnit` and `FrequencyUnit` are the time and frequency units of the estimation data.
- SISO filter — Specify a single-input-single-output (SISO) linear filter in one of the following ways:

  - A SISO LTI model
  - `{A,B,C,D}` format, which specifies the state-space matrices of a filter with the same sample time as estimation data.
  - `{numerator,denominator}` format, which specifies the numerator and denominator of the filter as a transfer function with same sample time as estimation data.

    This option calculates the weighting function as a product of the filter and the input spectrum to estimate the transfer function.
- Weighting vector — Applicable for frequency-domain data only. Specify a column vector of weights. This vector must have the same length as 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.

`'EnforceStability'` — Control whether to enforce stability of model

`false` (default) | `true`

Control whether to enforce stability of estimated model, specified as the comma-separated pair consisting of `'EnforceStability'` and either `true` or `false`.

This option is not available for multi-output models with a non-diagonal *A* polynomial array.

Data Types: `logical`

`'EstCovar'` — Control whether to generate parameter covariance data

`true` (default) | `false`

Controls whether parameter covariance data is generated, specified as `true` or `false`.

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

`'Display'` — Specify whether to display the estimation progress

`'off'` (default) | `'on'`

Specify whether to display the estimation progress, specified as one of the following values:

- `'on'` — Information on model structure and estimation results are displayed in a progress-viewer window.

- `'off'` — No progress or results information is displayed.

`'InputOffset'` — Removal of offset from time-domain input data during estimation

[ ] (default) | vector of positive integers | matrix

Removal of offset from time-domain input data during estimation, specified as the comma-separated pair consisting of `'InputOffset'` and one of the following:

- A column vector of positive integers of length *Nu*, where *Nu* is the number of inputs.
- `[]` — Indicates no offset.
- *Nu*-by-*Ne* matrix — For multi-experiment data, specify `InputOffset` as an *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.

`'OutputOffset'` — Removal of offset from time-domain output data during estimation

[ ] (default) | vector | matrix

Removal of offset from time-domain output data during estimation, specified as the comma-separated pair consisting of `'OutputOffset'` and one of the following:

- A column vector of length *Ny*, where *Ny* is the number of outputs.
- `[]` — Indicates no offset.
- *Ny*-by-*Ne* matrix — For multi-experiment 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.

`'OutputWeight'` — Weight of prediction errors in multi-output estimation

[ ] (default) | positive semidefinite, symmetric matrix

Weight of prediction errors in multi-output estimation, specified as one of the following values:

- Positive semidefinite, symmetric matrix (`W`). The software minimizes the trace of the weighted prediction error matrix `trace(E'*E*W/N)` where:
  - `E` is the matrix of prediction errors, with one column for each output, and `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-output models, or the reliability of corresponding data.
  - `N` is the number of data samples.
- `[]` — No weighting is used. Specifying as `[]` is the same as `eye(Ny)`, where `Ny` is the number of outputs.

This option is relevant only for multi-output models.

`'Regularization'` — Options for regularized estimation of model parameters

[ ] (default) | positive semidefinite, symmetric matrix

Options for regularized estimation of model parameters, specified as a structure with the following fields:

- `Lambda` — Constant that determines the bias versus variance tradeoff.

  Specify a positive scalar to add the regularization term to the estimation cost.

  The default value of zero implies no regularization.

  Default: 0
- `R` — Weighting matrix.

  Specify a positive scalar or a positive definite matrix. The length of the matrix must be equal to the number of free parameters (`np`) of the model. For ARX model, `np` = sum(sum([`na` `nb`]).

  Default: 1
- `Nominal` — This option is not used for ARX models.

  Default: 0

Use `arxRegul` to automatically determine Lambda and R values.

For more information on regularization, see “Regularized Estimates of Model Parameters”.

`'Advanced'` — Additional advanced options

structure

Additional advanced options, specified as 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(eps)`

Output Arguments

`opt` — Options set for `arx`

`arxOptions` option set

Option set for `arx`, returned as an `arxOptions` option set.

Examples

Create Default Options Set for ARX Estimation

```
opt = arxOptions;
```

Specify Options for ARX Estimation

Create an options set for `arx` using zero initial conditions for estimation. 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';
```

More About

- "Loss Function and Model Quality Metrics"

See Also

`arx` | `arxRegul` | `idfilt`

Introduced in R2012a

arxRegul

Determine regularization constants for ARX model estimation

Syntax

```
[lambda,R] = arxRegul(data,orders)
[lambda,R] = arxRegul(data,orders,options)
[lambda,R] = arxRegul(data,orders,Name,Value)
[lambda,R] = arxRegul(data,orders,options,Name,Value)
```

Description

`[lambda,R] = arxRegul(data,orders)` returns the regularization constants used for ARX model estimation. Use the regularization constants in `arxOptions` to configure the regularization options for ARX model estimation.

`[lambda,R] = arxRegul(data,orders,options)` specifies regularization options such as regularization kernel and I/O offsets.

`[lambda,R] = arxRegul(data,orders,Name,Value)` specifies model structure attributes, such as noise integrator and input delay, using one or more `Name,Value` pair arguments.

`[lambda,R] = arxRegul(data,orders,options,Name,Value)` specifies both regularization options and model structure attributes.

Examples

Determine Regularization Constants for ARX Model Estimation Using Default Kernel

```
load iddata1 z1;
orders = [10 10 1];
[Lambda,R] = arxRegul(z1,orders);
```

The ARX model is estimated using the default regularization kernel `TC`.

Use the `Lambda` and `R` values for ARX model estimation.

```
opt = arxOptions;
opt.Regularization.Lambda = Lambda;
opt.Regularization.R = R;
model = arx(z1,orders,opt);
```

Specify a Regularization Kernel

Specify `'DC'` as the regularization kernel and obtain a regularized ARX model of order [| 10 10 1|].

```
load iddata1 z1;
orders = [10 10 1];
option = arxRegulOptions('RegulKernel','DC');
[Lambda,R] = arxRegul(z1,orders,option);
```

Use the `Lambda` and `R` values for ARX model estimation.

```
arxOpt = arxOptions;
arxOpt.Regularization.Lambda = Lambda;
arxOpt.Regularization.R = R;
model = arx(z1,orders,arxOpt);
```

Specify Noise Source Integrator

Specify to include a noise source integrator in the noise component of the model.

```
load iddata1 z1;
orders = [10 10 1];
[Lambda,R] = arxRegul(z1,orders,'IntegrateNoise',true);
```

Specify Regularization Kernel And Noise Integrator

Specify the regularization kernel and include a noise source integrator in the noise component of the model.

```
load iddata1 z1;
orders = [10 10 1];
opt = arxRegulOptions('RegulKernel','DC');
[Lambda,R] = arxRegul(z1,orders,opt,'IntegrateNoise',true);
```

- “Estimate Regularized ARX Model Using System Identification App”

Input Arguments

`data` — Estimation data

`iddata` object

Estimation data, specified as an `iddata` object.

`orders` — ARX model orders

matrix of nonnegative integers

ARX model orders `[na nb nc]`, specified as a matrix of nonnegative integers. See the `arx` reference page for more information on model orders.

`options` — Regularization options

`arxRegulOptions` options set

Regularization options, specified as an options set you create using `arxRegulOptions`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example: `[Lambda, R] = arxRegul(z1,orders,option,'InputDelay',10);`

`'InputDelay'` — Input delay

`0` (default) | `positive integer`

Input delay, specified as a positive, nonzero numeric value representing the number of samples.

Example: `[Lambda, R] = arxRegul(z1,orders,'InputDelay',10);`

Data Types: `double`

`'IntegrateNoise'` — Noise source integrator

`false` (default) | `true`

Noise source integrator, specified as a logical. Specifies whether the noise source `e(t)` should contain an integrator. The default is `false`, indicating the noise integrator is off. To turn it on, change the value to `true`.

Example: `[Lambda, R] = arxRegul(z1,orders,'IntegrateNoise',true);`

Data Types: `logical`

Output Arguments

`lambda` — Constant that determines bias versus variance trade-off

positive scalar

Constant that determines the bias versus variance trade-off, returned as a positive scalar.

`R` — Weighting matrix

vector of nonnegative numbers | square positive semi-definite matrix

Weighting matrix, returned as a vector of nonnegative numbers or a positive definite matrix.

More About

Algorithms

Without regularization, the ARX model parameters vector θ is estimated by solving the normal equation:

$$\left(J^T J\right)\theta = J^T y$$

where J is the regressor matrix and y is the measured output. Therefore,

$$\theta = \left(J^T J\right)^{-1} J^T y\,.$$

Using regularization adds a regularization term:

$$\theta = \left(J^T J + \lambda R\right)^{-1} J^T y$$

where, λ and R are the regularization constants. See `arxOptions` for more information on the regularization constants.

- “Regularized Estimates of Model Parameters”

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

`arx` | `arxOptions` | `arxRegulOptions`

Introduced in R2013b

arxRegulOptions

Option set for `arxRegul`

Syntax

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

Description

`opt = arxRegulOptions` creates a default option set for `arxRegul`.

`opt = arxRegulOptions(Name,Value)` creates an options set with the options specified by one or more name-value pair arguments.

Examples

Create Default Options Set for Determining Regularization Constants

```
opt = arxRegulOptions;
```

Specify Regularizing Kernel for ARX Model Estimation

```
opt = arxRegulOptions('RegulKernel','DC');
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example: `option = arxRegulOptions('RegulKernel', 'DC')` specifies `'DC'` as the regularization kernel.

`'RegulKernel'` — Regularization kernel

`'TC'` (default) | `'SE'` | `'SS'` | `'HF'` | `'DI'` | `'DC'`

Regularization kernel, specified as one of the following values:

- `'TC'` — Tuned and correlated kernel
- `'SE'` — Squared exponential kernel
- `'SS'` — Stable spline kernel
- `'HF'` — High frequency stable spline kernel
- `'DI'` — Diagonal kernel
- `'DC'` — Diagonal and correlated kernel

The specified kernel is used for regularized estimation 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 about these choices, see [1].

Data Types: `char`

`'InputOffset'` — Offset levels present in the input signals of estimation data

`[]` (default) | vector | matrix

Offset levels present in the input signals of time-domain estimation data, specified as one of the following:

- An `Nu`-element column vector, where `Nu` is the number of inputs. For multi-experiment data, specify a `Nu`-by-`Ne` matrix, where `Ne` is the number of experiments. The offset value `InputOffset(i,j)` is subtracted from the i^{th} input signal of the j^{th} experiment.
- `[]` — No offsets.

Data Types: `double`

`'OutputOffset'` — Output signal offset levels

`[]` (default) | vector | matrix

Output signal offset level of time-domain estimation data, specified as one of the following:

- An `Ny`-element column vector, where `Ny` is the number of outputs. For multi-experiment data, specify a `Ny`-by-`Ne` matrix, where `Ne` is the number of experiments. The offset value `OputOffset(i,j)` is subtracted from the i^{th} output signal of the j^{th} experiment.
- `[]` — No offsets.

The specified values are subtracted from the output signals before using them for estimation.

Data Types: `double`

`'Advanced'` — Advanced estimation options

structure

Advanced options for regularized estimation, specified as a structure with the following fields:

- `MaxSize` — Maximum allowable size of Jacobian matrices formed during estimation, specified as a large positive number.

  Default: `250e3`
- `SearchMethod` — Search method for estimating regularization parameters, specified as one of the following values:
  - `'gn'`: Quasi-Newton line search.
  - `'fmincon'`: Trust-region-reflective constrained minimizer. In general, `'fmincon'` is better than `'gn'` for handling bounds on regularization parameters that are imposed automatically during estimation. Requires Optimization Toolbox software.

  Default: `'gn'`

  If you have the Optimization Toolbox software, the default is `'fmincon'`.

Output Arguments

`opt` — Regularization options

`arxRegulOptions` options set

Regularization options, returned as an `arxRegulOptions` options set.

More About

- “Regularized Estimates of Model Parameters”

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

`arxRegul`

Introduced in R2014a

arxstruc

Compute loss functions for single-output ARX models

Syntax

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

Arguments

`ze`

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

`zv`

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

`NN`

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

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

`maxsize`

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.

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`.

Examples

Generate Model-Order Combinations and Estimate Single-Input ARX Model

Create an ARX model for generating data.

```
A = [1 -1.5 0.7];
B = [0 1 0.5];
m0 = idpoly(A,B);
```

Generate random input and additive noise signals.

```
u = iddata([],idinput(400,'rbs'));
e = iddata([],0.1*randn(400,1));
```

Simulate the model output using the defined input and error signals.

```
y = sim(m0,[u e]);
z = [y,u];
```

Generate model-order combinations for estimation. Specify a delay of `1` for all models, and a model order range between `1` and `5` for `na` and `nb`.

```
NN = struc(1:5,1:5,1);
```

Estimate ARX models and compute the loss function for each model order combination. The input data is split into estimation and validation data sets.

```
V = arxstruc(z(1:200),z(201:400),NN);
```

Select the model order with the best fit to the validation data.

```
order = selstruc(V,0);
```

Estimate an ARX model of selected order.

```
M = arx(z,order);
```

Generate Model-Order Combinations and Estimate Multi-Input ARX Model

Create estimation and validation data sets.

```
load co2data;
Ts = 0.5; % Sample time 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.

```
NN = struc(2:4,2:5,[1 4],1:4,0);
```

Estimate an ARX model for each model order combination.

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

Select the model order with the best fit to the validation data.

```
order = selstruc(V,0);
```

Estimate an ARX model of selected order.

```
M = arx(ze,order);
```

More About

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.

See Also

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

Introduced before R2006a

balred

Model order reduction

Syntax

```
rsys = balred(sys,ORDERS)
rsys = balred(sys,ORDERS,BALDATA)
rsys = balred(___,opts)
```

Description

`rsys = balred(sys,ORDERS)` computes a reduced-order approximation `rsys` of the LTI model `sys`. The desired order (number of states) for `rsys` is specified by `ORDERS`. You can try multiple orders at once by setting `ORDERS` to a vector of integers, in which case `rsys` is a vector of reduced-order models. `balred` uses implicit balancing techniques to compute the reduced- order approximation `rsys`. Use `hsvd` to plot the Hankel singular values and pick an adequate approximation order. States with relatively small Hankel singular values can be safely discarded.

When `sys` has unstable poles, it is first decomposed into its stable and unstable parts using `stabsep`, and only the stable part is approximated. Use `balredOptions` to specify additional options for the stable/unstable decomposition.

This command requires Control System Toolbox™ license. `sys` can only be an identified state-space model (`idss`). The reduced-order model is also an `idss` model.

`rsys = balred(sys,ORDERS,BALDATA)` uses balancing data returned by `hsvd`. Because `hsvd` does most of the work needed to compute `rsys`, this syntax is more efficient when using `hsvd` and `balred` jointly.

`rsys = balred(___,opts)` computes the model reduction using options that you specify using `balredOptions`. Options include offset and tolerance options for computing the stable-unstable decompositions. There also options for emphasizing particular time or frequency intervals. See `balredOptions` for details.

Note: The order of the approximate model is always at least the number of unstable poles and at most the minimal order of the original model (number `NNZ` of nonzero Hankel singular values using an eps-level relative threshold)

Examples

Reduced-Order Approximation with Offset Option

Compute a reduced-order approximation of the system given by:

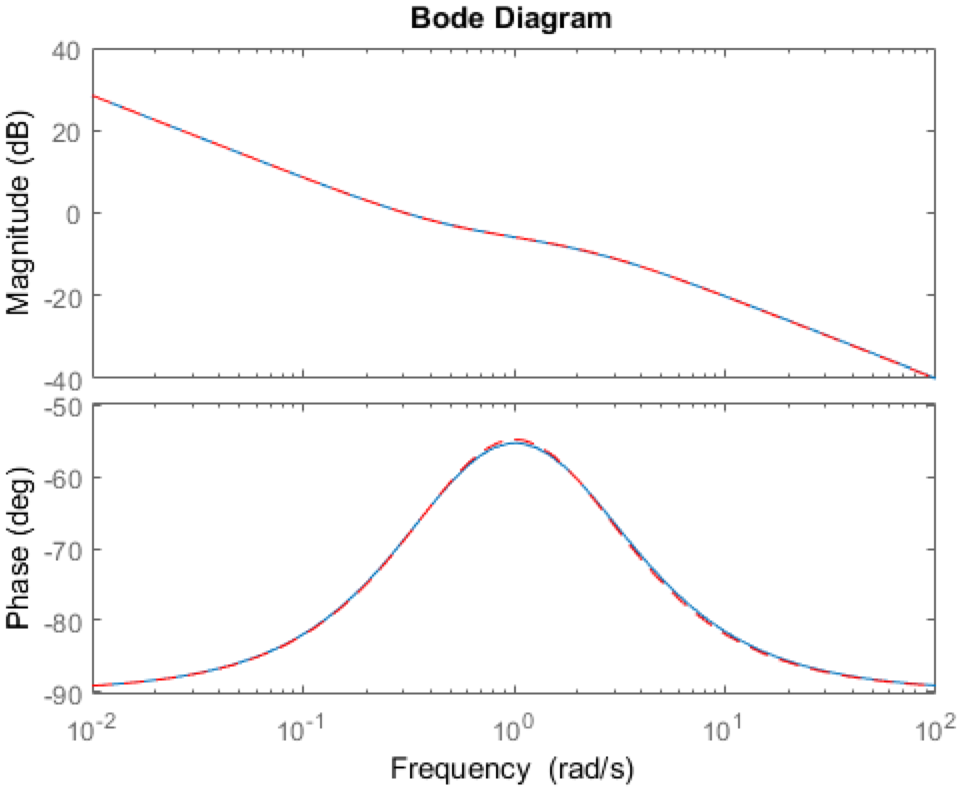
$$G(s) = \frac{(s+0.5)(s+1.1)(s+2.9)}{\left(s+10^{-6}\right)(s+1)(s+2)(s+3)}.$$

Use the `Offset` option to exclude the pole at $s = 10^{-6}$ from the stable term of the stable/unstable decomposition.

```
sys = zpk([-.5 -1.1 -2.9],[-1e-6 -2 -1 -3],1);
% Create balredOptions
opt = balredOptions('Offset',.001,'StateElimMethod','Truncate');
% Compute second-order approximation
rsys = balred(sys,2,opt);
```

Compare the responses of the original and reduced-order models.

```
bodeplot(sys,rsys,'r--')
```

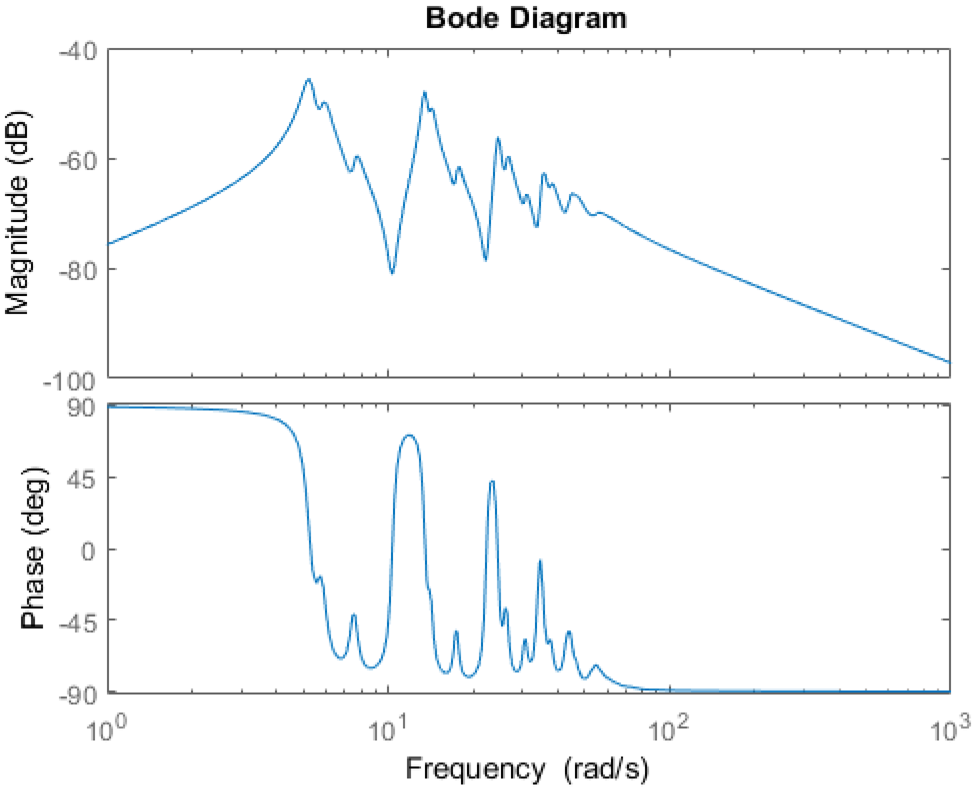


Model Reduction in a Particular Frequency Band

Reduce a high-order model with a focus on the dynamics in a particular frequency range.

Load a model and examine its frequency response.

```
load(fullfile(matlabroot,'examples','control','build.mat'),'G')
bodeplot(G)
```

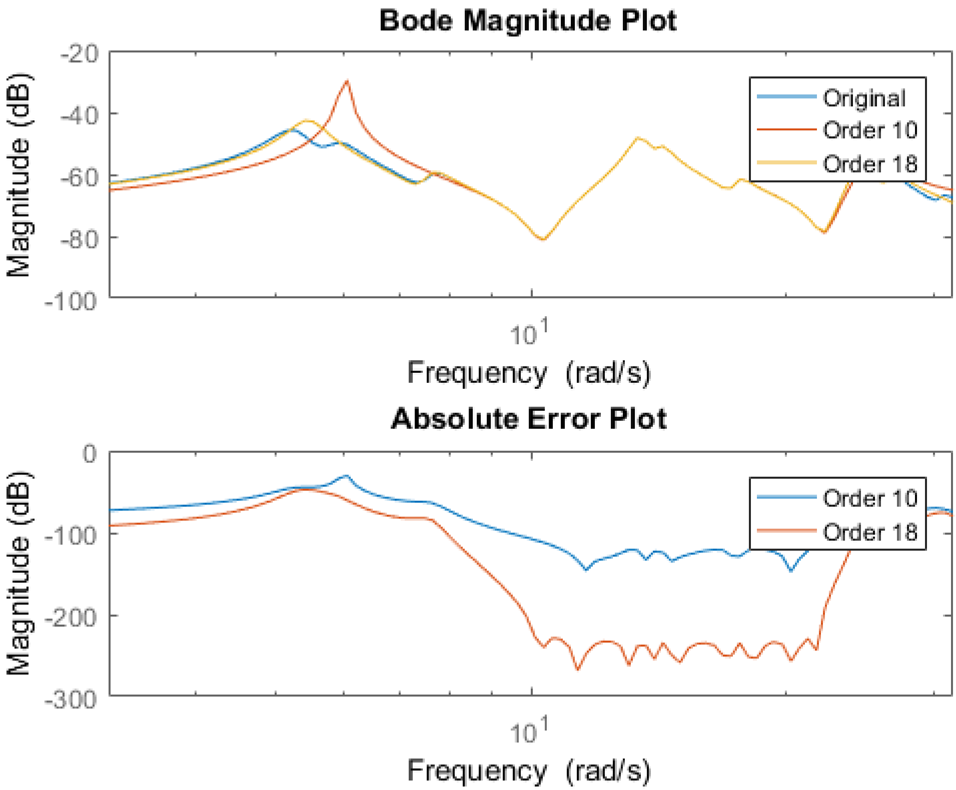


`G` is a 48th-order model with several large peak regions around 5.2 rad/s, 13.5 rad/s, and 24.5 rad/s, and smaller peaks scattered across many frequencies. Suppose that for your application you are only interested in the dynamics near the second large peak, between 10 rad/s and 22 rad/s. Focus the model reduction on the region of interest to obtain a good match with a low-order approximation. Use balredOptions to specify the frequency interval for `balred`.

```
bopt = balredOptions('FreqIntervals',[10,22]);
GLim10 = balred(G,10,bopt);
GLim18 = balred(G,18,bopt);
```

Examine the frequency responses of the reduced-order models. Also, examine the difference between those responses and the original response (the absolute error).

```
subplot(2,1,1);
bodemag(G,GLim10,GLim18,logspace(0.5,1.5,100));
title('Bode Magnitude Plot')
legend('Original','Order 10','Order 18');
subplot(2,1,2);
bodemag(G-GLim10,G-GLim18,logspace(0.5,1.5,100));
title('Absolute Error Plot')
legend('Order 10','Order 18');
```



With the frequency-limited energy computation, even the 10th-order approximation is quite good in the region of interest.

- “Balanced Truncation Model Reduction”
- “Model Reduction Basics”

References

[1] Varga, A., "Balancing-Free Square-Root Algorithm for Computing Singular Perturbation Approximations," Proc. of 30th IEEE CDC, Brighton, UK (1991), pp. 1062-1065.

See Also

`balredOptions` | `hsvd` | Model Reducer

Introduced before R2006a

bandwidth

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.

This command requires a Control System Toolbox license.

`fb = bandwidth(sys,dbdrop)` 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-Sp array of models, `bandwidth` returns an array of the same size such that

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

See Also

`dcgain` | `issiso`

Introduced before R2006a

bj

Estimate Box-Jenkins polynomial model using time domain data

Syntax

```
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)
```

Description

`sys = bj(data, [nb nc nd nf nk])` estimates a Box-Jenkins polynomial model, `sys`, using the time-domain data, `data`. `[nb nc nd nf nk]` define the orders of the polynomials used for estimation.

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

`sys = bj(data, init_sys)` estimates a Box-Jenkins polynomial using the polynomial model `init_sys` to configure the initial parameterization of `sys`.

`sys = bj(data, ___, opt)` estimates a Box-Jenkins polynomial using the option set, `opt`, to specify estimation behavior.

Input Arguments

`data`

Estimation data.

`data` is an `iddata` object that contains time-domain input and output signal values.

You cannot use frequency-domain data for estimating Box-Jenkins models.

Default:

`[nb nc nd nf nk]`

A vector of matrices containing the orders and delays of the Box-Jenkins model. Matrices must contain nonnegative integers.

- `nb` is the order of the B polynomial plus 1 (Ny-by-Nu matrix)
- `nc` is the order of the C polynomial plus 1 (Ny-by–1 matrix)
- `nd` is the order of the D polynomial plus 1 (Ny-by-1 matrix)
- `nf` is the order of the F polynomial plus 1 (Ny-by-Nu matrix)
- `nk` is the input delay (in number of samples, Ny-by-Nu matrix) where Nu is the number of inputs and Ny 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. `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 sample time `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.

Specify transport delays as integers denoting delay of a multiple of the sample time `Ts`.

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

`'IntegrateNoise'`

Logical specifying integrators in the noise channel.

`IntegrateNoise` is a logical vector of length *Ny*, where *Ny* 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)`(*Ny* is the number of outputs)

Output Arguments

`sys`

BJ model that fits the estimation data, returned as a discrete-time `idpoly` object. This model is created using the specified model orders, delays, and estimation options.

Information about the estimation results and options used is stored in the `Report` property of the model. `Report` has the following fields:

Report Field	Description
`Status`	Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.
`Method`	Estimation command used.
`InitialCo`	Handling of initial conditions during model estimation, returned as one of the following values: • `'zero'` — The initial conditions were set to zero. • `'estimate'` — The initial conditions were treated as independent estimation parameters. • `'backcast'` — The initial conditions were estimated using the best least squares fit. This field is especially useful to view how the initial conditions were handled when the `InitialCondition` option in the estimation option set is `'auto'`.

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Fit</code></td><td>Quantitative assessment of the estimation, returned as a structure. See “Loss Function and Model Quality Metrics” for more information on these quality metrics. The structure has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>FitPerc</code></td><td>Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as a percentage.</td></tr>
<tr><td><code>LossFcn</code></td><td>Value of the loss function when the estimation completes.</td></tr>
<tr><td><code>MSE</code></td><td>Mean squared error (MSE) measure of how well the response of the model fits the estimation data.</td></tr>
<tr><td><code>FPE</code></td><td>Final prediction error for the model.</td></tr>
<tr><td><code>AIC</code></td><td>Raw Akaike Information Criteria (AIC) measure of model quality.</td></tr>
<tr><td><code>AICc</code></td><td>Small sample-size corrected AIC.</td></tr>
<tr><td><code>nAIC</code></td><td>Normalized AIC.</td></tr>
<tr><td><code>BIC</code></td><td>Bayesian Information Criteria (BIC).</td></tr>
</table></td></tr>
<tr><td><code>Parameter</code></td><td>Estimated values of model parameters.</td></tr>
<tr><td><code>OptionsUs</code></td><td>Option set used for estimation. If no custom options were configured, this is a set of default options. See <code>bjOptions</code> for more information.</td></tr>
<tr><td><code>RandState</code></td><td>State of the random number stream at the start of estimation. Empty, <code>[]</code>, if randomization was not used during estimation. For more information, see <code>rng</code> in the MATLAB documentation.</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation, returned as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
<tr><td><code>Type</code></td><td>Data type.</td></tr>
<tr><td><code>Length</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time.</td></tr>
<tr><td><code>InterSam</code></td><td>Input intersample behavior, returned as one of the following values:
• <code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
• <code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
• <code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.</td></tr>
<tr><td><code>InputOff</code></td><td>Offset removed from time-domain input data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
<tr><td><code>OutputOf</code></td><td>Offset removed from time-domain output data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Terminati</code></td><td>Termination conditions for the iterative search used for prediction error minimization. Structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>WhyStop</code></td><td>Reason for terminating the numerical search.</td></tr>
<tr><td><code>Iteratio</code></td><td>Number of search iterations performed by the estimation algorithm.</td></tr>
<tr><td><code>FirstOrd</code></td><td>∞ -norm of the gradient search vector when the search algorithm terminates.</td></tr>
<tr><td><code>FcnCount</code></td><td>Number of times the objective function was called.</td></tr>
<tr><td><code>UpdateNo</code></td><td>Norm of the gradient search vector in the last iteration. Omitted when the search method is <code>'lsqnonlin'</code>.</td></tr>
<tr><td><code>LastImpr</code></td><td>Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is <code>'lsqnonlin'</code>.</td></tr>
<tr><td><code>Algorith</code></td><td>Algorithm used by <code>'lsqnonlin'</code> search method. Omitted when other search methods are used.</td></tr>
</table>
For estimation methods that do not require numerical search optimization, the <code>Termination</code> field is omitted.</td></tr>
</table>

For more information on using `Report`, see “Estimation Report”.

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.

Estimate Box-Jenkins Model Using Regularization

Estimate a regularized BJ model by converting a regularized ARX model.

Load data.

```
load regularizationExampleData.mat m0simdata;
```

Estimate an unregularized BJ model of order 30.

```
m1 = bj(m0simdata(1:150),[15 15 15 15 1]);
```

Estimate a regularized BJ model by determining Lambda value by trial and error.

```
opt = bjOptions;
opt.Regularization.Lambda = 1;
m2 = bj(m0simdata(1:150),[15 15 15 15 1],opt);
```

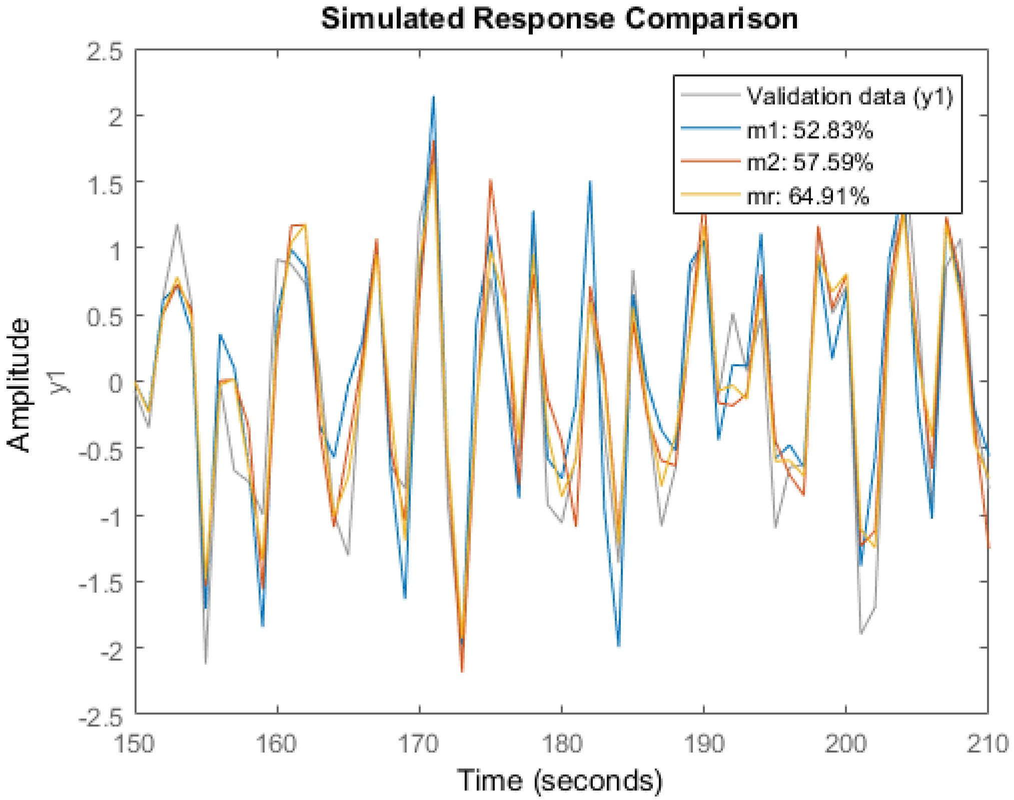
Obtain a lower-order BJ model by converting a regularized ARX model followed by order reduction.

```
opt1 = arxOptions;
```

```
[L,R] = arxRegul(m0simdata(1:150),[30 30 1]);
opt1.Regularization.Lambda = L;
opt1.Regularization.R = R;
m0 = arx(m0simdata(1:150),[30 30 1],opt1);
mr = idpoly(balred(idss(m0),7));
```

Compare the model outputs against data.

```
opt2 = compareOptions('InitialCondition','z');
compare(m0simdata(150:end),m1,m2,mr,opt2);
```



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`.

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`.

More About

Box-Jenkins Model Structure

The general Box-Jenkins model structure is:

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

where *nu* is the number of input channels.

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

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

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

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

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

- “Regularized Estimates of Model Parameters”

References

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

See Also

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

Introduced before R2006a

bjOptions

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. `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'` — Handling of initial conditions
`'auto'` (default) | `'zero'` | `'estimate'` | `'backcast'`

Handling of initial conditions during estimation, specified as 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.

`'Focus'` — Error to be minimized

`'prediction'` (default) | `'simulation'`

Error to be minimized in the loss function during estimation, specified as the comma-separated pair consisting of `'Focus'` and one of the following values:

- `'prediction'` — The one-step ahead prediction error between measured and predicted outputs is minimized during estimation. As a result, the estimation focuses on producing a good predictor model.
- `'simulation'` — The simulation error between measured and simulated outputs is minimized during estimation. As a result, the estimation focuses on making a good fit for simulation of model response with the current inputs.

The `Focus` option can be interpreted as a weighting filter in the loss function. For more information, see "Loss Function and Model Quality Metrics".

`'WeightingFilter'` — Weighting prefilter

[ ] (default) | vector | matrix | cell array | linear system

Weighting prefilter applied to the loss function to be minimized during estimation. To understand the effect of `WeightingFilter` on the loss function, see "Loss Function and Model Quality Metrics".

Specify `WeightingFilter` as one of the following values:

- `[]` — No weighting prefilter is used.
- Passbands — Specify a row vector or matrix containing frequency values that define desired passbands. You select a frequency band where the fit between estimated model and estimation data is optimized. For example, `[wl,wh]`, where `wl` and `wh` represent lower and upper limits of a passband. For a matrix with several rows defining frequency passbands, `[w1l,w1h;w2l,w2h;w3l,w3h;...]`, the estimation algorithm uses the union of the frequency ranges to define the estimation passband.

  Passbands are expressed in `rad/TimeUnit` for time-domain data and in `FrequencyUnit` for frequency-domain data, where `TimeUnit` and `FrequencyUnit` are the time and frequency units of the estimation data.
- SISO filter — Specify a single-input-single-output (SISO) linear filter in one of the following ways:

  - A SISO LTI model
  - `{A,B,C,D}` format, which specifies the state-space matrices of a filter with the same sample time as estimation data.
  - `{numerator,denominator}` format, which specifies the numerator and denominator of the filter as a transfer function with same sample time as estimation data.

  This option calculates the weighting function as a product of the filter and the input spectrum to estimate the transfer function.

`'EnforceStability'` — Control whether to enforce stability of model
`false` (default) | `true`

Control whether to enforce stability of estimated model, specified as the comma-separated pair consisting of `'EnforceStability'` and either `true` or `false`.

Data Types: `logical`

`'EstCovar'` — Control whether to generate parameter covariance data
`true` (default) | `false`

Controls whether parameter covariance data is generated, specified as `true` or `false`.

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

`'Display'` — Specify whether to display the estimation progress
`'off'` (default) | `'on'`

Specify whether to display the estimation progress, specified as one of the following values:

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

`'InputOffset'` — Removal of offset from time-domain input data during estimation
`[ ]` (default) | vector of positive integers | matrix

Removal of offset from time-domain input data during estimation, specified as the comma-separated pair consisting of `'InputOffset'` and one of the following:

- A column vector of positive integers of length *Nu*, where *Nu* is the number of inputs.
- `[]` — Indicates no offset.
- *Nu*-by-*Ne* matrix — For multi-experiment data, specify `InputOffset` as an *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.

`'OutputOffset'` — Removal of offset from time-domain output data during estimation

[ ] (default) | vector | matrix

Removal of offset from time-domain output data during estimation, specified as the comma-separated pair consisting of `'OutputOffset'` and one of the following:

- A column vector of length *Ny*, where *Ny* is the number of outputs.
- `[]` — Indicates no offset.
- *Ny*-by-*Ne* matrix — For multi-experiment 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.

`'Regularization'` — Options for regularized estimation of model parameters

structure

Options for regularized estimation of model parameters. For more information on regularization, see “Regularized Estimates of Model Parameters”.

`Regularization` is a structure with the following fields:

- `Lambda` — Constant that determines the bias versus variance tradeoff.

  Specify a positive scalar to add the regularization term to the estimation cost.

  The default value of zero implies no regularization.

  Default: 0
- `R` — Weighting matrix.

  Specify a vector of nonnegative numbers or a square positive semi-definite matrix. The length must be equal to the number of free parameters of the model.

For black-box models, using the default value is recommended. For structured and grey-box models, you can also specify a vector of `np` positive numbers such that each entry denotes the confidence in the value of the associated parameter.

The default value of 1 implies a value of `eye(npfree)`, where `npfree` is the number of free parameters.

Default: 1

- `Nominal` — The nominal value towards which the free parameters are pulled during estimation.

  The default value of zero implies that the parameter values are pulled towards zero. If you are refining a model, you can set the value to `'model'` to pull the parameters towards the parameter values of the initial model. The initial parameter values must be finite for this setting to work.

  Default: 0

`'SearchMethod'` — Search method used for iterative parameter estimation

`'auto'` (default) | `'gn'` | `'gna'` | `'lm'` | `'lsqnonlin'` | `'grad'`

Search method used for iterative parameter estimation, specified as one of the following values:

- `'gn'` — The subspace Gauss-Newton direction. Singular values of the Jacobian matrix less than `GnPinvConst*eps*max(size(J))*norm(J)` are discarded when computing the search direction. J is the Jacobian matrix. The Hessian matrix is approximated by J^TJ. 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 `gamma*max(sv)` of the Hessian are ignored, 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*LMStep` each time a search is successful without any bisections.
- `'lm'` — Uses the Levenberg-Marquardt method so that the next parameter value is `-pinv(H+d*I)*grad` from the previous one. 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 handle only the Trace criterion.
- `'grad'` — The steepest descent gradient search method.
- `'auto'` — The algorithm chooses one of the preceding options. The descent direction is calculated using `'gn'`, `'gna'`, `'lm'`, and `'grad'` successively at each iteration. The iterations continue until a sufficient reduction in error is achieved.

`'SearchOption'` — Option set for the search algorithm

search option set

Option set for the search algorithm with fields that depend on the value of `SearchMethod`.

SearchOption structure when SearchMethod is specified as 'gn', 'gna', 'lm', 'grad', or 'auto'

Field Name	Description
`Toleran`	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 stop. The estimate of the expected loss-function improvement at the next iteration is based on the Gauss-Newton vector computed for the current parameter value. **Default:** `0.01`
`MaxIter`	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 start-up procedure. Use `sys.Report.Termination.Iterations` to get the actual number of iterations during an estimation, where *sys* is an `idtf` model. **Default:** 20
`Advance`	Advanced search settings.

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>Specified as a structure with the following fields:
<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td>`GnPinvCon`</td><td>Singular values of the Jacobian matrix that are smaller than `GnPinvConst*max(size(J)*norm(J)*eps)` are discarded when computing the search direction. Applicable when `SearchMethod` is `'gn'`.
`GnPinvConst` must be a positive, real value.
Default: `10000`</td></tr>
<tr><td>`InitGnaTo`</td><td>Initial value of *gamma*. Applicable when `SearchMethod` is `'gna'`.
Default: `0.0001`</td></tr>
<tr><td>`LMStartVa`</td><td>Starting value of search-direction length *d* in the Levenberg-Marquardt method. Applicable when `SearchMethod` is `'lm'`.
Default: `0.001`</td></tr>
<tr><td>`LMStep`</td><td>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`</td></tr>
<tr><td>`MaxBisect`</td><td>Maximum number of bisections used by the line search along the search direction.
Default: `25`</td></tr>
<tr><td>`MaxFunEva`</td><td>Iterations stop if the number of calls to the model file exceeds this value.
`MaxFunEvals` must be a positive, integer value.
Default: `Inf`</td></tr>
<tr><td>`MinParCha`</td><td>Smallest parameter update allowed per iteration.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td><code>MinParChange</code> must be a positive, real value.
Default: <code>0</code></td></tr>
<tr><td><code>RelImprov</code></td><td>Iterations stop if the relative improvement of the criterion function is less than <code>RelImprovement</code>.
<code>RelImprovement</code> must be a positive, integer value.
Default: <code>0</code></td></tr>
<tr><td><code>StepReduc</code></td><td>Suggested parameter update is reduced by the factor <code>StepReduction</code> after each try. This reduction continues until either <code>MaxBisections</code> tries are completed or a lower value of the criterion function is obtained.
<code>StepReduction</code> must be a positive, real value that is greater than 1.
Default: <code>2</code></td></tr>
</table>
</td></tr>
</table>

SearchOption structure when SearchMethod is specified as 'lsqnonlin'

Field Name	Description
`TolFun`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values. The value of `TolFun` is the same as that of `opt.SearchOption.Advanced.TolFun`. **Default:** `1e-5`
`TolX`	Termination tolerance on the estimated parameter values. The value of `TolX` is the same as that of `opt.SearchOption.Advanced.TolX`. **Default:** `1e-6`

Field Name	Description
`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 `opt.SearchOption.Advanced.MaxIter`. **Default:** `20`
`Advance`	Options set for `lsqnonlin`. For more information, see the Optimization Options table in "Optimization Options". Use `optimset('lsqnonlin')` to create an options set for `lsqnonlin`, and then modify it to specify its various options.

`'Advanced'` — Additional advanced options

structure

Additional advanced options, specified as 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 loss function. The standard deviation is estimated robustly as the median of the absolute deviations from the median of the prediction errors, divided by `0.7`. For more information on robust norm choices, see section 15.2 of [2].

  `ErrorThreshold = 0` disables robustification and leads to a purely quadratic loss function. When estimating with frequency-domain data, the software sets `ErrorThreshold` to zero. For time-domain data that contains outliers, try setting `ErrorThreshold` to `1.6`.

  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 condition.

  The initial condition is estimated when

  $$\frac{\left\|y_{p,z} - y_{meas}\right\|}{\left\|y_{p,e} - y_{meas}\right\|} > \mathrm{AutoInitThreshold}$$

  - y_{meas} is the measured output.
  - $y_{p,z}$ is the predicted output of a model estimated using zero initial states.
  - $y_{p,e}$ is the predicted output of a model estimated using estimated initial states.

  Applicable when `InitialCondition` is `'auto'`.

  Default: `1.05`

Output Arguments

`opt` — Options set for `bj`

`bjOptions` option set

Option set for `bj`, returned as an `bjOptions` option set.

Examples

Create Default Options Set for Box-Jenkins Estimation

```
opt = bjOptions;
```

Specify Options for Box-Jenkins Estimation

Create an options set for `bj` using zero initial conditions for estimation. 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';
```

More About

- “Loss Function and Model Quality Metrics”

References

[1] 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.

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

See Also

`bj` | `idfilt`

Introduced in R2012a

blkdiag

Block-diagonal concatenation of models

Syntax

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

Description

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

$$\begin{bmatrix} sys1 & 0 & .. & 0 \\ 0 & sys2 & . & : \\ : & . & . & 0 \\ 0 & .. & 0 & sysN \end{bmatrix}$$

`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

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`

Introduced in R2009a

bode

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

Syntax

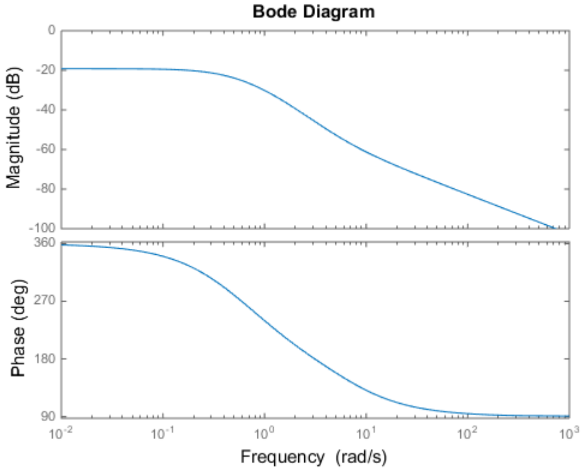
```
bode(sys)
bode(sys1,...,sysN)
bode(sys1,PlotStyle1,...,sysN,PlotStyleN)
bode(...,w)
[mag,phase] = bode(sys,w)
[mag,phase,wout] = bode(sys)
[mag,phase,wout,sdmag,sdphase] = bode(sys)
```

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(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)` specifies a color, linestyle, and marker for each system in the plot.

`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 vector of one, two, or three characters. The characters can appear in any order. For example, `'r:'` specifies a red dotted line. For more information about configuring the `PlotStyle` argument, see “Specify Line Style, Color, 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:

```
magdb = 20*log10(mag)
```

`phase`

Phase of the system response in degrees, returned as a 3-D array with dimensions are (number of outputs) × (number of inputs) × (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 `[]`.

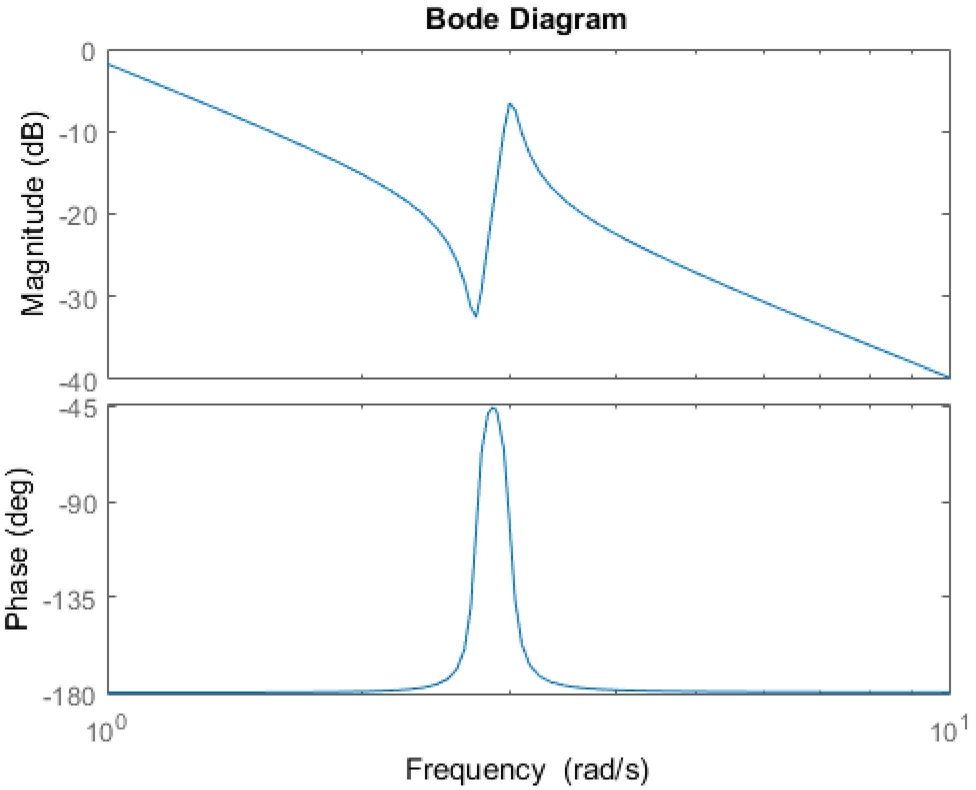
Examples

Bode Plot of Dynamic System

Create a Bode plot of the following continuous-time SISO dynamic system.

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

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

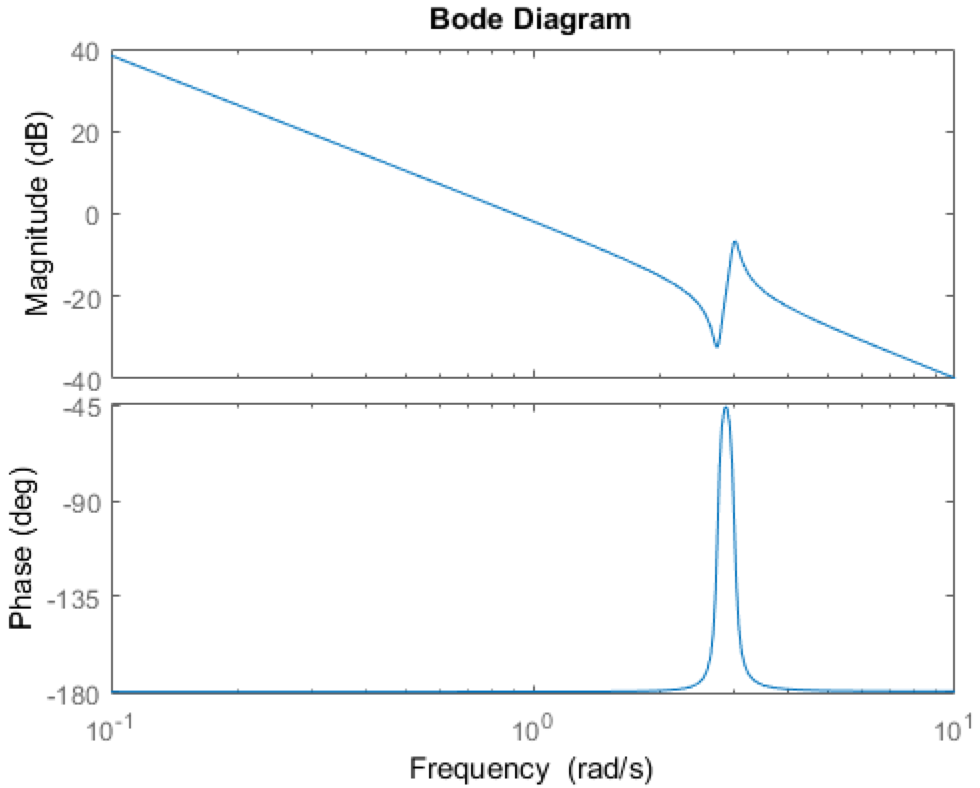


`bode` automatically selects the plot range based on the system dynamics.

Bode Plot at Specified Frequencies

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

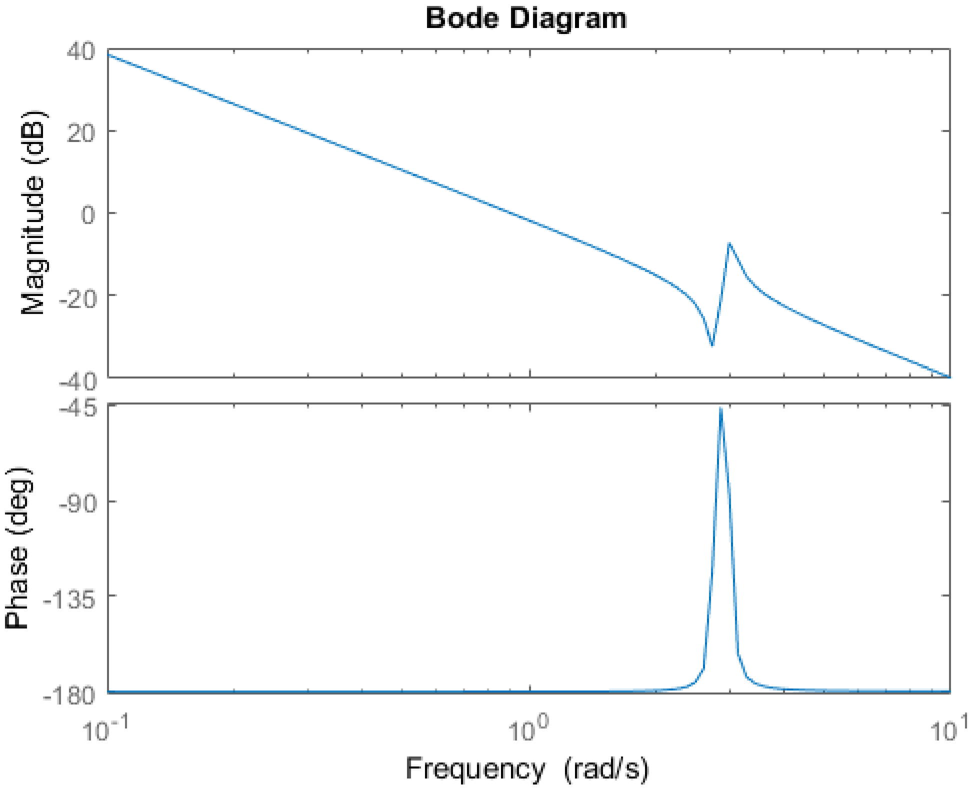
```
H = tf([1 0.1 7.5],[1 0.12 9 0 0]);
bode(H,{0.1,10})
```



The cell array `{0.1,10}` specifies the minimum and maximum frequency values in the Bode plot. When you provide frequency bounds in this way, the software selects intermediate points for frequency response data.

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

```
w = logspace(-1,1,100);
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

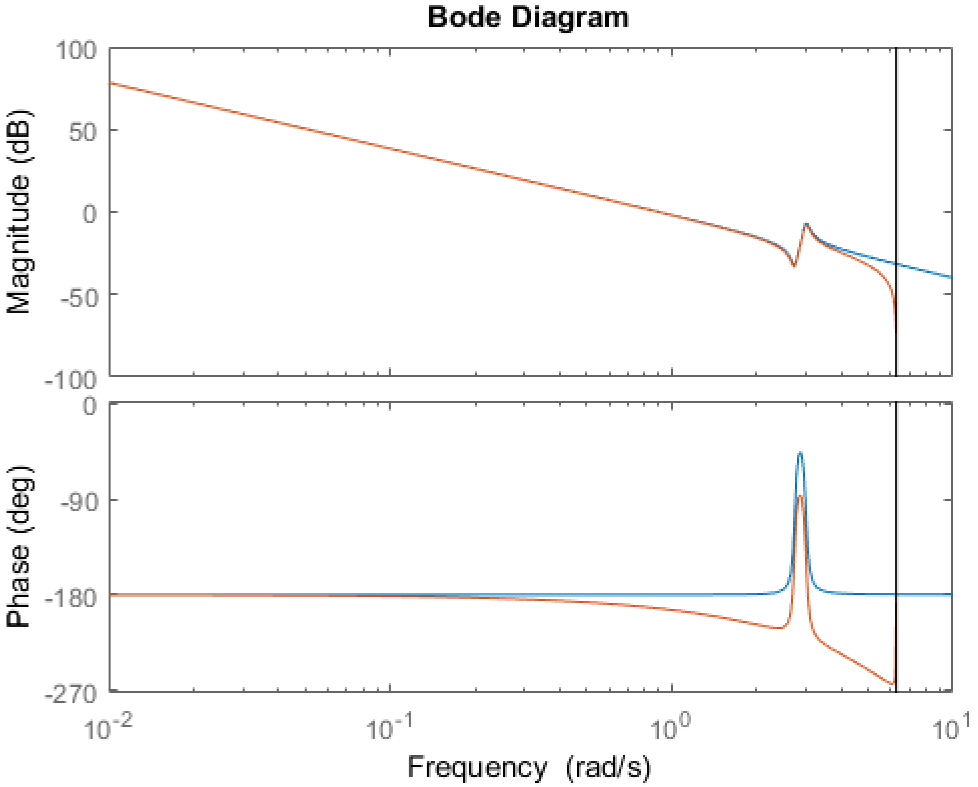
Compare the frequency response of a continuous-time system to an equivalent discretized system on the same Bode plot.

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');
```

Create a Bode plot that displays both systems.

```
bode(H,Hd)
```

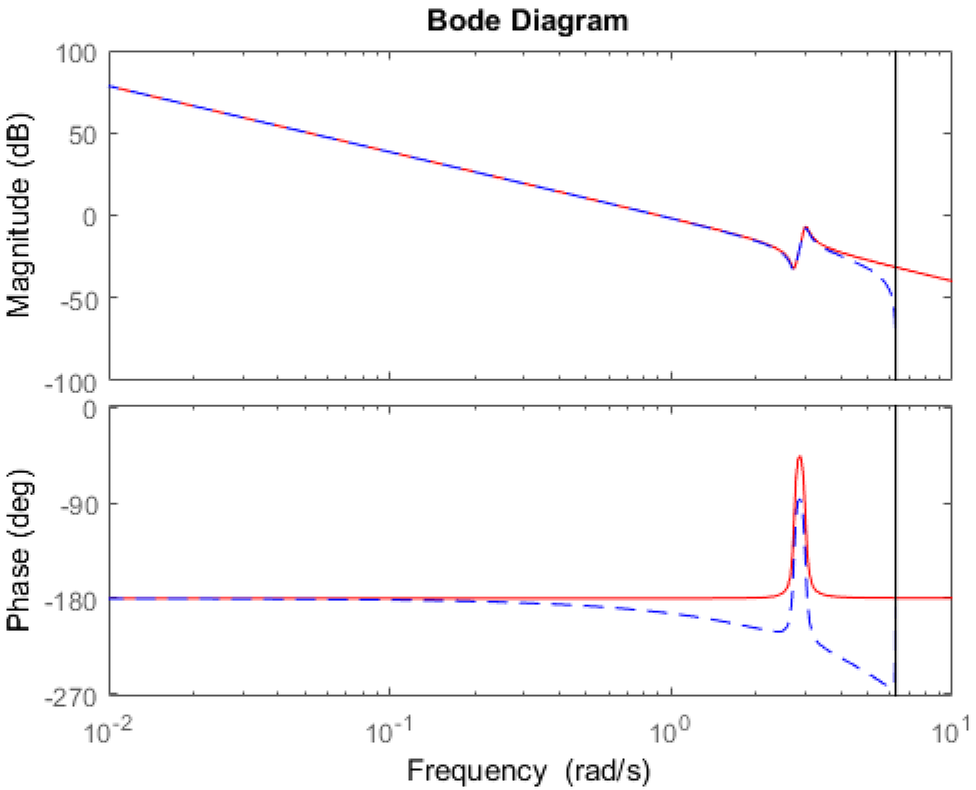


The Bode plot of a discrete-time system includes a vertical line marking the Nyquist frequency of the system.

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



The first `PlotStyle`, `'r'`, specifies a solid red line for the response of `H`. The second `PlotStyle`, `'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 nonparametric model identified using the same data.

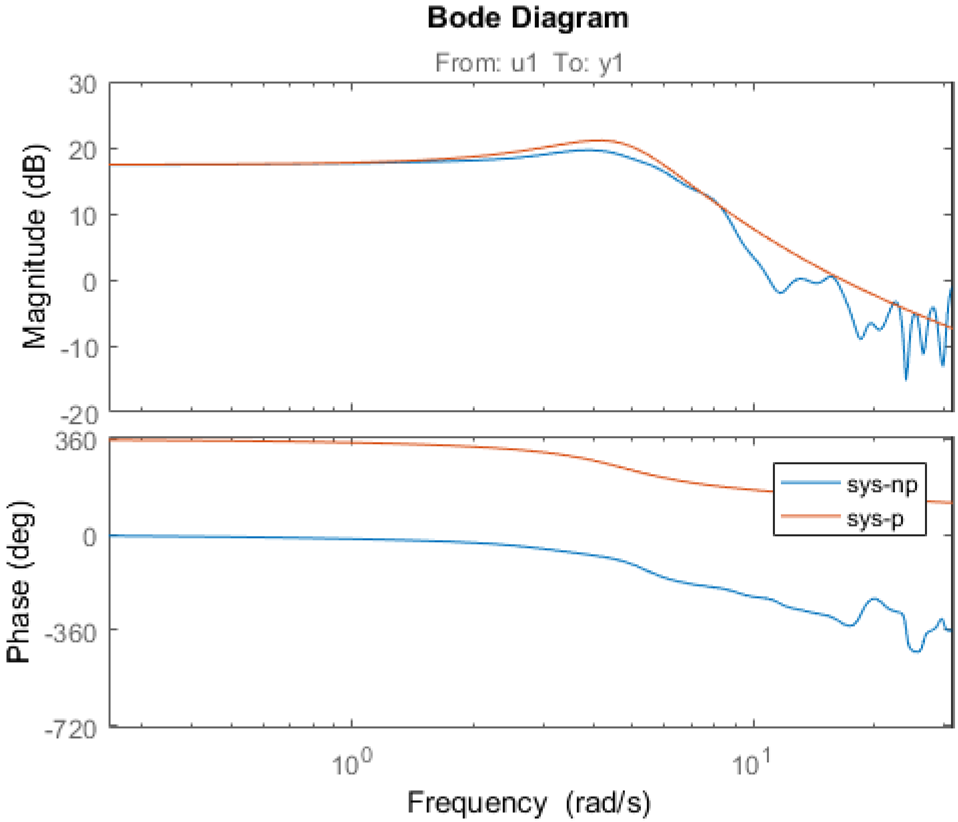
Identify parametric and non-parametric models based on data.

```
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.

Create a Bode plot that includes both systems.

```
bode(sys_np,sys_p,w);
legend('sys-np','sys-p')
```



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 3σ plot of the response uncertainty.

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

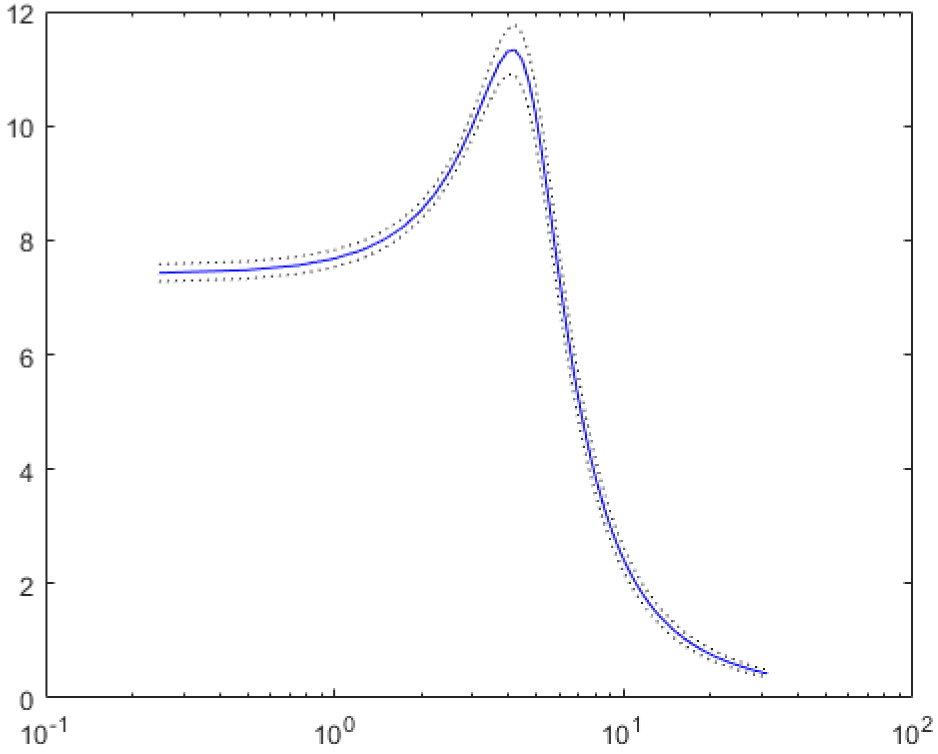
```
load iddata2 z2;
sys_p = tfest(z2,2);
w = linspace(0,10*pi,128);
```

```
[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.

Use the standard deviation data to create a 3 σ 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:');
```



You can also right-click the Bode plot and select **Characteristics > Confidence Region** or use the `showConfidence` command to plot the confidence region.

Alternatives

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

More About

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

- “Dynamic System Models”

See Also

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

Introduced before R2006a

bodemag

Bode magnitude response of LTI models

Syntax

```
bodemag(sys)
bodemag(sys,{wmin,wmax})
bodemag(sys,w)
bodemag(sys1,sys2,...,sysN,w)
```

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. For example:

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

See Also

`bode` | Linear System Analyzer

Introduced in R2012a

bodeoptions

Create list of Bode plot options

Syntax

```
P = bodeoptions
P = bodeoptions('cstprefs')
```

Description

`P = bodeoptions` returns a default set of plot options for use with the `bodeplot`. You can use these options to customize the Bode plot appearance using the command line. This syntax is useful when you want to write a script to generate plots that look the same regardless of the preference settings of the MATLAB session in which you run the script.

`P = bodeoptions('cstprefs')` initializes the plot options with the options you selected in the Control System and System Identification Toolbox Preferences Editor. For more information about the editor, see “Toolbox Preferences Editor” in the User's Guide documentation. This syntax is useful when you want to change a few plot options but otherwise use your default preferences. A script that uses this syntax may generate results that look different when run in a session with different preferences.

The following table summarizes the Bode plot options.

Option	Description
`Title`, `XLabel`, `YLabel`	Label text and style, specified as a structure with the following fields: • `String` — Label text, specified as a character vector, for example `'Amplitude'`. • `FontSize` — **Default**: 8 • `FontWeight` — **Default**: `'Normal'` • `Font Angle` — **Default**: `'Normal'` • `Color` — Vector of RGB values ranging from 0 to 1. **Default**: `[0,0,0]` • `Interpreter` — **Default**: `'tex'`

Option	Description
`TickLabel`	Tick label style, specified as a structure with the following fields: • `FontSize` **Default**: 8 • `FontWeight` — **Default**: `'Normal'` • `Font Angle` — **Default**: `'Normal'` • `Color` — Vector of RGB values ranging from 0 to 1. **Default**: `[0,0,0]`
`Grid`	Show or hide the grid Specified as one of the following values: `'off'` | `'on'` **Default**: `'off'`
`GridColor`	Color of the grid lines Specified as one of the following: Vector of RGB values in the range `[0,1]` | character vector of color name | `'none'`. For example, for yellow color, specify as one of the following: `[1 1 0]`, `'yellow'`, or `'y'`. **Default**: `[0.15,0.15,0.15]`
`XlimMode`, `YlimMode`	Axis limit modes. **Default**: `'auto'`
`Xlim`, `Ylim`	Axes limits, specified as an array of the form `[min,max]`
`IOGrouping`	Grouping of input-output pairs Specified as one of the following values: `'none'` `|'inputs'|'outputs'|'all'` **Default**: `'none'`
`InputLabels`, `OutputLabels`	Input and output label styles
`InputVisible`, `OutputVisible`	Visibility of input and output channels
`ConfidenceRegionNumb`	Number of standard deviations to use to plotting the response confidence region (identified models only). **Default**: `1`.

Option	Description
`FreqUnits`	Frequency units, specified as one of the following values: • `'Hz'` • `'rad/second'` • `'rpm'` • `'kHz'` • `'MHz'` • `'GHz'` • `'rad/nanosecond'` • `'rad/microsecond'` • `'rad/millisecond'` • `'rad/minute'` • `'rad/hour'` • `'rad/day'` • `'rad/week'` • `'rad/month'` • `'rad/year'` • `'cycles/nanosecond'` • `'cycles/microsecond'` • `'cycles/millisecond'` • `'cycles/hour'` • `'cycles/day'` • `'cycles/week'` • `'cycles/month'` • `'cycles/year'`
`FreqScale`	Frequency scale Specified as one of the following values: `'linear' | 'log'` **Default**: `'log'`

Option	Description
`MagUnits`	Magnitude units Specified as one of the following values: `'dB'` | `'abs'` **Default**: `'dB'`
`MagScale`	Magnitude scale Specified as one of the following values: `'linear'` | `'log'` **Default**: `'linear'`
`MagVisible`	Magnitude plot visibility Specified as one of the following values: `'on'` | `'off'` **Default**: `'on'`
`MagLowerLimMode`	Enables a lower magnitude limit Specified as one of the following values: `'auto'` | `'manual'` **Default**: `'auto'`
`MagLowerLim`	Specifies the lower magnitude limit
`PhaseUnits`	Phase units Specified as one of the following values: `'deg'` | `'rad'` **Default**: `'deg'`
`PhaseVisible`	Phase plot visibility Specified as one of the following values: `'on'` | `'off'` **Default**: `'on'`
`PhaseWrapping`	Enables phase wrapping Specified as one of the following values: `'on'` | `'off'` When you set `PhaseWrapping` to `'on'`, the plot wraps accumulated phase at the value specified by the `PhaseWrappingBranch` property. **Default**: `'off'`
`PhaseWrappingBranch`	Phase value at which the plot wraps accumulated phase when `PhaseWrapping` is set to `'on'`. **Default**: –180 (phase wraps into the interval [–180º,180º))
`PhaseMatching`	Enables phase matching Specified as one of the following values: `'on'` | `'off'` **Default**: `'off'`
`PhaseMatchingFreq`	Frequency for matching phase
`PhaseMatchingValue`	The value to which phase responses are matched closely

Examples

Create Bode Plot with Custom Settings

Create a Bode plot that suppresses the phase plot and uses frequency units Hz instead of the default radians/second. Otherwise, the plot uses the settings that are saved in the toolbox preferences.

First, create an options set based on the toolbox preferences.

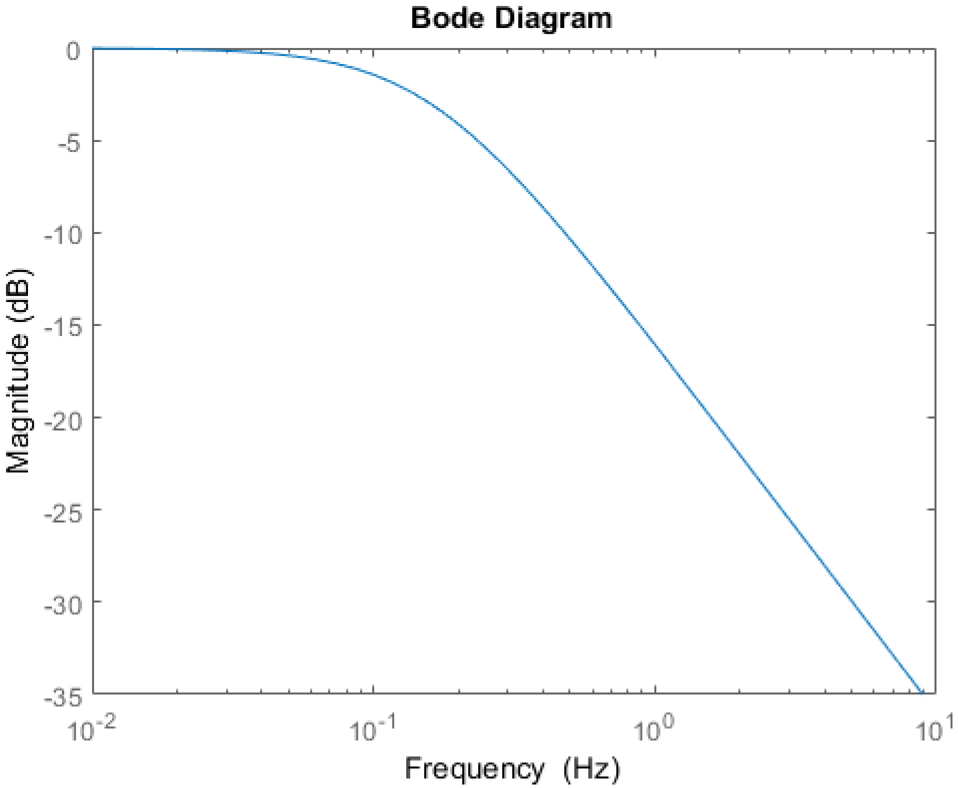
```
opts = bodeoptions('cstprefs');
```

Change properties of the options set.

```
opts.PhaseVisible = 'off';
opts.FreqUnits = 'Hz';
```

Create a plot using the options.

```
h = bodeplot(tf(1,[1,1]),opts);
```



Depending on your own toolbox preferences, the plot you obtain might look different from this plot. Only the properties that you set explicitly, in this example `PhaseVisible` and `FreqUnits`, override the toolbox preferences.

Custom Plot Settings Independent of Preferences

Create a Bode plot that uses 14-point red text for the title. This plot should look the same, regardless of the preferences of the MATLAB session in which it is generated.

First, create a default options set.

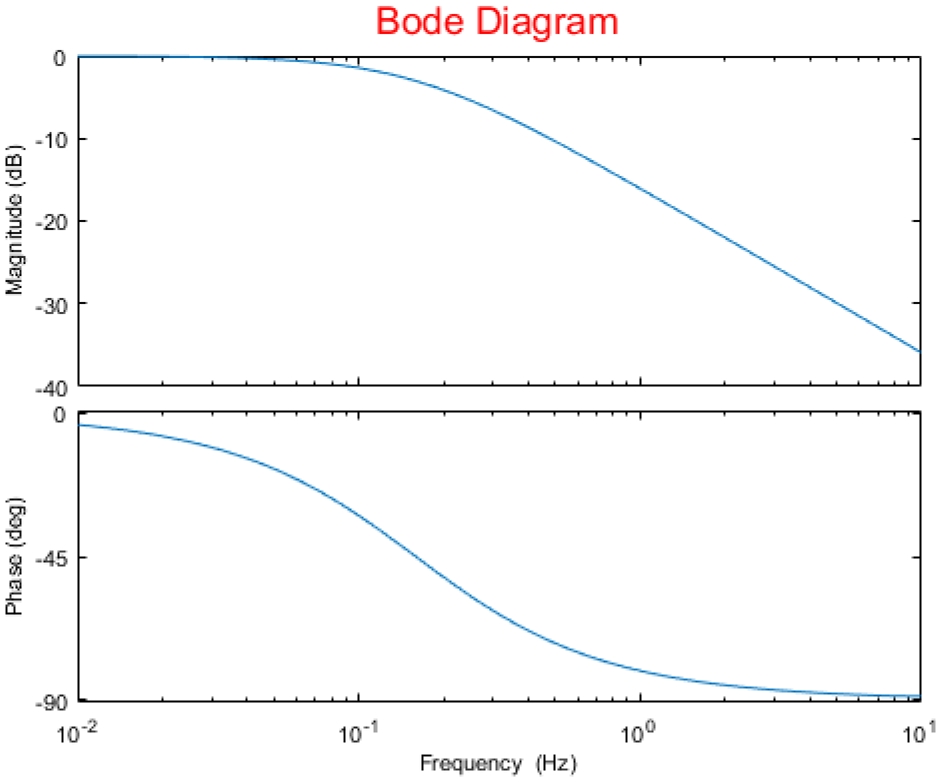
```
opts = bodeoptions;
```

Change properties of the options set.

```
opts.Title.FontSize = 14;
opts.Title.Color = [1 0 0];
opts.FreqUnits = 'Hz';
```

Create a plot using the options.

```
h = bodeplot(tf(1,[1,1]),opts);
```



Because `opts` begins with a fixed set of options, the plot result is independent of the toolbox preferences of the MATLAB session.

See Also

`showConfidence` | `bode` | `bodeplot` | `getoptions` | `setoptions`

Introduced in R2012a

bodeplot

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 “Match Phase at Specified Frequency” on page 1-146 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.

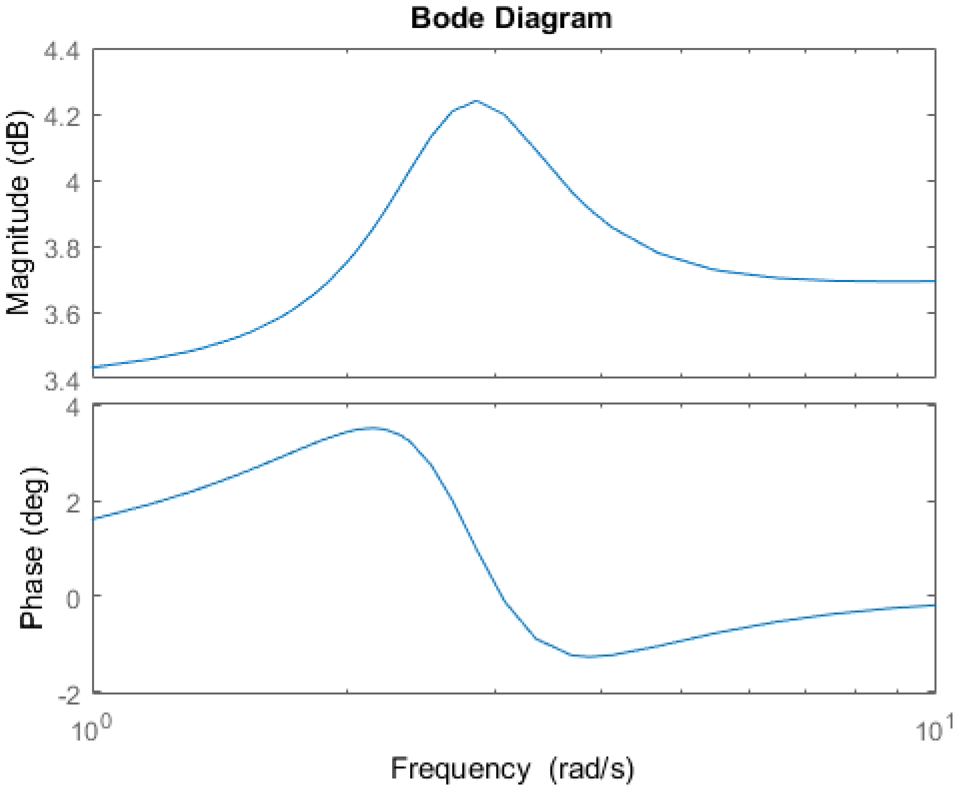
See `logspace` to generate logarithmically spaced frequency vectors.

Examples

Change Bode Plot Options with Plot Handle

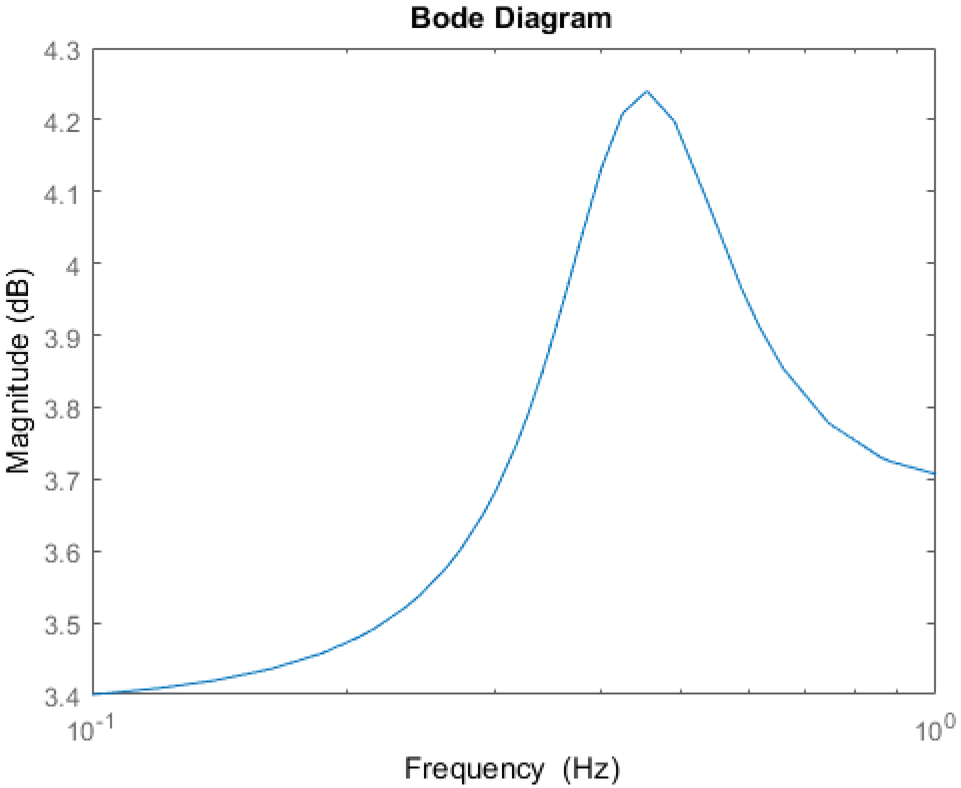
Generate a Bode plot.

```
sys = rss(5);
h = bodeplot(sys);
```



Change the units to Hz and suppress the phase plot. To do so, edit properties of the plot handle, `h`.

```
setoptions(h,'FreqUnits','Hz','PhaseVisible','off');
```

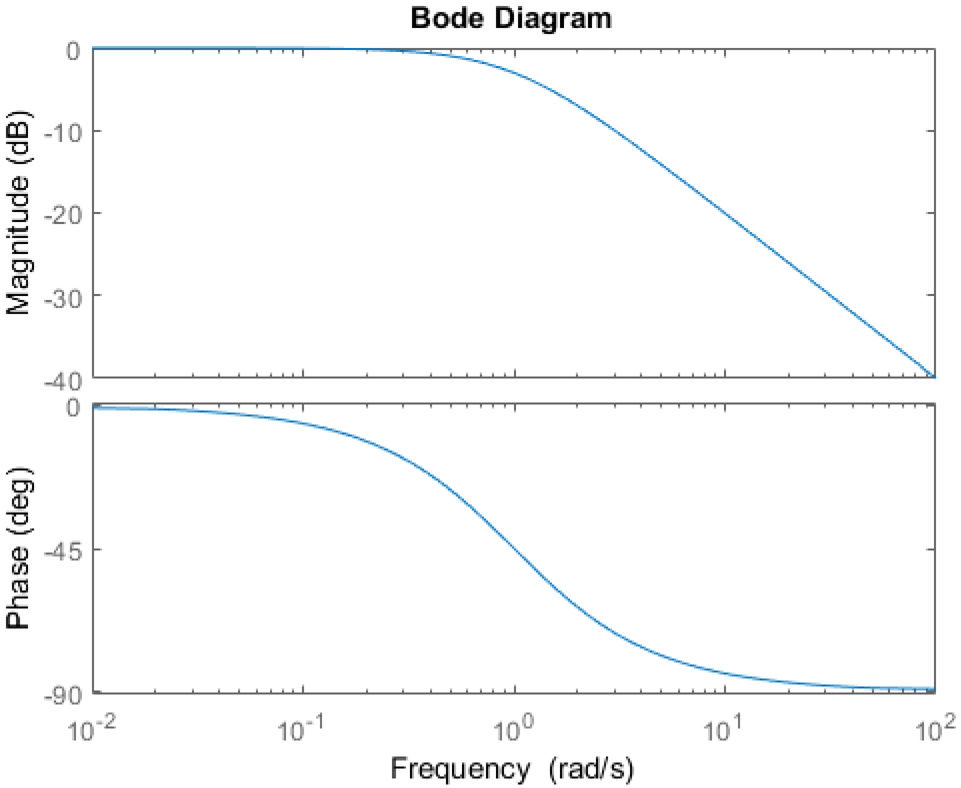


The plot automatically updates when you call `setoptions`.

Match Phase at Specified Frequency

Create a Bode plot of a dynamic system.

```
sys = tf(1,[1 1]);
h = bodeplot(sys);
```

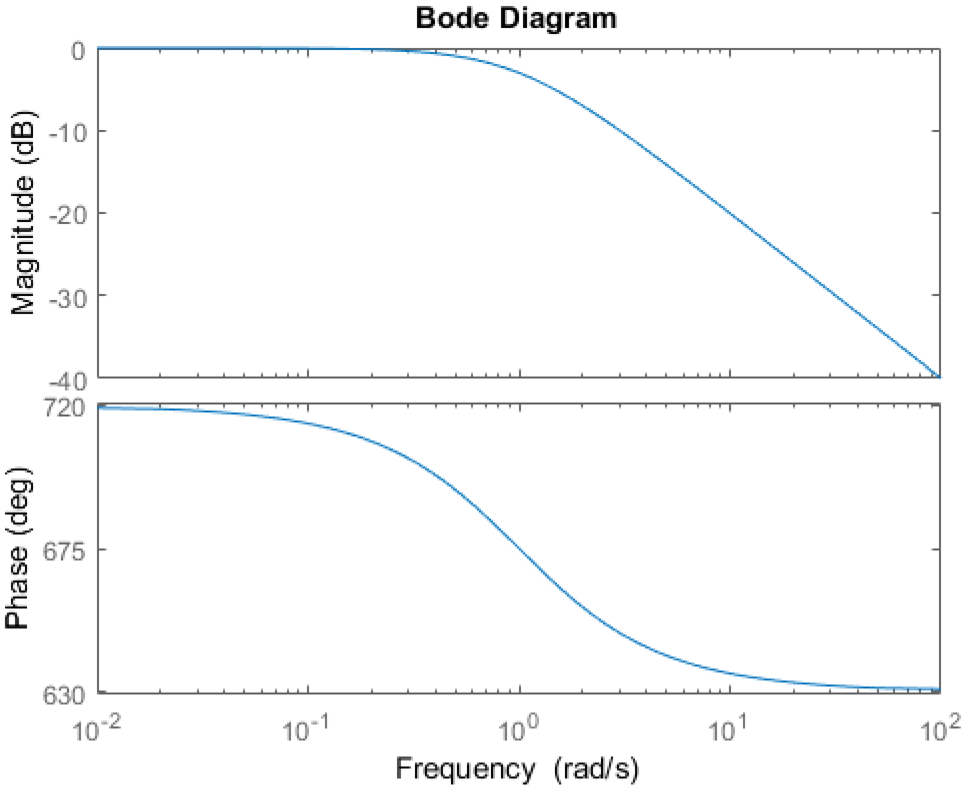


Fix the phase at 1 rad/s to 750 degrees. To do so, get the plot properties. Then alter the properties `PhaseMatchingFreq` and `PhaseMatchingValue` to match a phase to a specified frequency.

```
p = getoptions(h);
p.PhaseMatching = 'on';
p.PhaseMatchingFreq = 1;
p.PhaseMatchingValue = 750;
```

Update the plot.

```
setoptions(h,p);
```



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*360, where N is an integer, and so the plot is set to the nearest allowable phase, namely 675 degrees (or 2*360 - 45 = 675).

Display Confidence Regions of Identified Models

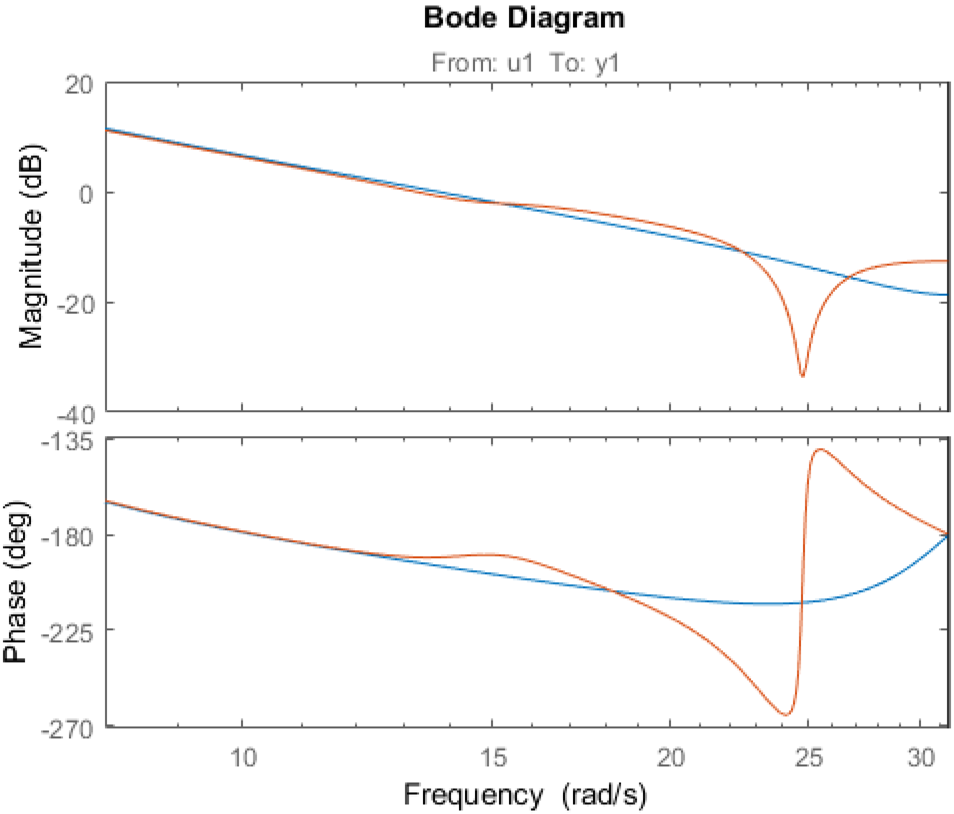
Compare the frequency responses of identified state-space models of order 2 and 6 along with their 2 σ confidence regions.

```
load iddata1
sys1 = n4sid(z1, 2);
```

```
sys2 = n4sid(z1, 6);
```

Both models produce about 70% fit to data. However, `sys2` shows higher uncertainty in its frequency response, especially close to Nyquist frequency as shown by the plot:

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



Right-click the plot and select **Characteristics > Confidence Region** to turn on the confidence region characteristic. Alternatively, type `showConfidence(h)` to plot the confidence region.

Frequency Response of Identified Parametric and Nonparametric Models

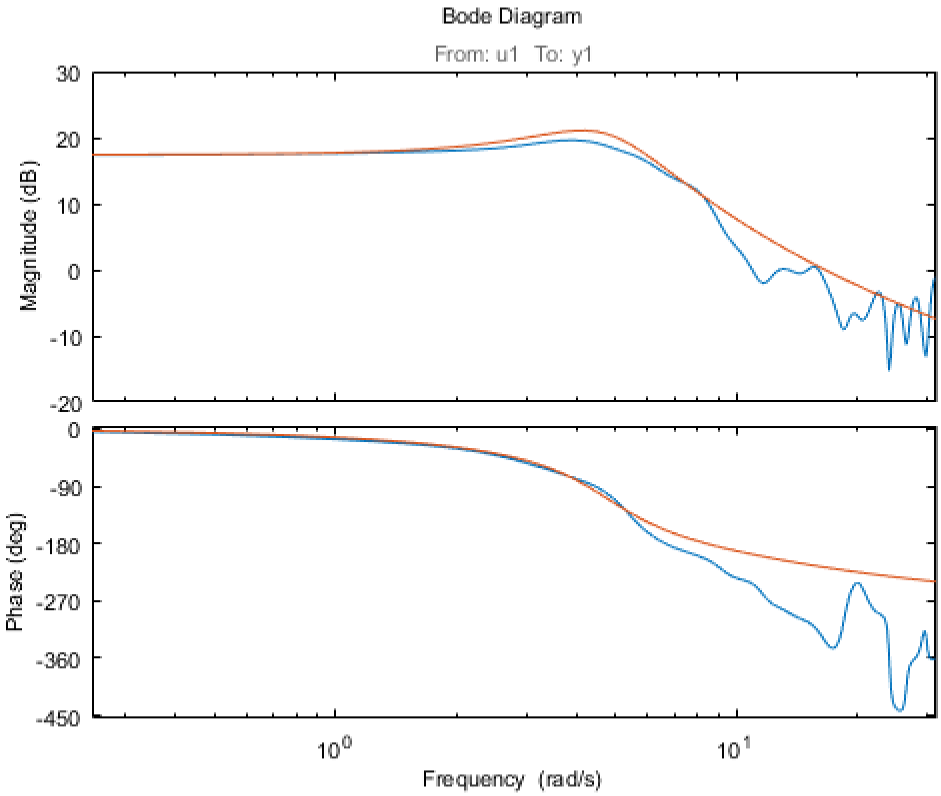
Compare the frequency response of a parametric model, identified from input/output data, to a nonparametric model identified using the same data. Identify parametric and non-parametric models based on data.

```
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 nonparametric identified model. `sys_p` is a parametric identified model.

Create a Bode plot that includes both systems.

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



More About

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”.

See Also

`showConfidence` | `bode` | `bodeoptions` | `getoptions` | `setoptions`

Introduced before R2006a

c2d

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)`.

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 whose `FunctionType` 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`

Discretization method, specified as one of the following values:

- `'zoh'` — Zero-order hold (default). Assumes the control inputs are piecewise constant over the sample time `Ts`.
- `'foh'` — Triangle approximation (modified first-order hold). Assumes the control inputs are piecewise linear over the sample time `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 λ 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 λ/Ts.
- 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 a Transfer Function

Discretize the following continuous-time transfer function:

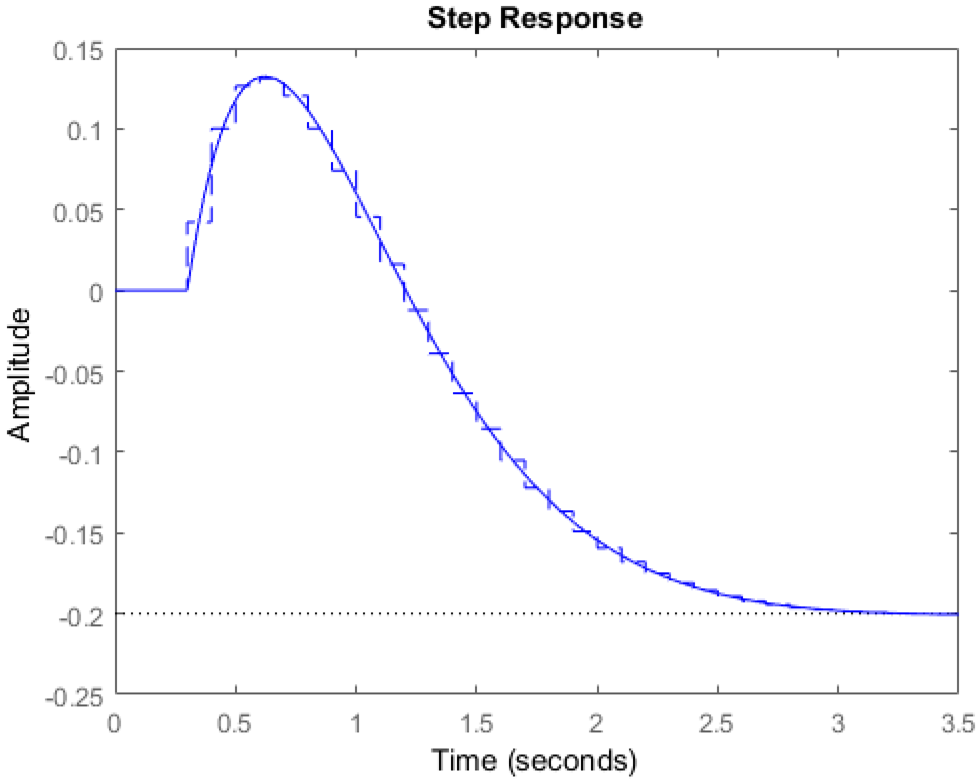
$$H(s) = e^{-0.3s} \frac{s-1}{s^2+4s+5}.$$

This system has an input delay of 0.3 s. Discretize the system using the triangle (first-order-hold) approximation with sample time `Ts` = 0.1 s.

```
H = tf([1 -1],[1 4 5],'InputDelay', 0.3);
Hd = c2d(H,0.1,'foh');
```

Compare the step responses of the continuous-time and discretized systems.

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



Discretize Model with Fractional Delay Absorbed into Coefficients

Discretize the following delayed transfer function using zero-order hold on the input, and a 10-Hz sampling rate.

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

```
h = tf(10,[1 3 10],'IODelay',0.25);
```

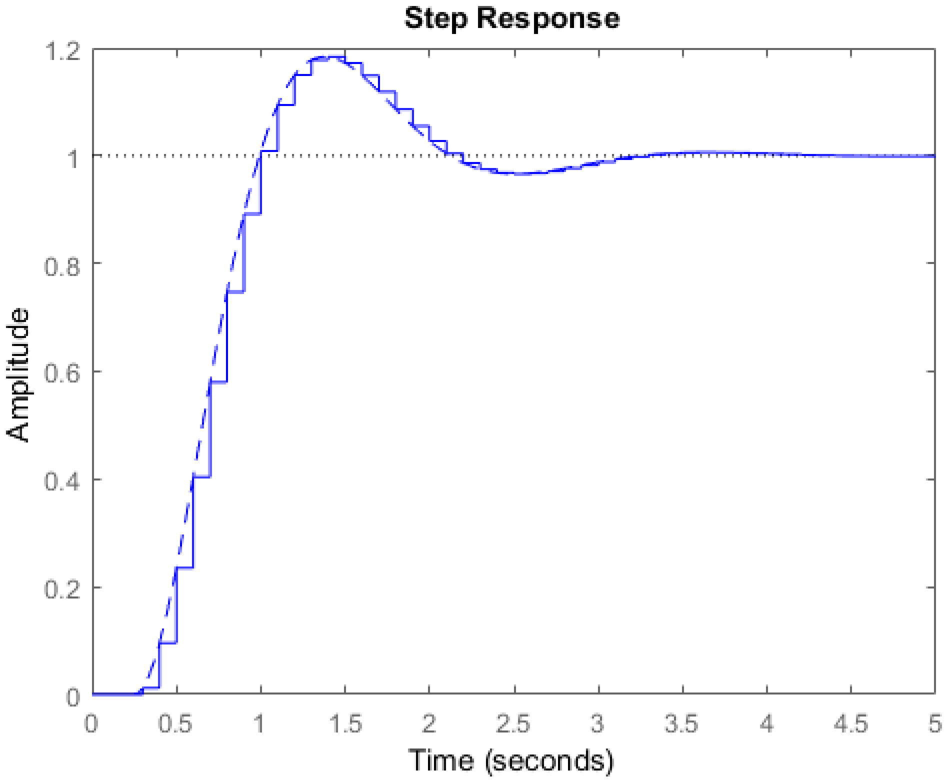
```
hd = c2d(h,0.1)
```

```
hd =
 
              0.01187 z^2 + 0.06408 z + 0.009721
  z^(-3) * ----------------------------------
                    z^2 - 1.655 z + 0.7408
 
Sample time: 0.1 seconds
Discrete-time transfer function.
```

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-time and discretized models.

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



Discretize Model With Approximated Fractional Delay

Create a continuous-time state-space model with two states and an input delay.

```
sys = ss(tf([1,2],[1,4,2]));
sys.InputDelay = 2.7
```

```
sys =

  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 (seconds): 2.7 
 
Continuous-time state-space model.
```

Discretize the model using the Tustin discretization method and a Thiran filter to model fractional delays. The sample time `Ts` = 1 second.

```
opt = c2dOptions('Method','tustin','FractDelayApproxOrder',3);
sysd1 = c2d(sys,1,opt)
```

```
sysd1 =
 
  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

Sample time: 1 seconds
Discrete-time state-space model.
```

The discretized model now contains three additional states `x3`, `x4`, and `x5` corresponding to a third-order Thiran filter. Since the time delay divided by the sample time is 2.7, the third-order Thiran filter (`'FractDelayApproxOrder'` = 3) can approximate the entire time delay.

Discretize Identified Model

Estimate a continuous-time transfer function, and discretize it.

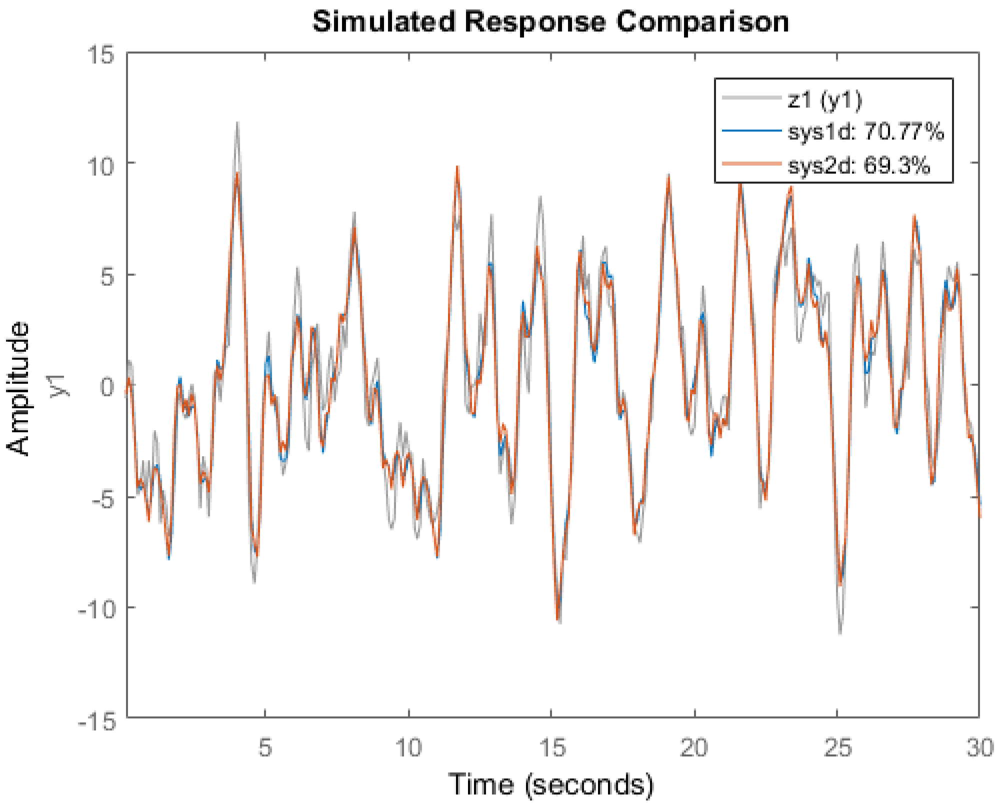
```
load iddata1
sys1c = tfest(z1,2);
sys1d = c2d(sys1c,0.1,'zoh');
```

Estimate a second order discrete-time transfer function.

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

Compare the response of the discretized continuous-time transfer function model, `sys1d`, and the directly estimated discrete-time model, `sys2d`.

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



The two systems are almost identical.

Build Predictor Model

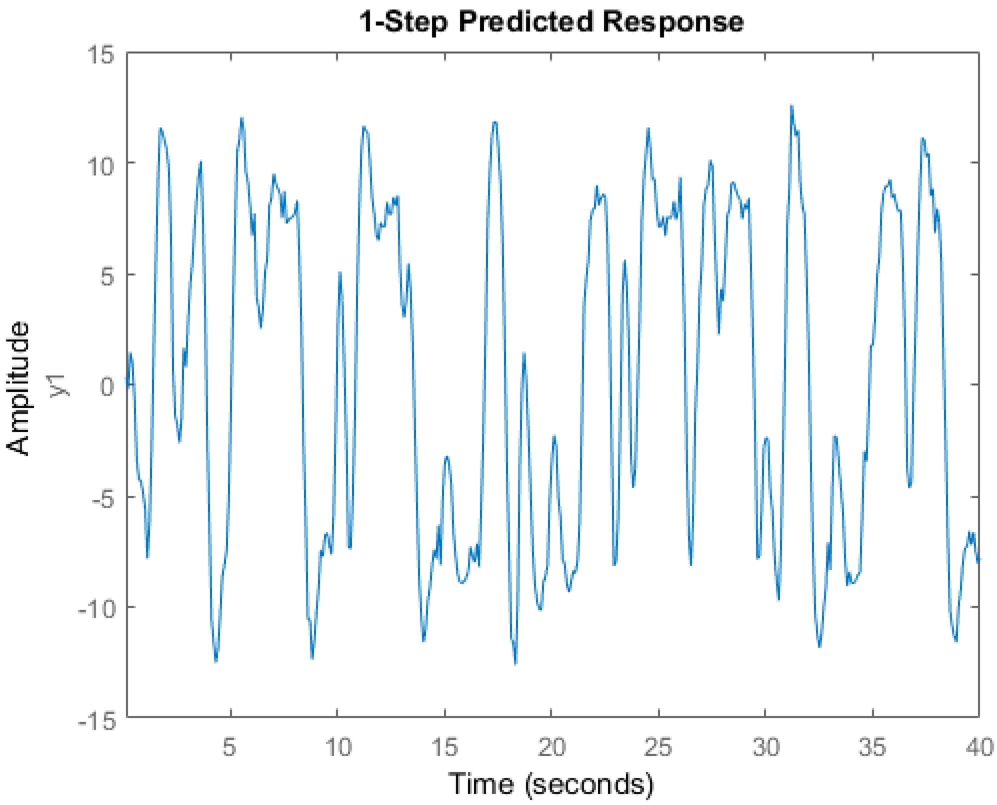
Discretize an identified state-space model to build a one-step ahead predictor of its response.

Create a continuous-time identified state-space model using estimation data.

```
load iddata2
sysc = ssest(z2,4);
```

Predict the 1-step ahead predicted response of `sysc`.

```
predict(sysc,z2)
```



Discretize the model.

```
sysd = c2d(sysc,0.1,'zoh');
```

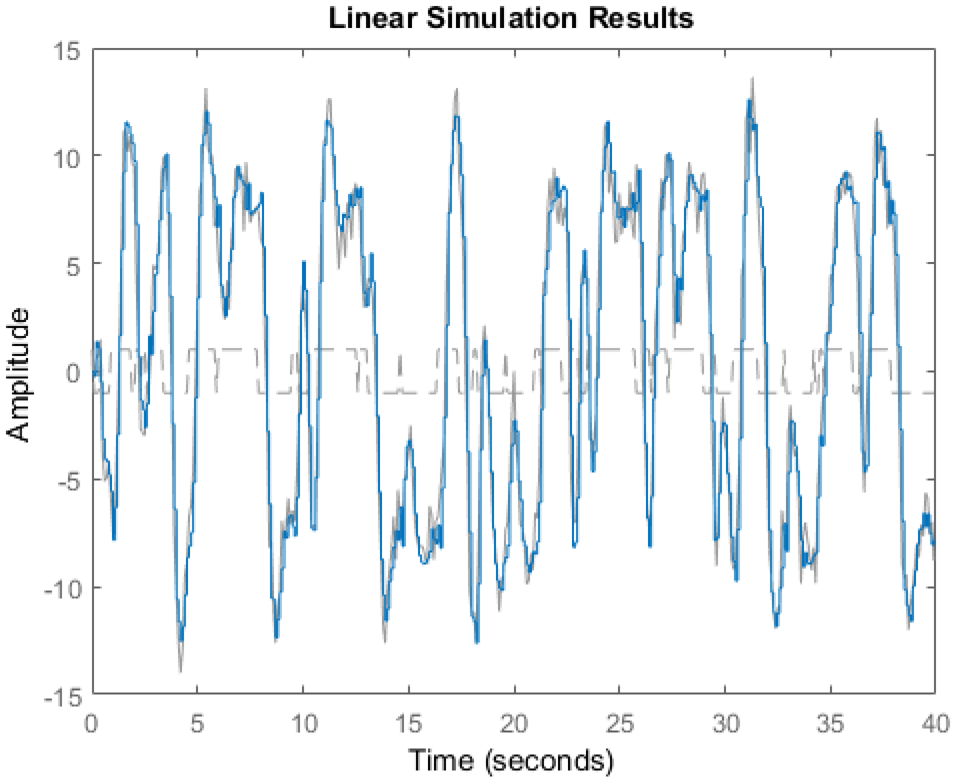
Build a predictor model from the discretized model, `sysd`.

```
[A,B,C,D,K] = idssdata(sysd);
Predictor = ss(A-K*C,[K B-K*D],C,[0 D],0.1);
```

`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`.

Simulate the predictor model to get the same response as the `predict` command.

```
lsim(Predictor,[z2.y,z2.u])
```



The simulation of the predictor model gives the same response as `predict(sysc,z2)`.

More About

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`.

Algorithms

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

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

See Also

`c2dOptions` | `d2c` | `d2d` | `thiran` | `translatecov`

Introduced before R2006a

c2dOptions

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 `c2d` assumes the control inputs are piecewise constant over the sample time `Ts`.
`'foh'`	Triangle approximation (modified first-order hold), where `c2d` assumes the control inputs are piecewise linear over the sample time `Ts`. (See [1], p. 228.)
`'impulse'`	Impulse-invariant discretization.
`'tustin'`	Bilinear (Tustin) approximation. By default, `c2d` discretizes with no prewarp and rounds any fractional time delays to the nearest multiple of the sample time. To include prewarp, use the `PrewarpFrequency` option. To approximate fractional time delays, use the`FractDelayApproxOrder` option.

`'matched'`	Zero-pole matching method. (See [1], p. 224.) By default, `c2d` rounds any fractional time delays to the nearest multiple of the sample time. To approximate fractional time delays, use the `FractDelayApproxOrder` 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 Tustin Discretization Method

Generate two random continuous-time state-space models.

```
sys1 = rss(3,2,2);
sys2 = rss(4,4,1);
```

Create an option set for `c2d` to use the Tustin discretization method and 3.4 rad/s prewarp frequency.

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

Discretize the models, `sys1` and `sys2`, using the same option set, but different sample times.

```
dsys1 = c2d(sys1,0.1,opt);
```

```
dsys2 = c2d(sys2,0.2,opt);
```

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`

Introduced in R2012a

canon

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

Canonical form of `csys`, specified as one of the 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.

Examples

Convert System To Modal Canonical Form

Consider a system with doubled poles and clusters of close poles:

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

Create a linear model of this system, and convert it to modal canonical form.

```
G = zpk([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.

```
Gc.A


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  -10.0001
```

Separate the two poles near $s = -10$ by increasing the value of the condition number of the block-diagonalizing transformation. The default value of the condition number is 1e8.

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


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         0
         0         0         0         0         0         0  -10.0001
```

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

```
format shortE
Gc2.B
```

```
ans =

   3.2000e-01
  -6.5691e-03
   5.4046e-02
  -1.9502e-01
   1.0637e+00
   3.2533e+05
   3.2533e+05
```

Convert System to Companion Canonical Form

Estimate a state-space model that is freely parameterized.

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

Convert the estimated model to companion canonical form.

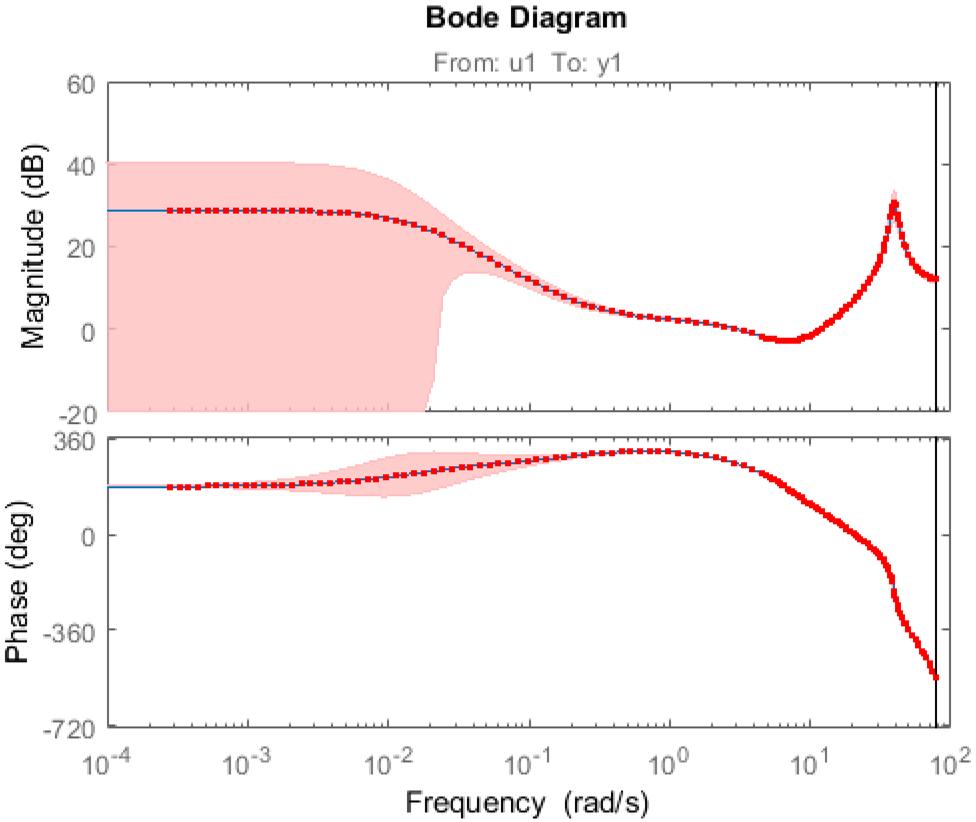
```
CanonicalModel = canon(FreeModel,'companion');
```

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,'r.');
showConfidence(h)
```



The frequency response confidence bounds are identical.

More About

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

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

the corresponding companion A matrix is

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

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.

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`

Introduced before R2006a

chgFreqUnit

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.

Input Arguments

`sys`

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

`newfrequnits`

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

- `'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

Change Frequency Units of Frequency-Response Data Model

Create a frequency-response data model.

```
load(fullfile(matlabroot,'examples','controls_id','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.

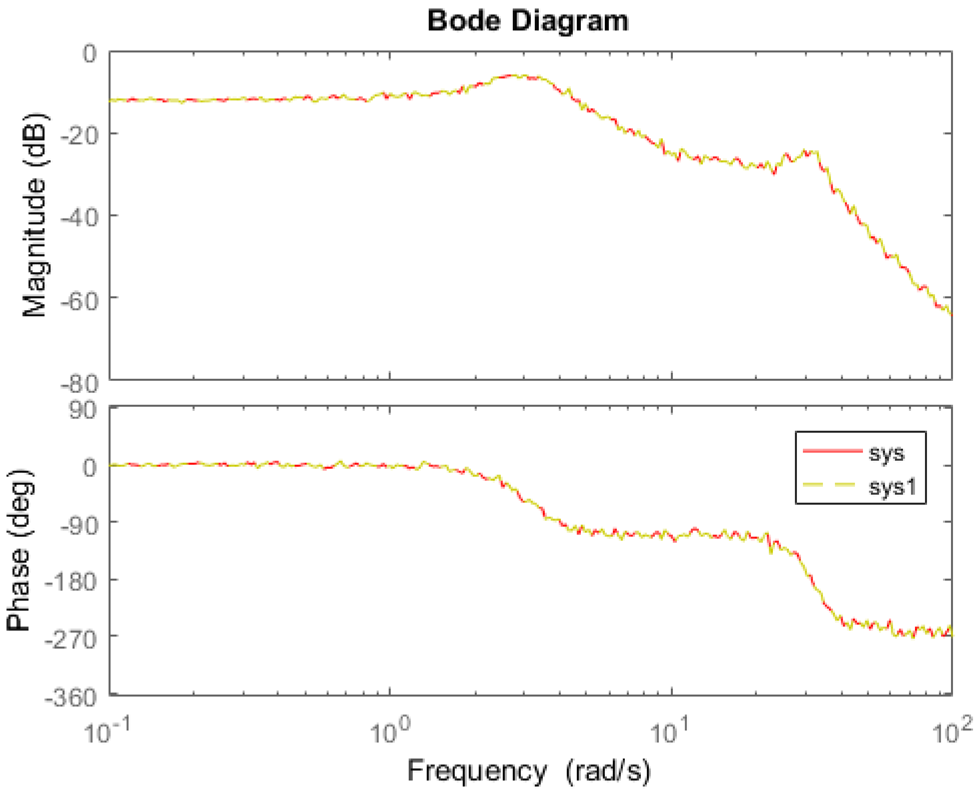
Change the frequency units.

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

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

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

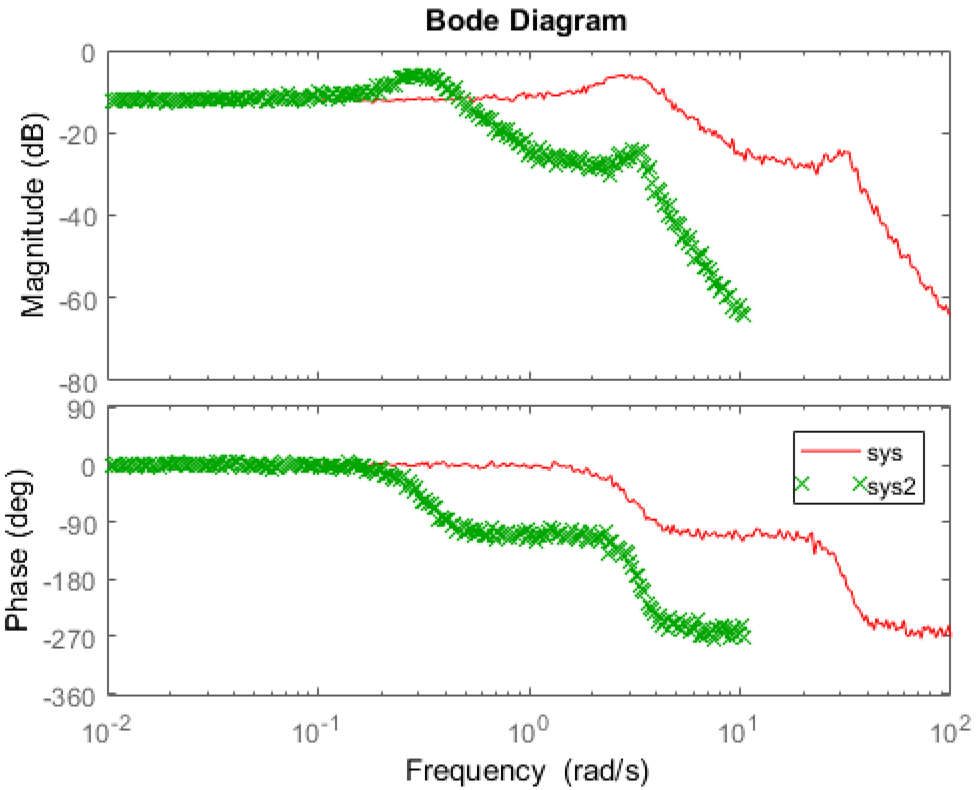
```
bodeplot(sys,'r',sys1,'y--');
legend('sys','sys1')
```



The magnitude and phase of `sys` and `sys1` match because `chgFreqUnit` command changes the units of frequency points in `sys` without modifying system behavior.

Change the `FrequencyUnit` property of `sys` to compare the Bode response with the original system.

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



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

- "Specify Frequency Units of Frequency-Response Data Model"

More About

Tips

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

See Also

`idfrd` | `chgTimeUnit` | `frd`

Introduced in R2012a

chgTimeUnit

Change time units of dynamic system

Syntax

```
sys_new = chgTimeUnit(sys,newtimeunits)
```

Description

`sys_new = chgTimeUnit(sys,newtimeunits)` changes the time units of `sys` to `newtimeunits`. The time- and frequency-domain characteristics of `sys` and `sys_new` match.

Input Arguments

`sys`

Dynamic system model

`newtimeunits`

New time units, specified as one of the following values:

- `'nanoseconds'`
- `'microseconds'`
- `'milliseconds'`
- `'seconds'`
- `'minutes'`
- `'hours'`
- `'days'`
- `'weeks'`

- `'months'`
- `'years'`

Default: `'seconds'`

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

Change Time Units of Dynamic System Model

Create a transfer function model.

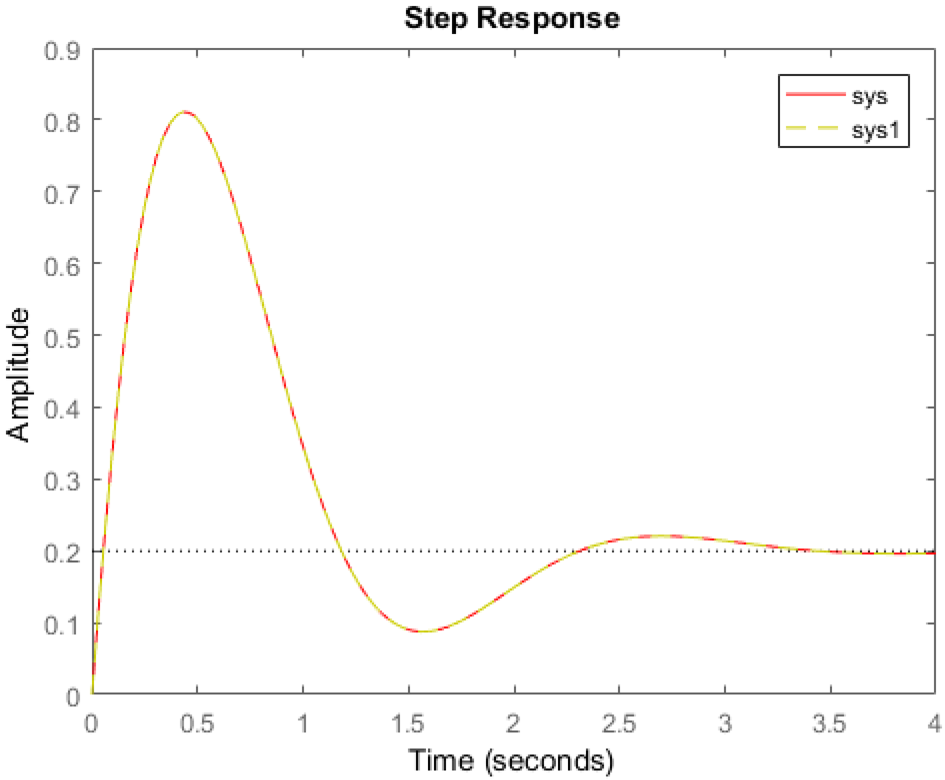
```
num = [4 2];
den = [1 3 10];
sys = tf(num,den);
```

By default, the time unit of `sys` is `'seconds'`. Create a new model with the time units changed to minutes.

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

This command sets the `TimeUnit` property of `sys1` to `'minutes'`, without changing the dynamics. To confirm that the dynamics are unchanged, compare the step responses of `sys` and `sys1`.

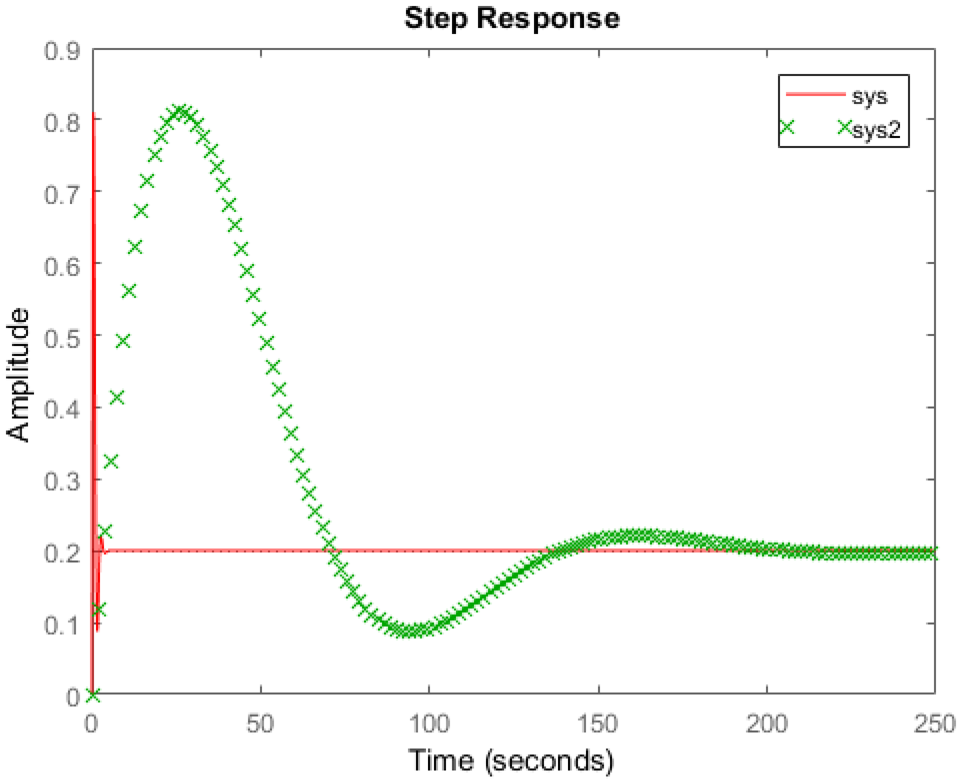
```
stepplot(sys,'r',sys1,'y--');
legend('sys','sys1');
```



The step responses are the same.

If you change the `TimeUnit` property of the system instead of using `chgTimeUnit`, the dynamics of the system do change. To see this, change the `TimeUnit` property of a copy of `sys` and compare the step response with the original system.

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



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.

- “Specify Model Time Units”

More About

Tips

- Use `chgTimeUnit` to change the time units without modifying system behavior.

See Also

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

Introduced in R2012a

clone

Copy online parameter estimation System object

Syntax

```
obj_clone = clone(obj)
```

Description

`obj_clone = clone(obj)` creates a copy of the online parameter estimation System object™, `obj`, with the same property values. If the object you clone is locked, the new object is also locked.

`clone` is not supported for code generation using MATLAB Coder™.

Note: If you want to copy an existing System object and then modify properties of the copied object, use the `clone` command. Do not create additional objects using syntax `obj2 = obj`. Any changes made to the properties of the new System object created this way (`obj2`) also change the properties of the original System object (`obj`).

Examples

Clone an Online Estimation System Object

Create a System object™ for online estimation of an ARX model with default properties.

```
obj = recursiveARX
```

```
obj = 

  recursiveARX with properties:

                            A: []
                            B: []
                     InitialA: [1 2.2204e-16]
```

```
                  InitialB: [0 2.2204e-16]
       ParameterCovariance: []
InitialParameterCovariance: [2×2 double]
          EstimationMethod: 'ForgettingFactor'
          ForgettingFactor: 1
          EnableAdaptation: true
                  DataType: 'double'
```

Use `clone` to generate an object with the same properties as the original object.

```
obj2 = clone(obj)
```

```
obj2 = 

  recursiveARX with properties:

                         A: []
                         B: []
                  InitialA: [1 2.2204e-16]
                  InitialB: [0 2.2204e-16]
       ParameterCovariance: []
InitialParameterCovariance: [2×2 double]
          EstimationMethod: 'ForgettingFactor'
          ForgettingFactor: 1
          EnableAdaptation: true
                  DataType: 'double'
```

Input Arguments

`obj` — System object for online parameter estimation

`recursiveAR` object | `recursiveARMA` object | `recursiveARX` object | `recursiveARMAX` object | `recursiveOE` object | `recursiveBJ` object | `recursiveLS` object

System object for online parameter estimation, created using one of the following commands:

- `recursiveAR`
- `recursiveARMA`

- `recursiveARX`
- `recursiveARMAX`
- `recursiveOE`
- `recursiveBJ`
- `recursiveLS`

Output Arguments

`obj_clone` — Copy of online estimation System object

System object

Copy of online estimation System object, `obj`, returned as a System object with the same properties as `obj`.

More About

- “What Is Online Estimation?”

See Also

`isLocked` | `recursiveAR` | `recursiveARMA` | `recursiveARMAX` | `recursiveARX` | `recursiveBJ` | `recursiveLS` | `recursiveOE` | `release` | `reset` | `step`

Introduced in R2015b

clone

Copy online state estimation object

Syntax

```
obj_clone = clone(obj)
```

Description

`obj_clone = clone(obj)` creates a copy of the online state estimation object `obj` with the same property values.

If you want to copy an existing object and then modify properties of the copied object, use the `clone` command. Do not create additional objects using syntax `obj2 = obj`. Any changes made to the properties of the new object created in this way (`obj2`) also change the properties of the original object (`obj`).

Examples

Clone an Online State Estimation Object

Create an extended Kalman filter object for a van der Pol oscillator with two states and one output. To create the object, use the previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`. Specify the initial state values for the two states as [2;0].

```
obj = extendedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,[2;0])
```

```
obj = 

  extendedKalmanFilter with properties:

        HasAdditiveProcessNoise: 1
             StateTransitionFcn: @vdpStateFcn
```

```
    HasAdditiveMeasurementNoise: 1
                MeasurementFcn: @vdpMeasurementFcn
     StateTransitionJacobianFcn: []
         MeasurementJacobianFcn: []
                          State: [2×1 double]
                StateCovariance: [2×2 double]
                   ProcessNoise: [2×2 double]
               MeasurementNoise: 1
```

Use `clone` to generate an object with the same properties as the original object.

```
obj2 = clone(obj)
```

```
obj2 = 

  extendedKalmanFilter with properties:

         HasAdditiveProcessNoise: 1
              StateTransitionFcn: @vdpStateFcn
    HasAdditiveMeasurementNoise: 1
                MeasurementFcn: @vdpMeasurementFcn
     StateTransitionJacobianFcn: []
         MeasurementJacobianFcn: []
                          State: [2×1 double]
                StateCovariance: [2×2 double]
                   ProcessNoise: [2×2 double]
               MeasurementNoise: 1
```

Modify the `MeasurementNoise` property of `obj2`.

```
obj2.MeasurementNoise = 2;
```

Verify that `MeasurementNoise` property of original object `obj` remains unchanged and equals 1.

```
obj.MeasurementNoise
```

```
ans =

     1
```

Input Arguments

`obj` — Object for online state estimation
`extendedKalmanFilter` object | `unscentedKalmanFilter` object

Object for online state estimation of a nonlinear system, created using one of the following commands:

- `extendedKalmanFilter`
- `unscentedKalmanFilter`

Output Arguments

`obj_clone` — Clone of online state estimation object
`extendedKalmanFilter` object | `unscentedKalmanFilter` object

Clone of online state estimation object `obj`, returned as an `extendedKalmanFilter` or `unscentedKalmanFilter` object with the same properties as `obj`.

More About

- “What Is Online Estimation?”

See Also

`correct` | `extendedKalmanFilter` | `predict` | `unscentedKalmanFilter`

Introduced in R2016b

compare

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

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.

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

To change display options, right-click the plot to access the context menu. For more details about the menu, see “Tips” on page 1-199.

`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`.

`compare(data,sys,style,prediction_horizon)` uses `style` to specify the line type, marker symbol, and color.

`compare(data,sys1,...,sysN,prediction_horizon)` compares multiple dynamic systems responses on the same axes. `compare` automatically chooses colors and line styles in the order specified by the `ColorOrder` and `LineStyleOrder` properties of the current axes.

`compare(data,sys1,style1,...,sysN,styleN,prediction_horizon)` compares multiple systems responses on the same axes using the line type, marker symbol, and color specified for each system.

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

`[y,fit,x0] = compare(___)` 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.

`sys`

`iddata` object or dynamic system model.

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 specified as one of the following:

- `Inf` — Compare simulated response of the system to data.

- Positive finite integer, `K`— Compare `K`-step ahead predicted response of the system to data.

`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 line and marker, specified as a character vector. For example, `'b'`, `'b+:'`.

For more information about configuring `style`, see “Specify Line Style, Color, and Markers” in the MATLAB documentation.

`opt`

Comparison option set.

`opt` is an option set created using `compareOptions`, which specifies options including:

- Handling of initial conditions
- Sample range for computing fit numbers
- Data offsets
- Output weighting

Output Arguments

`y`

Model response.

Measured output values in `data` up to time `t = t-prediction_horizon` are used to predict the output of `sys` at time *t*.

For multimodel comparisons, `y` is a cell array, with one entry for each input model.

For multiexperiment data, `y` is a cell array, with one entry for each experiment.

For multimodel comparisons using multiexperiment data, `y` is an *Nsys*-by-*Nexp* cell array. *Nsys* is the number of models, and *Nexp* 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`.

By default, the initial conditions required for computing the response are estimated to maximize the fit to data. Use the `compareOptions` option set to specify handling of initial conditions.

`fit`

NRMSE fitness value.

The fit is calculated (in percentage) using:

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

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 *Ny*-by-1 vector, where *Ny* is the number of outputs.

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

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

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

For multimodel comparisons using multiexperiment data, `fit` is an *Nsys*-by-*Nexp* cell array. *Nsys* is the number of models, and *Nexp* is the number of experiments.

x0

Initial conditions used to compute system response.

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

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

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

For multimodel comparisons using multiexperiment data, `x0` is an *Nsys*-by-*Nexp* cell array. *Nsys* is the number of models, and *Nexp* is the number of experiments.

Examples

Compare Estimated Model to Measured Data

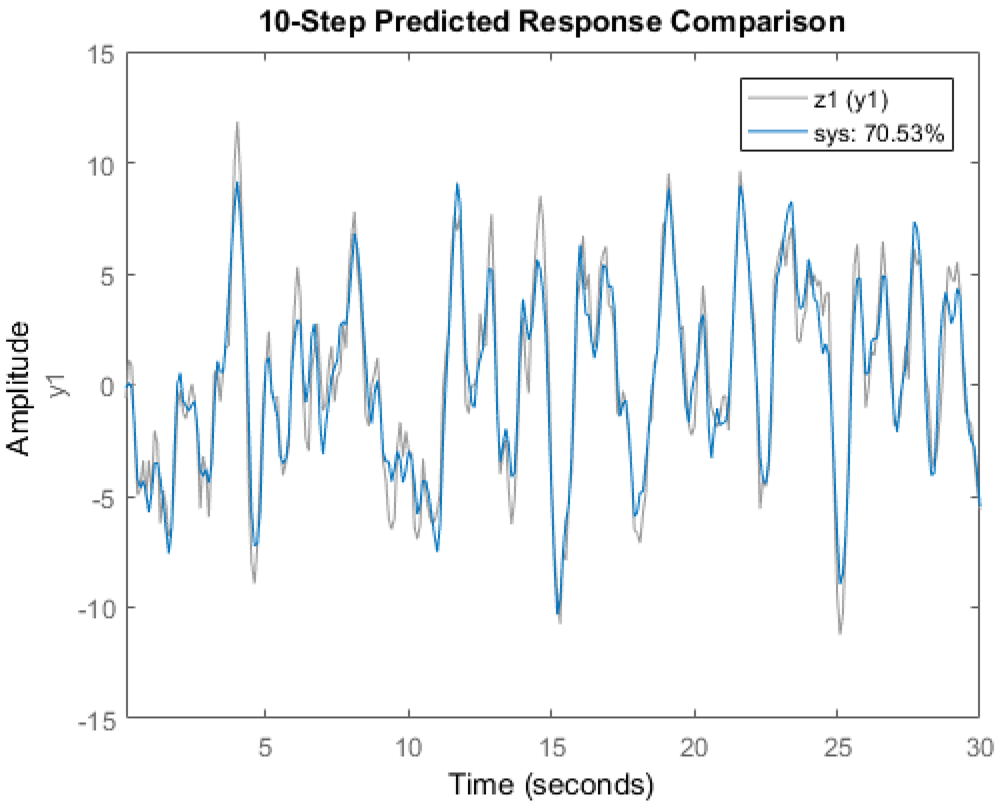
Estimate a state-space model for measured data.

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

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

Compare the predicted output for 10 steps ahead to the measured output.

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



To change display options in the plot, right-click the plot to access the context menu. For example, to plot the error between the predicted output and measured output, select **Error Plot** from the context menu.

Compare Multiple Estimated Models to Measured Data

Compare the outputs of multiple estimated models, of differing types, to measured data.

This example compares 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.

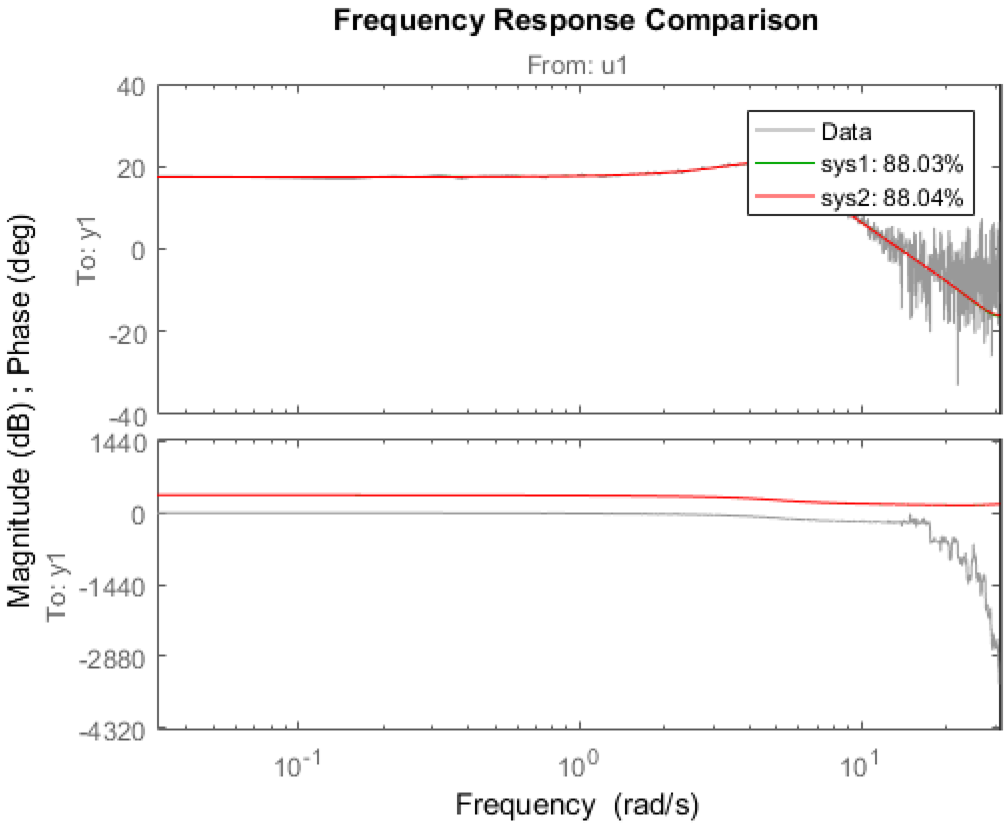
```
load demofr  % frequency response data
```

```
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. 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.

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

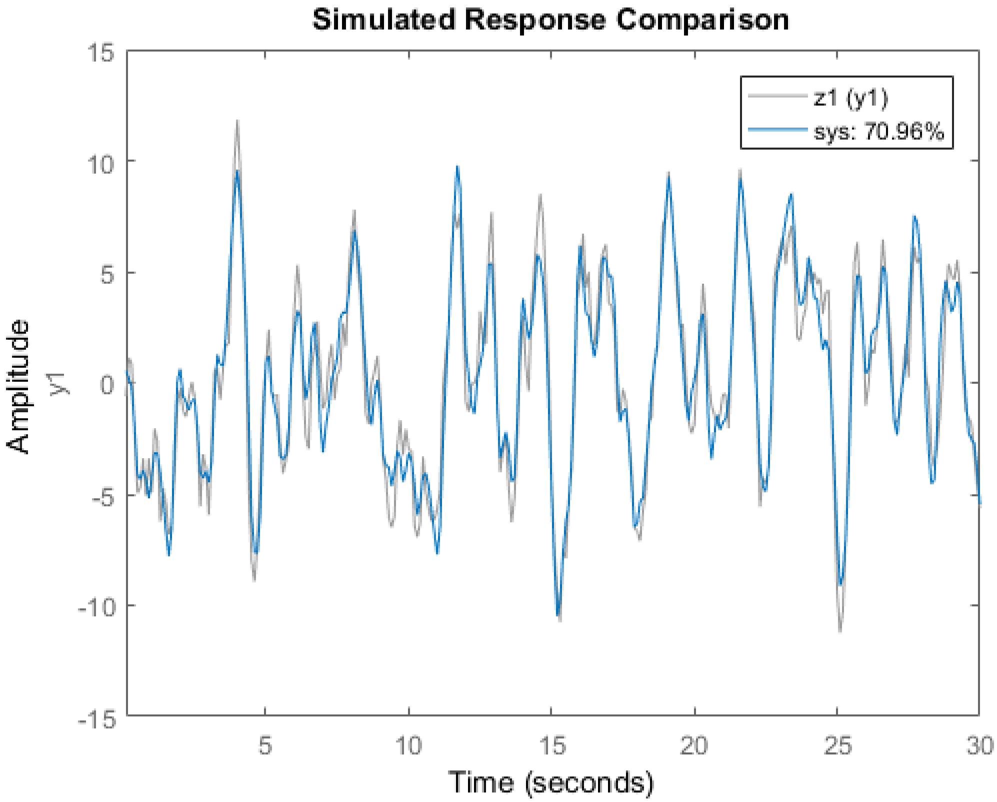
`sys`, an `idtf` model, is a continuous-time transfer function model.

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);
```



To change display options in the plot, right-click the plot to access the context menu. For example, to view the confidence region for the simulated response, select **ConfidenceRegion** from the context menu. To specify number of standard deviations to plot, double-click the plot and open the Property Editor dialog box. In the dialog box, in the **Options** tab, specify the number of standard deviations in **Confidence Region for Identified Models**. The default value is `1` standard deviation.

More About

Tips

- Right-clicking the plot opens the context menu, where you can access the following options:
  - **Systems** — Select systems to view simulated or predicted response. By default, the response of all systems is plotted.
  - **Data Experiment** — For multi-experiment data only. Toggle between data from different experiments.
  - **Characteristics** — View the following data characteristics:
    - **Peak Value** — View peak value of the data. Not applicable for frequency-response data.
    - **Peak Response** — View peak response of the data. Applicable for frequency-response data only.
    - **Mean Value** — View mean value of the data. Not applicable for frequency-domain or frequency-response data.
    - **Confidence Region** — View the confidence region for the simulated response. Applicable when the prediction horizon is `Inf`. To specify number of standard deviations for plotting the response confidence region, double-click the plot and open the Property Editor dialog box. Specify the number of standard deviations in the **Options** tab, in **Confidence Region for Identified Models**. The default value is `1` standard deviation.
  - **Show** — For frequency-domain and frequency-response data only.
    - **Magnitude** — View magnitude of frequency response of the system.
    - **Phase** — View phase of frequency response of the system.
  - **Show Validation Data** — Plot validation data. By default, the validation data is always plotted.
  - **I/O Grouping** — For datasets containing more than one input or output channel. Select grouping of input and output channels on the plot.

  - **None** — Plot input-output channels in their own separate axes.
  - **All** — Group all input channels together and all output channels together.
- **I/O Selector** — For datasets containing more than one input or output channel. Select a subset of the input and output channels to plot. By default, all output channels are plotted.
- **Grid** — Add grids to the plot.
- **Normalize** — Normalize the y-scale of all data in the plot.
- **Full View** — Return to full view. By default, the plot is scaled to full view.
- **Prediction Horizon** — For time-domain data with noise-component only. Set the prediction horizon, or choose simulation.
- **Initial Condition** — Specify handling of initial conditions. Not applicable for frequency-response data.

  Specify as one of the following:

  - **Estimate** — Treat the initial conditions as estimation parameters.
  - **Zero** — Set all initial conditions to zero.
  - **Absorb delays and estimate** — Absorb nonzero delays into the model coefficients and treat the initial conditions as estimation parameters. Use this option for discrete-time models only.
- **Model Response Plot** — Plot the simulated or predicted model response. Be default, the response plot is always shown.
- **Error Plot** — Plot the error between the model response and validation data.
- **Properties** — Open the Property Editor dialog box to customize plot attributes.

See Also

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

Introduced before R2006a

compareOptions

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. `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 *Ne* columns, where *Ne* 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 *Ne* elements, where *Ne* 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.

For an `idnlgrey` model, `sys`, `InitialCondition` can also be one of the following:

- `'fixed'` — `sys.InitialStates` determines the values of the initial states, but all the states are considered fixed for estimation.
- `'model'` — `sys.InitialStates` determines the values of the initial states, which states to estimate and their minimum/maximum values.

Default: `'e'`

`'InputOffset'`

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

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 for model response prediction.

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` requires one of the following values:

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

Introduced in R2012a

correct

Correct state and state estimation error covariance using extended or unscented Kalman filter and measurements

The `correct` command updates the state and state estimation error covariance of an `extendedKalmanFilter` or `unscentedKalmanFilter` object using measured system outputs. To implement extended or unscented Kalman filter algorithms, use the `correct` and `predict` commands together. If the current output measurement exists, you can use `correct` and `predict`. If the measurement is missing, you can only use `predict`. For information about the order in which to use the commands, see “Using predict and correct Commands” on page 1-210.

Syntax

```
[CorrectedState,CorrectedStateCovariance] = correct(obj,y)
[CorrectedState,CorrectedStateCovariance] = correct(obj,y,
Um1,...,Umn)
```

Description

`[CorrectedState,CorrectedStateCovariance] = correct(obj,y)` corrects the state estimate and state estimation error covariance of an extended or unscented Kalman filter object `obj` using the measured output `y`.

You create `obj` using the `extendedKalmanFilter` or `unscentedKalmanFilter` commands. You specify the state transition function and measurement function of your nonlinear system in `obj`. You also specify whether the process and measurement noise terms are additive or nonadditive in these functions. The `State` property of the object stores the latest estimated state value. Assume that at time step `k`, `obj.State` is $\hat{x}[k|k-1]$. This value is the state estimate for time `k`, estimated using measured outputs until time `k-1`. When you use the `correct` command with measured system output `y[k]`, the software returns the corrected state estimate $\hat{x}[k|k]$ in the `CorrectedState` output. Where $\hat{x}[k|k]$ is the state estimate at time `k`, estimated using measured outputs until time `k`. The command returns the state estimation error covariance of $\hat{x}[k|k]$ in the `CorrectedStateCovariance` output. The software also updates the `State` and `StateCovariance` properties of `obj` with these corrected values.

Use this syntax if the measurement function *h* that you specified in `obj.MeasurementFcn` has one of the following forms:

- `y(k) = h(x(k))` — for additive measurement noise.
- `y(k) = h(x(k),v(k))` — for nonadditive measurement noise.

Where `y(k)`, `x(k)`, and `v(k)` are the measured output, states, and measurement noise of the system at time step `k`. The only inputs to *h* are the states and measurement noise.

`[CorrectedState,CorrectedStateCovariance] = correct(obj,y,Um1,...,Umn)` specifies additional input arguments, if the measurement function of the system requires these inputs. You can specify multiple arguments.

Use this syntax if the measurement function *h* has one of the following forms:

- `y(k) = h(x(k),Um1,...,Umn)` — for additive measurement noise.
- `y(k) = h(x(k),v(k),Um1,...,Umn)` — for nonadditive measurement noise.

`correct` command passes these inputs to the measurement function to calculate the estimated outputs.

Examples

Estimate States Online Using Extended Kalman Filter

Estimate the states of a van der Pol oscillator using an extended Kalman filter algorithm and measured output data. The oscillator has two states and one output.

Create an extended Kalman filter object for the oscillator. Use previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`. These functions describe a discrete-approximation to a van der Pol oscillator with nonlinearity parameter, mu, equal to 1. The functions assume additive process and measurement noise in the system. Specify the initial state values for the two states as [1;0]. This is the guess for the state value at initial time `k`, using knowledge of system outputs until time `k-1`, $\hat{x}[k|k-1]$.

```
obj = extendedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,[1;0]);
```

Load the measured output data, `y`, from the oscillator. In this example, use simulated static data for illustration. The data is stored in the `vdp_data.mat` file.

```
load vdp_data.mat y
```

Specify the process noise and measurement noise covariances of the oscillator.

```
obj.ProcessNoise = 0.01;
obj.MeasurementNoise = 0.16;
```

Implement the extended Kalman filter algorithm to estimate the states of the oscillator by using the `correct` and `predict` commands. You first correct $\hat{x}[k|k-1]$ using measurements at time `k` to get $\hat{x}[k|k]$. Then, you predict the state value at next time step, $\hat{x}[k+1|k]$, using $\hat{x}[k|k]$, the state estimate at time step `k` that is estimated using measurements until time `k`.

To simulate real-time data measurements, use the measured data one time step at a time.

```
for k = 1:size(y)
    [CorrectedState,CorrectedStateCovariance] = correct(obj,y(k));
    [PredictedState,PredictedStateCovariance] = predict(obj);
end
```

When you use the `correct` command, `obj.State` and `obj.StateCovariance` are updated with the corrected state and state estimation error covariance values for time step `k`, `CorrectedState` and `CorrectedStateCovariance`. When you use the `predict` command, `obj.State` and `obj.StateCovariance` are updated with the predicted values for time step `k+1`, `PredictedState` and `PredictedStateCovariance`.

In this example, you used `correct` before `predict` because the initial state value was $\hat{x}[k|k-1]$, a guess for the state value at initial time `k` using system outputs until time `k-1`. If your initial state value is $\hat{x}[k-1|k-1]$, the value at previous time `k-1` using measurement until `k-1`, then use the `predict` command first. For more information about the order of using `predict` and `correct`, see “Using predict and correct Commands”.

Specify State Transition and Measurement Functions with Additional Inputs

Consider a nonlinear system with input `u` whose state `x` and measurement `y` evolve according to the folowing state transition and measurement equations:

$$x[k] = \sqrt{x[k-1] + u[k-1]} + w[k-1]$$

$$y[k] = x[k] + 2 * u[k] + v[k]^2$$

The process noise `w` of the system is additive while the measurement noise `v` is nonadditive.

Create the state transition function and measurement function for the system. Specify the functions with an additional input `u`.

```
f = @(x,u)(sqrt(x+u));
h = @(x,v,u)(x+2*u+v^2);
```

`f` and `h` are function handles to the anonymous functions that store the state transition and measurement functions, respectively. In the measurement function, because the measurement noise is nonadditive, `v` is also specified as an input. Note that `v` is specified as an input before the additional input `u`.

Create an extended Kalman filter object for estimating the state of the nonlinear system using the specified functions. Specify the initial value of the state as 1, and the measurement noise as nonadditive.

```
obj = extendedKalmanFilter(f,h,1,'HasAdditiveMeasurementNoise',false);
```

Specify the measurement noise covariance.

```
obj.MeasurementNoise = 0.01;
```

You can now estimate the state of the system using the `predict` and `correct` commands. You pass the values of `u` to `predict` and `correct`, which in turn pass them to the state transition and measurement functions, respectively.

Correct the state estimate with measurement y[k]=0.8 and input u[k]=0.2 at time step k.

```
correct(obj,0.8,0.2)
```

Predict the state at next time step, given u[k]=0.2.

```
predict(obj,0.2)
```

- "Nonlinear State Estimation Using Unscented Kalman Filter"
- "Fault Detection Using an Extended Kalman Filter"
- "Generate Code for Online State Estimation in MATLAB"

Input Arguments

`obj` — Extended or unscented Kalman filter object

`extendedKalmanFilter` object | `unscentedKalmanFilter` object

Extended or unscented Kalman filter object for online state estimation, created using one of the following commands:

- `extendedKalmanFilter` — Uses the extended Kalman filter algorithm.
- `unscentedKalmanFilter` — Uses the unscented Kalman filter algorithm.

`y` — Measured system output

vector

Measured system output at the current time step, specified as an *N*-element vector, where *N* is the number of measurements.

`Um1,...,Umn` — Additional input arguments to measurement function

input arguments of any type

Additional input arguments to the measurement function of the system, specified as input arguments of any type. The measurement function, *h*, is specified in the `MeasurementFcn` property of `obj`. If the function requires input arguments in addition to the state and measurement noise values, you specify these inputs in the `correct` command syntax. `correct` command passes these inputs to the measurement function to calculate estimated outputs. You can specify multiple arguments.

For example, suppose that your measurement function calculates the estimated system output `y` using system inputs `u` and current time `k`, in addition to the state `x`:

```
y(k) = h(x(k),u(k),k)
```

Then when you perform online state estimation at time step `k`, specify these additional inputs in the `correct` command syntax:

```
[CorrectedState,CorrectedStateCovariance] = correct(obj,y,u(k),k);
```

Output Arguments

`CorrectedState` — Corrected state estimate

vector

Corrected state estimate, returned as a vector of size M, where M is the number of states of the system. If you specify the initial states of `obj` as a column vector then M is returned as a column vector, otherwise M is returned as a row vector.

For information about how to specify the initial states of the object, see the `extendedKalmanFilter` and `unscentedKalmanFilter` reference pages.

`CorrectedStateCovariance` — Corrected state estimation error covariance
matrix

Corrected state estimation error covariance, returned as an M-by-M matrix, where M is the number of states of the system.

More About

Using `predict` and `correct` Commands

After you have created an extended or unscented Kalman filter object, `obj`, to implement the extended or unscented Kalman filter algorithms, use the `correct` and `predict` commands together.

At time step `k`, `correct` command returns the corrected value of states and state estimation error covariance using measured system outputs `y[k]` at the same time step. If your measurement function has additional input arguments U_m, you specify these as inputs to the `correct` command. The command passes these values to the measurement function.

```
[CorrectedState,CorrectedCovariance] = correct(obj,y,Um)
```

The `correct` command updates the `State` and `StateCovariance` properties of the object with the estimated values, `CorrectedState` and `CorrectedCovariance`.

The `predict` command returns the prediction of state and state estimation error covariance at the next time step. If your state transition function has additional input arguments U_s, you specify these as inputs to the `predict` command. The command passes these values to the state transition function.

```
[PredictedState,PredictedCovariance] = predict(obj,Us)
```

The `predict` command updates the `State` and `StateCovariance` properties of the object with the predicted values, `PredictedState` and `PredictedCovariance`.

If the current output measurement exists at a given time step, you can use `correct` and `predict`. If the measurement is missing, you can only use `predict`. For details about how these commands implement the algorithms, see “Extended and Unscented Kalman Filter Algorithms for Online State Estimation”.

The order in which you implement the commands depends on the availability of measured data `y`, U_s, and U_m for your system:

- `correct` then `predict` — Assume that at time step `k`, the value of `obj.State` is $\hat{x}[k|k-1]$. This value is the state of the system at time `k`, estimated using measured outputs until time `k-1`. You also have the measured output `y[k]` and inputs U_s`[k]` and U_m`[k]` at the same time step.

  Then you first execute the `correct` command with measured system data `y[k]` and additional inputs U_m`[k]`. The command updates the value of `obj.State` to be $\hat{x}[k|k]$, the state estimate for time `k`, estimated using measured outputs up to time `k`. When you then execute the `predict` command with input U_s`[k]`, `obj.State` now stores $\hat{x}[k+1|k]$. The algorithm uses this state value as an input to the `correct` command in the next time step.
- `predict` then `correct` — Assume that at time step `k`, the value of `obj.State` is $\hat{x}[k-1|k-1]$. You also have the measured output `y[k]` and input U_m`[k]` at the same time step but you have U_s`[k-1]` from the previous time step.

  Then you first execute the `predict` command with input U_s`[k-1]`. The command updates the value of `obj.State` to $\hat{x}[k|k-1]$. When you then execute the `correct` command with input arguments `y[k]` and U_m`[k]`, `obj.State` is updated with $\hat{x}[k|k]$. The algorithm uses this state value as an input to the `predict` command in the next time step.

Thus, while in both cases the state estimate for time `k`, $\hat{x}[k|k]$ is the same, if at time `k` you do not have access to the current state transition function inputs U_s`[k]`, and instead have U_s`[k-1]`, then use `predict` first and then `correct`.

For an example of estimating states using the `predict` and `correct` commands, see “Estimate States Online Using Extended Kalman Filter” on page 1-206.

- “What Is Online Estimation?”

- "Extended and Unscented Kalman Filter Algorithms for Online State Estimation"

See Also

`clone` | `extendedKalmanFilter` | `predict` | `unscentedKalmanFilter`

Introduced in R2016b

cra

Estimate impulse response using prewhitened-based correlation analysis

Syntax

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

Description

`ir=cra(data)` estimates the impulse response for the time-domain data, `data`.

`[ir,R,cl] = cra(data,M,na,plot)` estimates correlation/covariance information, `R`, and the 99% confidence level for the impulse response, `cl`.

`cra` prewhitens the input sequence; that is, `cra` filters `u` through a filter chosen so that the result is as uncorrelated (white) as possible. The output `y` is subjected to the same filter, and then the covariance functions of the filtered `y` and `u` are computed and graphed. The cross correlation function between (prewhitened) input and output is also computed and graphed. Positive values of the lag variable then correspond to an influence from `u` to later values of `y`. In other words, significant correlation for negative lags is an indication of feedback from `y` to `u` in the data.

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

Input Arguments

`data`

Input-output data.

Specify `data` as an `iddata` object containing time-domain data only.

`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`

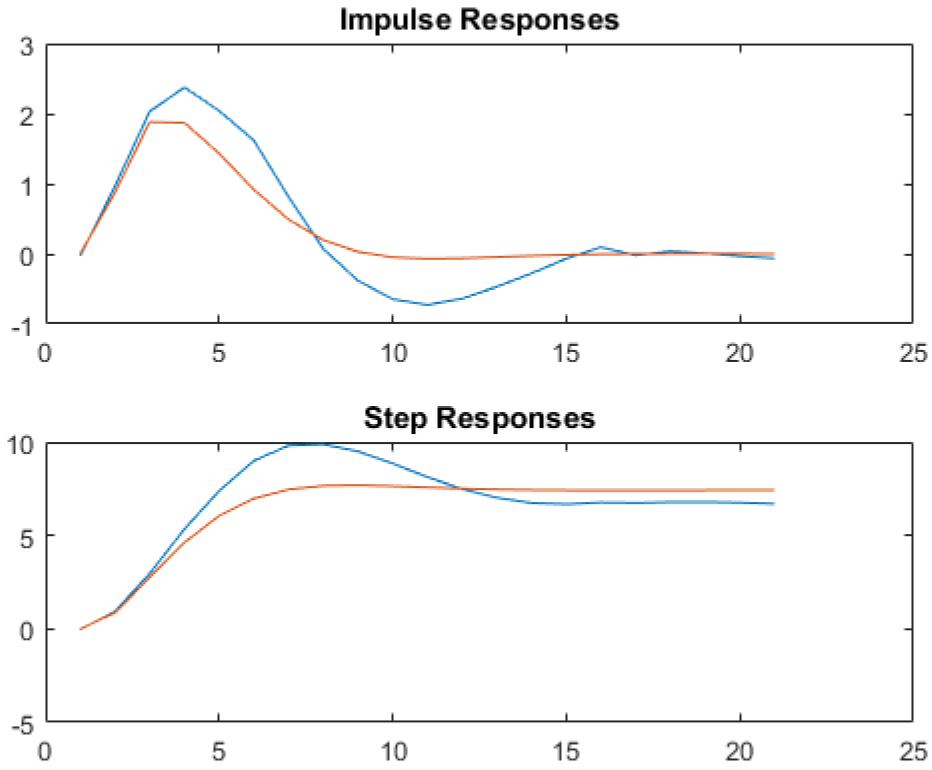
99 % significance level for the impulse response.

Examples

Estimate the Impulse Response of an ARX Model

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')
```



Impulse Responses

Step Responses

Alternatives

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

See Also

`impulse` | `impulseest` | `spa` | `step`

Introduced before R2006a

customnet

Custom nonlinearity estimator for nonlinear ARX and Hammerstein-Wiener models

Syntax

```
C=customnet(H)
C=customnet(H,PropertyName,PropertyValue)
```

Description

`customnet` 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.

Construction

`C=customnet(H)` creates a nonlinearity estimator object with a user-defined unit function using the function handle `H`. `H` must point to a function of the form `[f,g,a] = function_name(x)`, where `f` is the value of the function, `g = df/dx`, and `a` indicates the unit function active range. `g` is significantly nonzero in the interval `[-a a]`. Hammerstein-Wiener models require that your custom nonlinearity have only one input and one output.

`C=customnet(H,PropertyName,PropertyValue)` creates a nonlinearity estimator using property-value pairs defined in "customnet Properties" on page 1-217.

customnet Properties

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

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

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

```
C.NumberOfUnits
```

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

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

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

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

Examples

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

Use custom networks in `nlarx` and `nlhw` 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));
```

More About

Tips

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

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

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.

Algorithms

`customnet` uses an iterative search technique for estimating parameters.

- "Identifying Nonlinear ARX Models"
- "Identifying Hammerstein-Wiener Models"

See Also

`evaluate` | `nlarx` | `nlhw`

Introduced in R2007a

customreg

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))*, $u(t\text{-}1)^2$, and *u(t-1)*y(t-3)*.

For simpler regressor expressions, specify custom regressors directly in the app 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 `(xi,yi)` 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 function of input and output variables. *`Variables`* represent the names of model inputs and outputs in the function *`Function`*. Each input and output name must coincide with 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 character vector representing a function of standards regressors. For example: `cr = @(x,y) x*y`
`Variables`	Cell array of character vectors that represent the names of model input and output variables in the function `Function`. Each input and output name must coincide with the `InputName` and `OutputName` properties of the `idnlarx` object—the model for which you define custom regressors. The size of `Variables` must match the number of `Function` inputs. For example, `Variables` correspond to `{'y1','u1'}` in: `C = customreg(cr,{'y1','u1'},[2 3])`
`Delays`	Vector of positive integers representing the delays of `Variables`. The size of `Delays` must match the size of `Arguments`. Default: `1` for each vector element. For example, `Delays` are `[2 3]` in: `C = customreg(cr,{'y1','u1'},[2 3])`
`Vectorized`	Assignable values:

Property Name	Description
	- `0` (default)—`Function` is not computed in vectorized form. - `1`—`Function` is computed in vectorized form when called with vector arguments.

Examples

Define Custom Regressors

Load estimation data.

```
load iddata1
```

Specify the regressors as a cell array of character vectors.

```
C = {'u1(t-1)*sin(y1(t-3))','u1(t-2)^3'};
```

u1 and y1 are input and output data, respectively.

Estimate a nonlinear ARX model using the custom regressors.

```
m = nlarx(z1,[2 2 1],'linear','CustomRegressors',C);
```

Define Custom Regressors During Estimation

Load the estimation data.

```
load iddata1
```

Estimate a nonlinear ARX model with custom regressors.

```
m = nlarx(z1,[2 2 1],'linear','CustomRegressors',...
          {'u1(t-1)*sin(y1(t-3))','u1(t-2)^3'});
```

Define Custom Regressors as Array of `customreg` Objects

Load the estimation data.

```
load iddata1
```

Construct a nonlinear ARX model.

```
m = idnlarx([2 2 1]);
```

Define the custom regressors.

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

Add custom regressors to the model.

```
m2 = addreg(m,C);
```

Use Vectorization Rules to Evaluate Regressor Expression

Load the estimation data.

```
load iddata1
```

Specify the regressors.

```
C = customreg(@(x,y) x*sin(y),{'u' 'y'},[1 3]);
set(C,'Vectorized',1);
```

Estimate a nonlinear ARX model with custom regressors.

```
m = nlarx(z1,[2 2 1],'sigmoidnet','CustomReg',C);
```

More About

- "Identifying Nonlinear ARX Models"

See Also

`addreg` | `getreg` | `idnlarx` | `nlarx` | `polyreg`

Introduced in R2007a

d2c

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`.

Input Arguments

`sysd`

Discrete-time dynamic system model

You cannot directly use an `idgrey` model whose `FunctionType` is `'d'` with `d2c`. Convert the model into `idss` form first.

Default:

`method`

Discrete-to-continuous time conversion method, specified as one of the following values:

- `'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 λ in `sysd`, then the continuous-time model `sysc` has an indicated level of noise spectral density equal to $Ts*\lambda$.
- 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 `xd[k]` of the state-space model `sysd` to the states `xc(t)` of `sysc`:

$$x_c(kT_s) = G\begin{bmatrix} x_d[k] \\ u[k] \end{bmatrix}.$$

Given an initial condition `x0` for `sysd` and an initial input `u0 = 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

Convert Discrete-Time Transfer Function to Continuous Time

Create the following discrete-time transfer function:

$$H(z) = \frac{z-1}{z^2+z+0.3}$$

```
H = tf([1 -1],[1 1 0.3],0.1);
```

The sample time of the model is $T_s = 0.1s$.

Derive a continuous-time, zero-order-hold equivalent model.

```
Hc = d2c(H)
```

```
Hc =

   121.7 s + 1.405e-12
  ---------------------
  s^2 + 12.04 s + 776.7

Continuous-time transfer function.
```

Discretize the resulting model, `Hc`, with the default zero-order hold method and sample time 0.1s to return the original discrete model, `H`.

```
c2d(Hc,0.1)
```

```
ans =
```

```
      z - 1
  -------------
  z^2 + z + 0.3

Sample time: 0.1 seconds
Discrete-time transfer function.
```

Use the Tustin approximation method to convert `H` to a continuous time model.

```
Hc2 = d2c(H,'tustin');
```

Discretize the resulting model, Hc2, to get back the original discrete-time model, `H`.

```
c2d(Hc2,0.1,'tustin');
```

Convert Identified Discrete-Time Transfer Function to Continuous Time

Estimate a discrete-time transfer function model, and convert it to a continuous-time model.

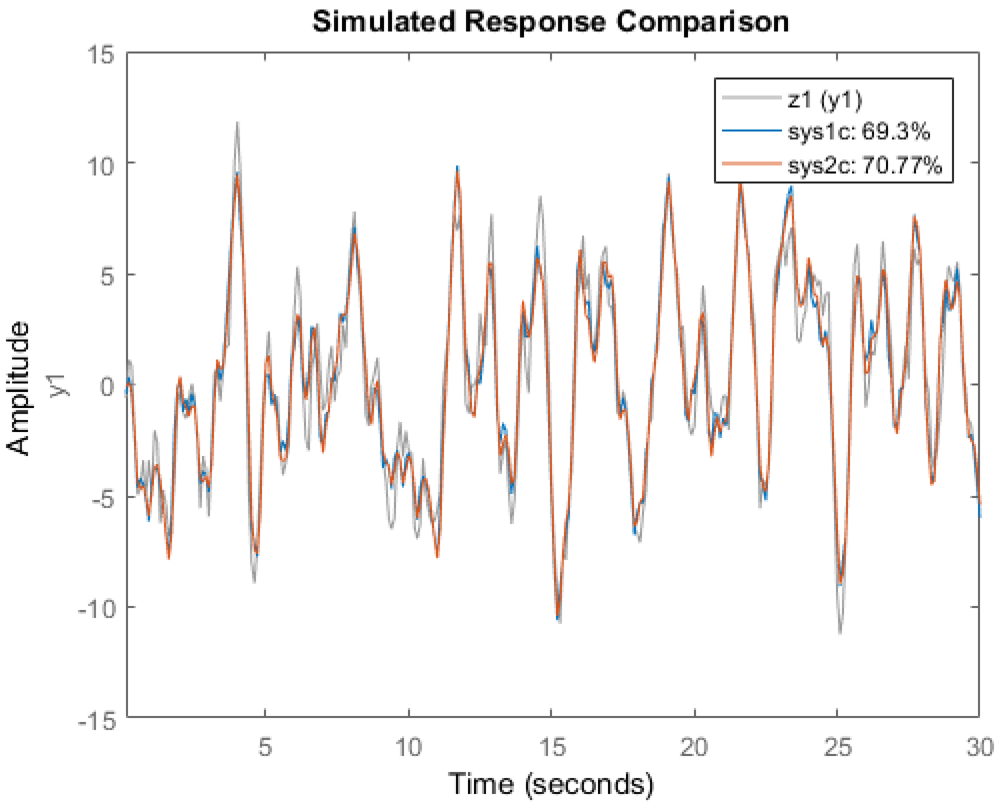
```
load iddata1
sys1d = tfest(z1,2,'Ts',0.1);
sys1c = d2c(sys1d,'zoh');
```

Estimate a continuous-time transfer function model.

```
sys2c = tfest(z1,2);
```

Compare the response of `sys1c` and the directly estimated continuous-time model, `sys2c`.

```
compare(z1,sys1c,sys2c)
```



The two systems are almost identical.

Regenerate Covariance Information After Converting to Continuous-Time Model

Convert an identified discrete-time transfer function model to continuous-time.

```
load iddata1
sysd = tfest(z1,2,'Ts',0.1);
sysc = d2c(sysd,'zoh');
```

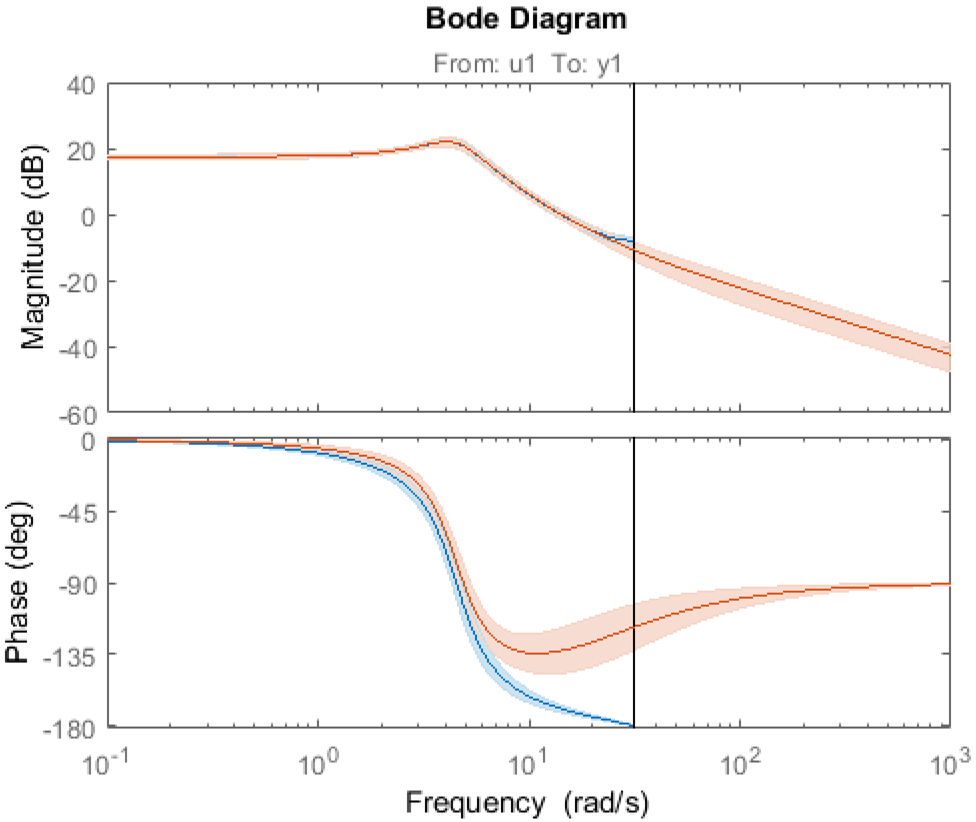
`sys1c` has no covariance information. The `d2c` operation leads to loss of covariance data of identified models.

Regenerate the covariance information using a zero iteration update with the same estimation command and estimation data.

```
opt = tfestOptions;
opt.SearchOption.MaxIter = 0;
sys1c = tfest(z1,sysc,opt);
```

Analyze the effect on frequency-response uncertainty.

```
h = bodeplot(sysd,sys1c);
showConfidence(h,3)
```



The uncertainties of `sys1c` and `sysd` are comparable up to the Nyquist frequency. However, `sys1c` 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 the `translatecov` command which is a Gauss-approximation formula based translation of covariance across model type conversion operations.

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

$$H(z) = \frac{z + 0.2}{(z + 0.5)\left(z^2 + z + 0.4\right)}$$

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)

Sample time: 0.1
```

This discrete model coincides with $H(z)$ after canceling the pole/zero pair at $z = -0.5$.

More About

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.

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.

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`

Introduced before R2006a

d2cOptions

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 `d2c` assumes the control inputs are piecewise constant over the sample time `Ts`.
`'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, `d2c` converts with no prewarp. To include prewarp, use the `PrewarpFrequency` 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 sample 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`

Introduced in R2012a

d2d

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`.

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 = zpk(0.7,0.5,1,0.1)
```

```
H2 = d2d(H,0.05)
Zero/pole/gain:
(z-0.8243)
----------
(z-0.7071)

Sample 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)

Sample time: 0.1
```

Example 2

Suppose you estimates a discrete-time model of a sample time commensurate with the estimation data (`Ts = 0.1 seconds`). However, your deployment application demands a faster sampling frequency (`Ts = 0.01 seconds`).

```
load iddata1
sys = oe(z1, [2 2 1]);
sysFast = d2d(sys, 0.01, 'zoh')

bode(sys, sysFast)
```

More About

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`.

See Also

`d2dOptions` | `c2d` | `d2c` | `upsample` | `translatecov`

Introduced before R2006a

d2dOptions

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 `d2d` assumes the control inputs are piecewise constant over the sample time `Ts`.
`'tustin'`	Bilinear (Tustin) approximation. By default, `d2d` resamples with no prewarp. To include prewarp, use the `PrewarpFrequency` 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

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 sample time
```

To resample to a different sample 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`

Introduced in R2012a

damp

Natural frequency and damping ratio

Syntax

```
damp(sys)
[Wn,zeta] = damp(sys)
[Wn,zeta,P] = damp(sys)
```

Description

`damp(sys)` displays a table of the damping ratio (also called *damping factor*), natural frequency, and time constant of the poles of the linear model `sys`. For a discrete-time model, the table also includes the magnitude of each pole. Frequencies are expressed in units of the reciprocal of the `TimeUnit` property of `sys`. Time constants are expressed in the same units as the `TimeUnit` property of `sys`.

This command requires a Control System Toolbox license.

`[Wn,zeta] = damp(sys)` returns the natural frequencies, `Wn`, and damping ratios,`zeta`, of the poles of `sys`.

`[Wn,zeta,P] = damp(sys)` returns the poles of `sys`.

Input Arguments

`sys`

Any linear dynamic system model.

Output Arguments

`Wn`

Vector containing the natural frequencies of each pole of `sys`, in order of increasing frequency. Frequencies are expressed in units of the reciprocal of the `TimeUnit` property of `sys`.

If `sys` is a discrete-time model with specified sample time, `Wn` contains the natural frequencies of the equivalent continuous-time poles (see “Algorithms” on page 1-246). If `sys` has an unspecified sample time `(Ts = -1)`, then the software uses `Ts = 1` and calculates `Wn` accordingly.

`zeta`

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 sample time, `zeta` contains the damping ratios of the equivalent continuous-time poles (see “Algorithms” on page 1-246). If `sys` has an unspecified sample time `(Ts = -1)`, then the software uses `Ts = 1` and calculates `zeta` accordingly.

`P`

Vector containing the poles of `sys`, in order of increasing natural frequency. `P` is the same as the output of `pole(sys)`, except for the order.

Examples

Display Natural Frequency, Damping Ratio, and Poles of Continuous-Time System

Create the following continuous-time 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, time constants, and poles of *H*.

```
damp(H)
```

```
        Pole               Damping       Frequency      Time Constant
                                       (rad/seconds)      (seconds)

 -1.00e+00 + 1.41e+00i     5.77e-01       1.73e+00          1.00e+00
 -1.00e+00 - 1.41e+00i     5.77e-01       1.73e+00          1.00e+00
```

Obtain vectors containing the natural frequencies and damping ratios of the poles.

```
[Wn,zeta] = damp(H);
```

Calculate the associated time constants.

```
tau = 1./(zeta.*Wn);
```

Display Natural Frequency, Damping Ratio, and Poles of Discrete-Time System

Create 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)
```

```
        Pole              Magnitude     Damping        Frequency      Time Constant
                                                     (rad/seconds)      (seconds)

 -3.02e-01 + 8.06e-01i     8.61e-01     7.74e-02        1.93e+02          6.68e-02
 -3.02e-01 - 8.06e-01i     8.61e-01     7.74e-02        1.93e+02          6.68e-02
 -5.40e+00                 5.40e+00    -4.73e-01        3.57e+02         -5.93e-03
```

The `Magnitude` column displays the discrete-time pole magnitudes. The `Damping`, `Frequency`, and `Time Constant` columns display values calculated using the equivalent continuous-time poles.

Obtain vectors containing the natural frequencies and damping ratios of the poles.

```
[Wn,zeta] = damp(H);
```

Calculate the associated time constants.

```
tau = 1./(zeta.*Wn);
```

More About

Algorithms

The natural frequency, time constant, and damping ratio of the system poles are defined in the following table:

	Continuous Time	**Discrete Time with Sample Time Ts**
Pole Location	s	z
Equivalent Continuous-Time Pole	Not applicable	$s = \frac{ln(z)}{T_s}$
Natural Frequency	$\omega_n = \lvert s \rvert$	$\omega_n = \lvert s \rvert = \left\lvert \frac{ln(z)}{T_s} \right\rvert$
Damping Ratio	$\zeta = -cos(\angle s)$	$\zeta = -cos(\angle s) = -cos(\angle ln(z))$
Time Constant	$\tau = \frac{1}{\omega_n \zeta}$	$\tau = \frac{1}{\omega_n \zeta}$

See Also

`eig` | `esort` | `dsort` | `pole` | `pzmap` | `zero`

Introduced before R2006a

data2state

Estimate current states of model

Syntax

```
X = data2state(sys,PastData)
[X,XCov] = data2state(sys,PastData)
```

Description

`X = data2state(sys,PastData)` estimates the current states, `X`, of identified model, `sys`, based on the past data.

`X` contains the state values at the time instant immediately after the most recent data sample in `PastData`.

`data2state` is useful for continued model simulation. If you have simulated a model up to a certain time instant and would like to then simulate the model for future inputs, use `data2state` to estimate states of the model at the beginning of the second simulation.

`[X,XCov] = data2state(sys,PastData)` returns the estimated covariance, `XCov`, of the current states.

Examples

Compute Current States of Identified Model

Load data.

```
load iddata3 z3
```

Divide the data into past data and future data.

```
PastData = z3(1:150);
FutureData = z3(151:end);
```

Estimate a state-space model.

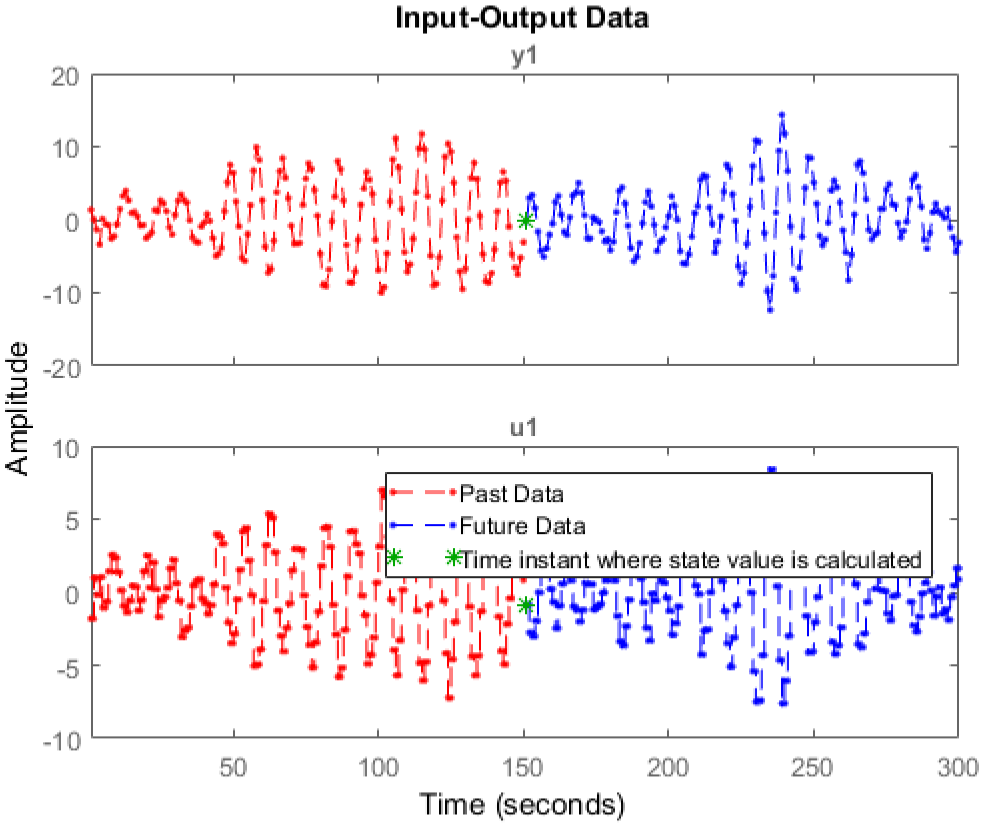
```
sys = ssest(z3,2);
```

Calculate the initial states at the start of `FutureData` using `PastData`.

```
X = data2state(sys,PastData);
```

`data2state` returns the state values at the time instant immediately after the most recent data sample in `PastData`. This time point is also the start of the future data.

```
plot(PastData,'r--.',FutureData,'b--.',...
    FutureData(1),'g*');
legend('Past Data','Future Data','Time instant where state value is calculated')
```



Simulate the model using `FutureData`, and set the initial state values to `X`.

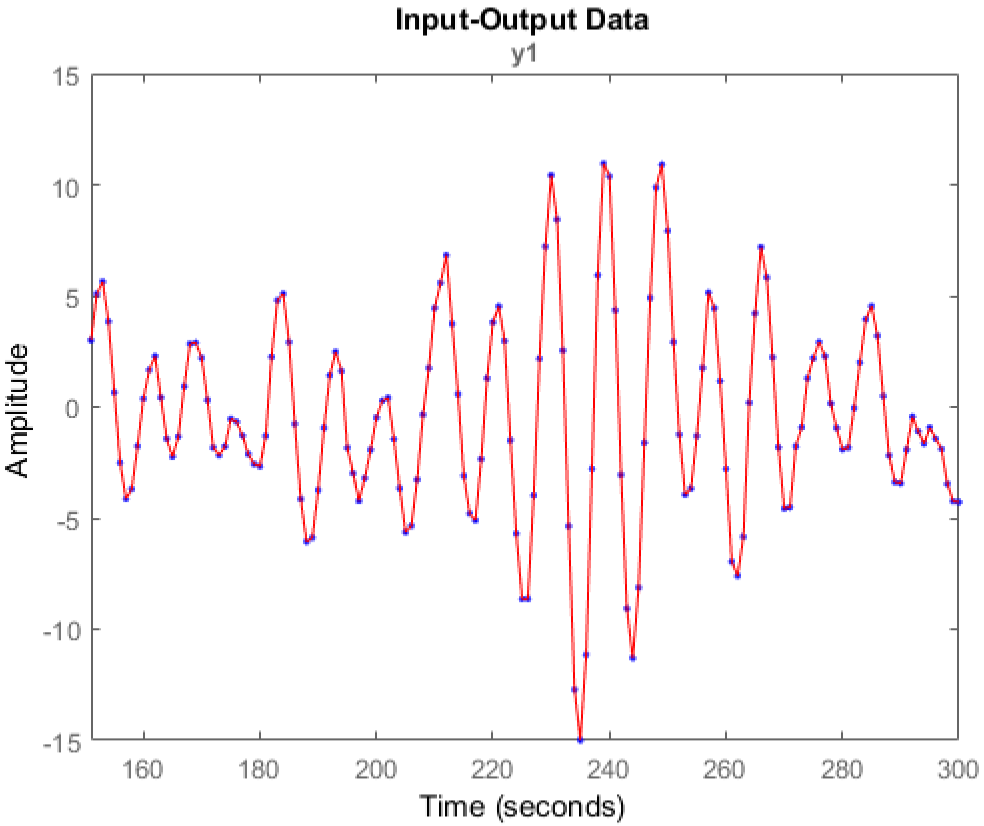
```
opt = simOptions('InitialCondition',X);
y_f  = sim(sys,FutureData,opt);
```

Simulate the model using the entire data, `z3`.

```
y_all = sim(sys,z3);
```

Plot the simulated response, `y_all`, for time samples 151 to 300. Verify that `y_f` matches the simulation.

```
plot(y_all(151:end),'.', y_f,'r')
```



Calculate Current States and Covariance of States

Load the past data.

```
load iddata1 z1
PastData = z1;
```

Estimate an ARX model.

```
sys = arx(PastData,[1 1 0]);
```

Calculate the current states and covariance of states using `PastData`.

```
[X,XCov] = data2state(sys,PastData);
```

Since `sys` is an ARX model, it is converted to a state-space model, and the current states of the converted model are calculated.

Determine Current State of a Nonlinear ARX model

Load your data and create a data object.

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

Estimate a nonlinear ARX model.

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

The estimated model has six inputs and two outputs.

Determine the model order, `nx`.

```
nx = order(mw1);
```

Use the first `nx` samples of data to generate initial conditions.

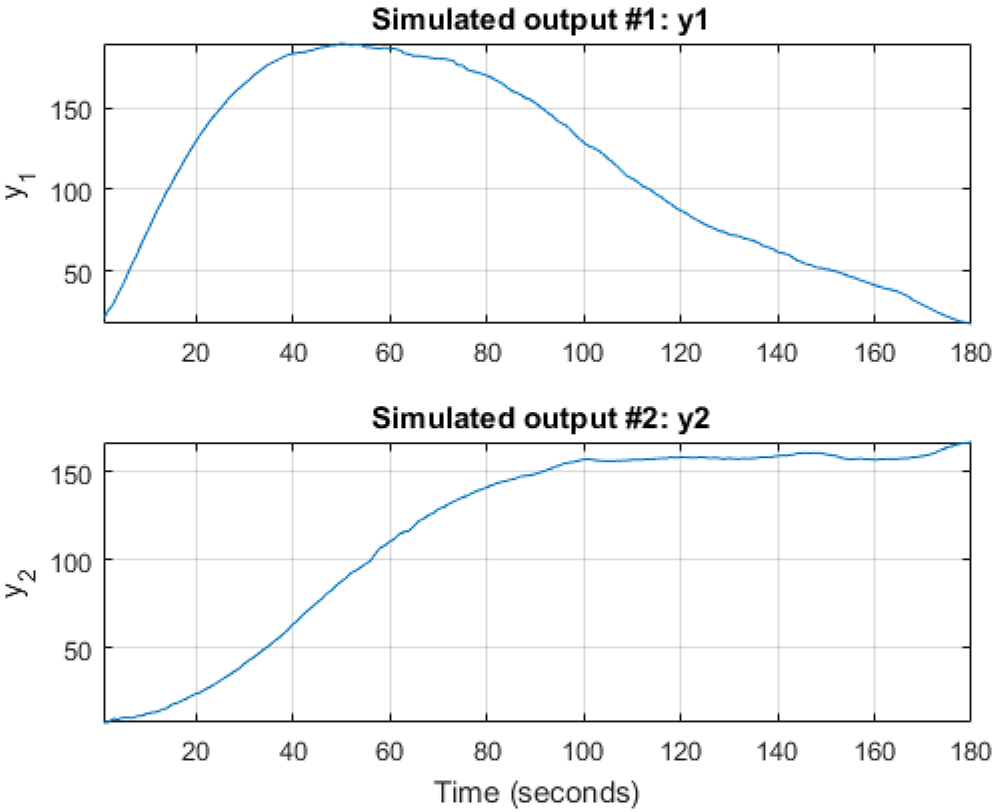
```
PastData = struct('Input', z.u(1:nx,:),'Output',z.y(1:nx,:));
```

Compute the current states of the model.

```
X = data2state(mw1,PastData);
```

Simulate the model using the remaining input data, and specify the initial conditions for simulation.

```
InputSignal = z.u(nx+1:end,:);
opt = simOptions;
opt.InitialCondition = X;
sim(mw1,InputSignal,opt)
```



Input Arguments

`sys` — Identified model

linear model | nonlinear model

Identified model whose current states are estimated, specified as one of the following:

- Linear model (`idpoly`, `idproc`, `idss`, `idtf`, `idgrey`) – If `sys` is not a state-space model (`idss` or `idgrey`), then it is converted to a state-space model, and the current states of the converted model are calculated.

If conversion of `sys` to `idss` is not possible, `X` is returned empty. For example, if `sys` is a MIMO continuous-time model with irreducible internal delays, then `X` is empty.

- Nonlinear model (`idnlgrey`, `idnlhw`, `idnlarx`) — For a definition of the states of `idnlarx` and `idnlhw` models, see "Definition of idnlarx States" on page 1-528, and "Definition of `idnlhw` States" on page 1-567.

`PastData` — Past input-output data

`iddata` object | structure

Past input-output data, specified as one of the following:

- `iddata` object — The number of samples must be 1 (constant data) or greater than or equal to the model order. To determine model order, use `order`.

  `X` is the value of model states at time `PastData.SamplingInstants(end) + PastData.Ts`.

  When `sys` is continuous-time, specify `PastData` as an `iddata` object.
- Structure — Specified as a structure with the following fields:
  - Input — Past input data, specified as one of the following:
    - 1-by-*Nu* row vector — Denotes a constant signal level for each input, where *Nu* is the number of inputs.

      For example, if a 3 input model has constant inputs with values 10,20, and 30, use `PastData.Input = [10,20,30]`.
    - *N*-by-*Nu* matrix — Where *N* is great than or equal to the model order.
  - Output — Past output data, specified as one of the following:
    - 1-by-*Ny* row vector — Denotes a constant signal level for each output, where *Ny* is the number of outputs.

      For example, if a 3-output model has constant outputs with values 15,25, and 35, use `PastData.Output = [15,25,35]`
    - *N*-by-*Ny* matrix — Where *N* is great than or equal to the model order.

  Specify `PastData` as a structure only when `sys` is a discrete-time model.

The data samples in `PastData` should be in the order of increasing time. That is, the last row in `PastData` should correspond to the latest time.

Output Arguments

`X` — Current state of model

row vector

Current state of model, returned as a row vector of size equal to the number of states. `X` is the states values at the time instant immediately after the most recent data sample in `PastData`.

When `sys` is not a state-space model (`idss`, `idgrey`, or `idnlgrey`), the definition of states depends on if `sys` is linear or nonlinear:

- Linear model (`idpoly`, `idproc`, `idtf`) – `sys` is converted to a state-space model, and the current states of the converted model are calculated.

  If conversion of `sys` to `idss` is not possible, `X` is returned empty. For example, if `sys` is a MIMO continuous-time model with irreducible internal delays.
- Nonlinear model (`idnlhw` or `idnlarx`) — For a definition of the states of `idnlarx` and `idnlhw` models, see “Definition of idnlarx States” on page 1-528, and “Definition of `idnlhw` States” on page 1-567.

`XCov` — Estimated covariance of state values

matrix

Estimated covariance of state values, returned as a matrix of size *Nx*-by-*Nx*, where *Nx* is the number of states.

`XCov` is empty if `sys` is a nonlinear ARX (`idnlarx`) or Hammerstein-Wiener model (`idnlhw`).

See Also

`findstates` | `getDelayInfo` | `idnlarx/findop` | `order` | `sim`

Introduced in R2008a

db2mag

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`

Introduced in R2008a

dcgain

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

$K = D + C\,(I - A)^{-1}B$

Examples

Example 1

To compute the DC gain of the MIMO transfer function

$$H(s) = \begin{bmatrix} 1 & \dfrac{s-1}{s^2+s+3} \\ \dfrac{1}{s+1} & \dfrac{s+2}{s-3} \end{bmatrix}$$

type

```
H = [1 tf([1 -1],[1 1 3]) ; tf(1,[1 1]) tf([1 2],[1 -3])];
dcgain(H)
```

to get the result:

```
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`.

More About

Tips

The DC gain is infinite for systems with integrators.

See Also

`evalfr` | `norm`

Introduced in R2012a

deadzone

Create a dead-zone nonlinearity estimator object

Syntax

```
NL = deadzone
NL = deadzone('ZeroInterval',[a,b])
```

Description

`NL = deadzone` creates a default dead-zone nonlinearity estimator object for estimating Hammerstein-Wiener models. The interval in which the dead-zone exists (zero interval) is set to `[NaN NaN]`. The initial value of the zero interval is determined from the estimation data range, during estimation using `nlhw`. Use dot notation to customize the object properties, if needed.

`NL = deadzone('ZeroInterval',[a,b])` creates a dead-zone nonlinearity estimator object initialized with zero interval, `[a,b]`.

Alternatively, use `NL = deadzone([a,b])`.

Object Description

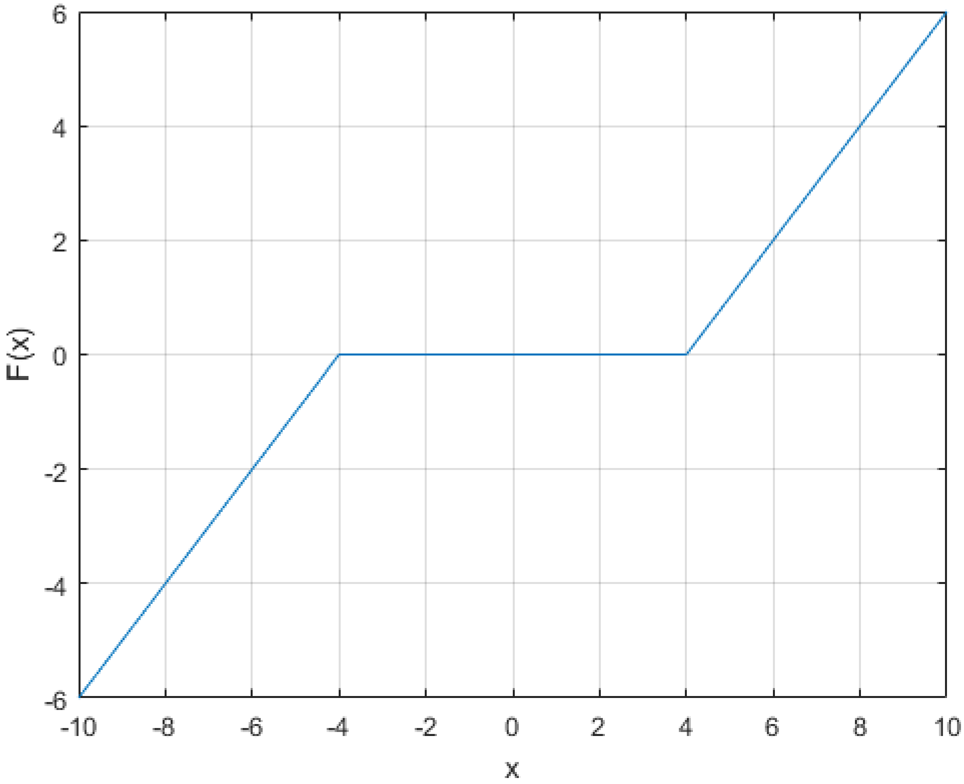
`deadzone` is an object that stores the dead-zone nonlinearity estimator for estimating Hammerstein-Wiener models.

Use `deadzone` to define a nonlinear function $y = F(x,\theta)$, where y and x are scalars, and θ represents the parameters a and b, which define the zero interval.

The dead-zone nonlinearity function has the following characteristics:

$a \le x < b$	$F(x) = 0$
$x < a$	$F(x) = x - a$
$x \ge b$	$F(x) = x - b$

For example, in the following plot, the dead-zone is in the interval `[-4,4]`.



The value `F(x)` is computed by `evaluate(NL,x)`, where `NL` is the `deadzone` object.

For `deadzone` object properties, see "Properties" on page 1-262.

Examples

Create a Default Dead-Zone Nonlinearity Estimator

```
NL = deadzone;
```

Specify the zero interval.

```
NL.ZeroInterval = [-4,5];
```

Estimate a Hammerstein-Wiener Model with Dead-zone Nonlinearity

Load estimation data.

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

Create a `deadzone` object, and specify the initial guess for the zero-interval.

```
OutputNL = deadzone('ZeroInterval',[-0.1 0.1]);
```

Estimate model with no input nonlinearity.

```
m = nlhw(z1,[2 3 0],[],OutputNL);
```

Estimate MIMO Hammerstein-Wiener Model

Load the estimation data.

```
load motorizedcamera;
```

Create an `iddata` object.

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

`z` is an `iddata` object with 6 inputs and 2 outputs.

Specify the model orders and delays.

```
Orders = [ones(2,6),ones(2,6),ones(2,6)];
```

Specify the same nonlinearity estimator for each input channel.

```
InputNL = saturation;
```

Specify different nonlinearity estimators for each output channel.

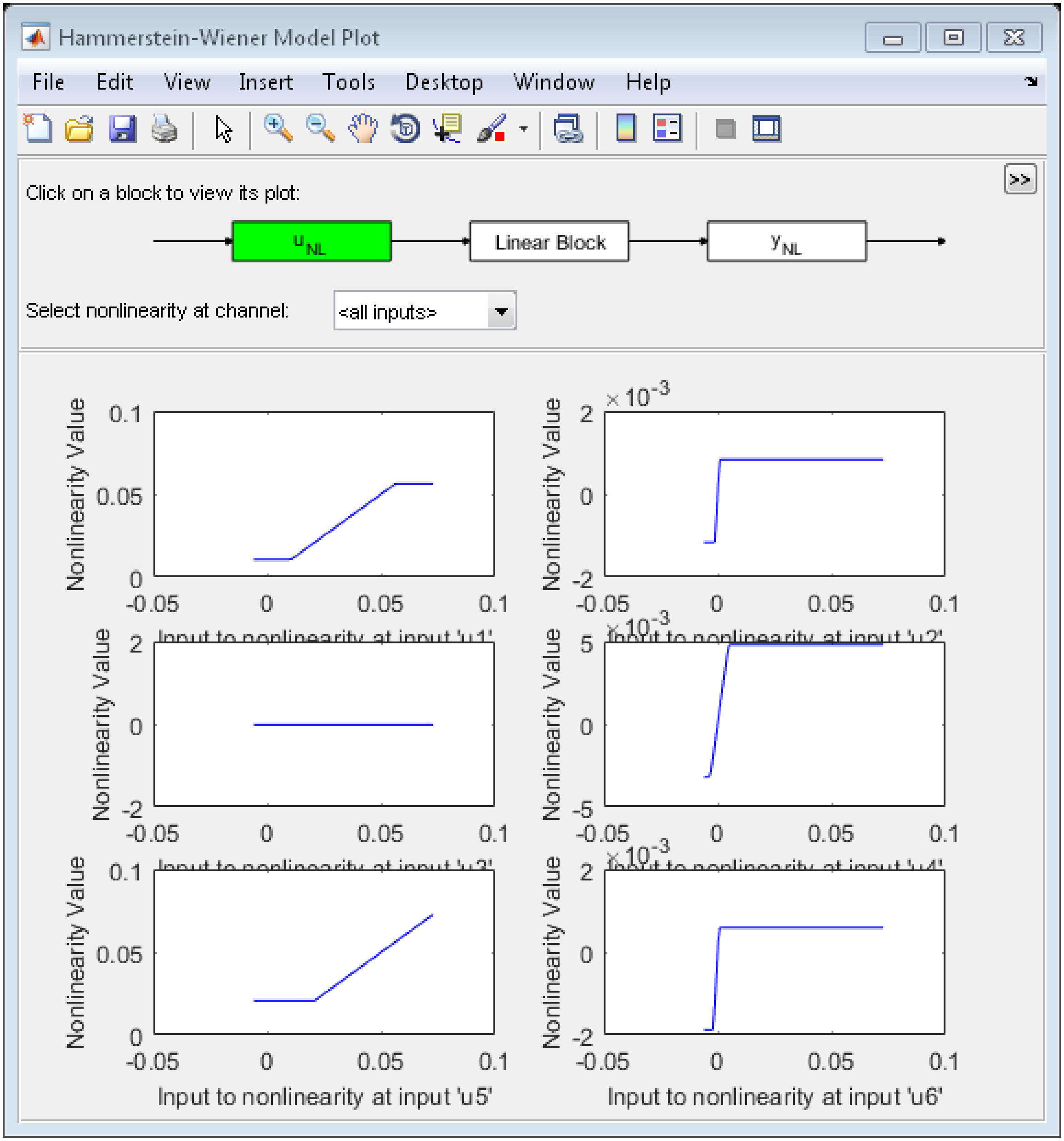
```
 OutputNL = [deadzone,wavenet];
```

Estimate the Hammerstein-Wiener model.

```
sys = nlhw(z,Orders,InputNL,OutputNL);
```

To see the shape of the estimated input and output nonlinearities, plot the nonlinearities.

```
plot(sys)
```



Click on a block to view its plot:

Select nonlinearity at channel: <all inputs>

Click on the input and output nonlinearity blocks on the top of the plot to see the nonlinearites.

Input Arguments

`[a,b]` — Zero interval

`[NaN NaN]` (default) | 2–element row vector

Zero interval of the dead-zone, specified as a 2–element row vector of doubles.

The dead-zone nonlinearity is initialized at the interval `[a,b]`. The interval values are adjusted to the estimation data by `nlhw`. To remove the lower limit, set `a` to `-Inf`. The lower limit is not adjusted during estimation. To remove the upper limit, set `b` to `Inf`. The upper limit is not adjusted during estimation.

When the interval is `[NaN NaN]`, the initial value of the zero interval is determined from the estimation data range during estimation using `nlhw`.

Example: `[-2 1]`

Properties

`ZeroInterval`

Zero interval of the dead-zone, specified as a 2–element row vector of doubles.

Default: `[NaN NaN]`

Output Arguments

`NL` — Dead-zone nonlinearity estimator object

`deadzone` object

Dead-zone nonlinearity estimator object, returned as a `deadzone` object.

See Also

`nlhw`

Introduced in R2007a

delayest

Estimate time delay (dead time) from data

Syntax

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

Description

`nk = delayest(Data)` estimates time delay from data. `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:

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

`nk = delayest(Data,na,nb,nkmin,nkmax,maxtest)` specifies additional options. 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).

Introduced before R2006a

detrend

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 Signals

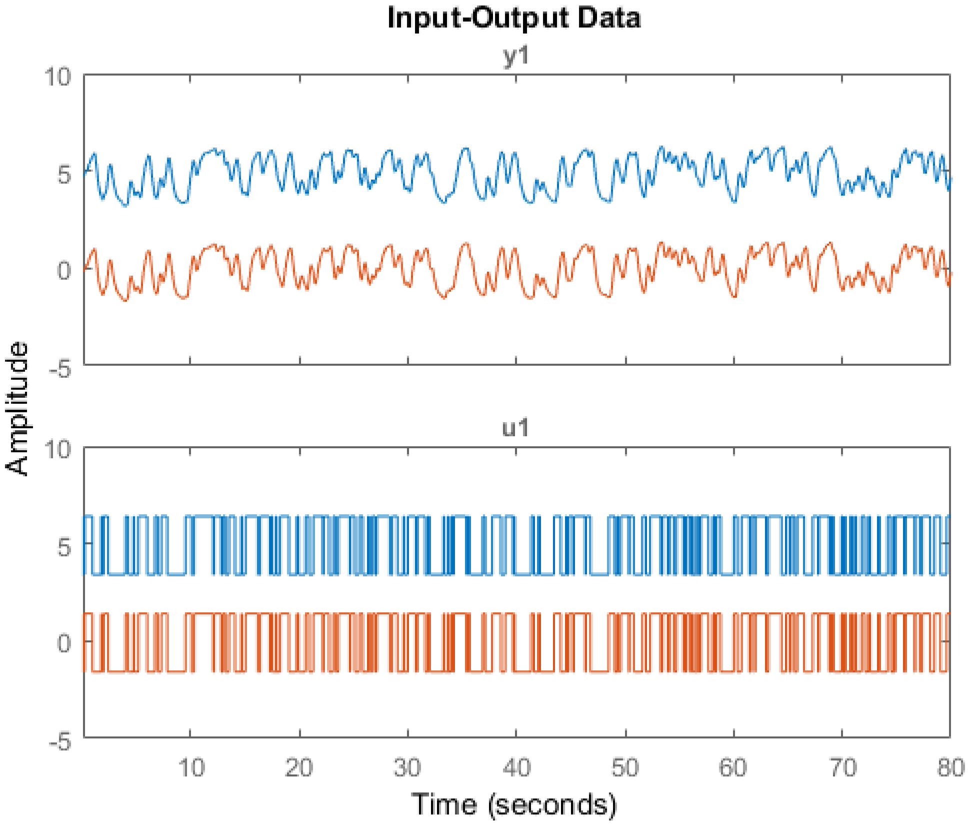
Load the data.

```
load dryer2
```

```
data = iddata(y2,u2,0.08);
```

Subtract mean values from input and output signals and store the trend information.

```
[data_d,T] = detrend(data,0);
plot(data,data_d)
```



Note that the detrended data has zero mean value.

Remove Offsets From Data

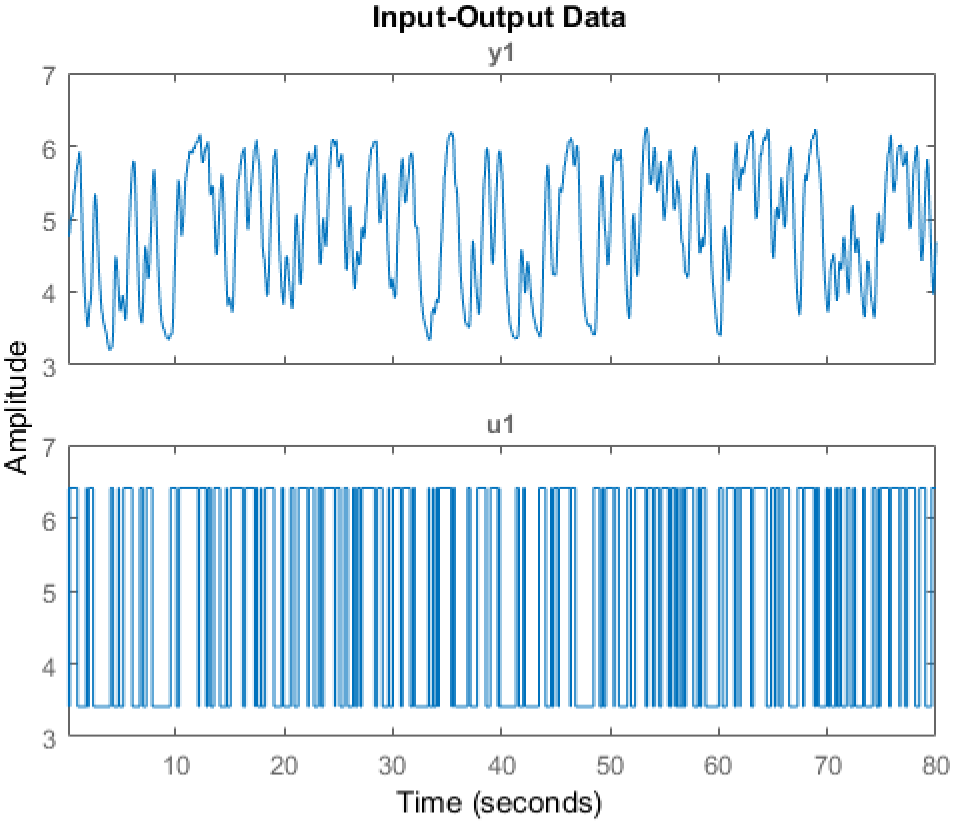
Remove specified offset from input and output signals.

Load SISO data containing vectors `u2` and `y2`.

```
load dryer2
```

Create a data object with sample time of 0.08 seconds and plot it.

```
data = iddata(y2,u2,0.08);
plot(data)
```



The data has a nonzero mean value.

Store the data offset and trend information in a `TrendInfo` object.

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

Assign offset values to the `TrendInfo` object.

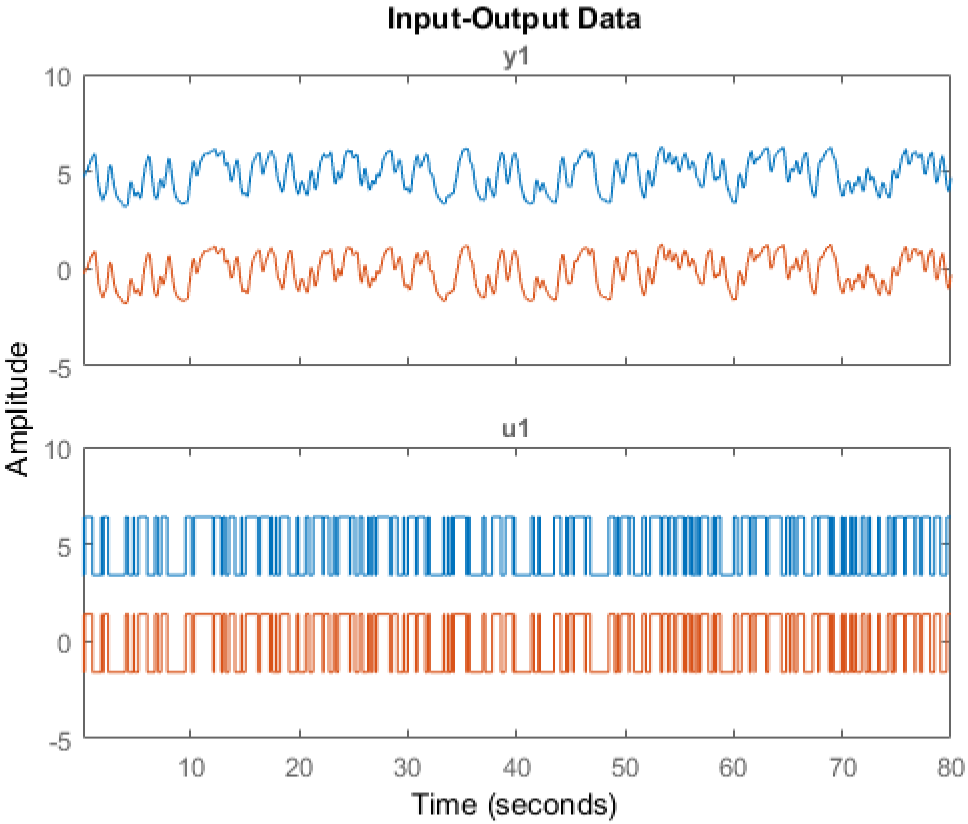
```
T.InputOffset = 5;
T.OutputOffset = 5;
```

Subtract offset from the data.

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

Plot the detrended data on the same plot.

```
hold on
plot(data_d)
```



View the mean value removed from the data.

```
get(T)
```

```
ans = 

  struct with fields:

        DataName: 'data'
     InputOffset: 5
    OutputOffset: 5
      InputSlope: 0
     OutputSlope: 0
```

Subtract Several Linear Trends

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 Mean Value and a V-shaped Trend

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];
```

More About

- "Handling Offsets and Trends in Data"

See Also

| `TrendInfo` | `getTrend`

Introduced before R2006a

diff

Difference signals in iddata objects

Syntax

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

Description

`zdi = diff(z)` and `zdi = diff(z,n)` return the difference signals in `iddata` objects. `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`.

Introduced before R2006a

etfe

Estimate empirical transfer functions and periodograms

Syntax

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

Description

`g = etfe(data)` estimates a transfer function of the form:

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

`data` contains time- or frequency-domain input-output data or time-series data:

- If `data` is time-domain input-output signals, `g` is the ratio of the output Fourier transform to the input Fourier transform for the data.

  For nonperiodic data, the transfer function is estimated at 128 equally-spaced frequencies `[1:128]/128*pi/Ts`.

  For periodic data that contains a whole number of periods (`data.Period = integer`), the response is computed at the frequencies `k*2*pi/period` for `k = 0` up to the Nyquist frequency.
- If `data` is frequency-domain input-output signals, `g` is the ratio of output to input at all frequencies, where the input is nonzero.
- If `data` is time-series data (no input channels), `g` is the periodogram, that is the normed absolute square of the Fourier transform, of the data. The corresponding spectral estimate is normalized, as described in "Spectrum Normalization" and differs from the `spectrum` normalization in the Signal Processing Toolbox™ product.

`g = etfe(data,M)` applies a smoothing operation on the raw spectral estimates using a Hamming Window that yields a frequency resolution of about `pi/M`. The effect of `M` is similar to the effect of `M` in `spa`. `M` is ignored for periodic data. Use this syntax as an alternative to `spa` for narrowband spectra and systems that require large values of `M`.

`g = etfe(data,M,N)` specifies the frequency spacing for nonperiodic data.

- For nonperiodic time-domain data, `N` specifies the frequency grid `[1:N]/N*pi/Ts` rad/TimeUnit. When not specified, `N` is 128.
- For periodic time-domain data, `N` is ignored.
- For frequency-domain data, the `N` is `fmin:delta_f:fmax`, where `[fmin fmax]` is the range of frequencies in `data`, and `delta_f` is `(fmax-fmin)/(N-1)` rad/TimeUnit. When not specified, the response is computed at the frequencies contained in data where input is nonzero.

Examples

Compare an Empirical Transfer Function to a Smoothed Spectral Estimate

Load estimation data.

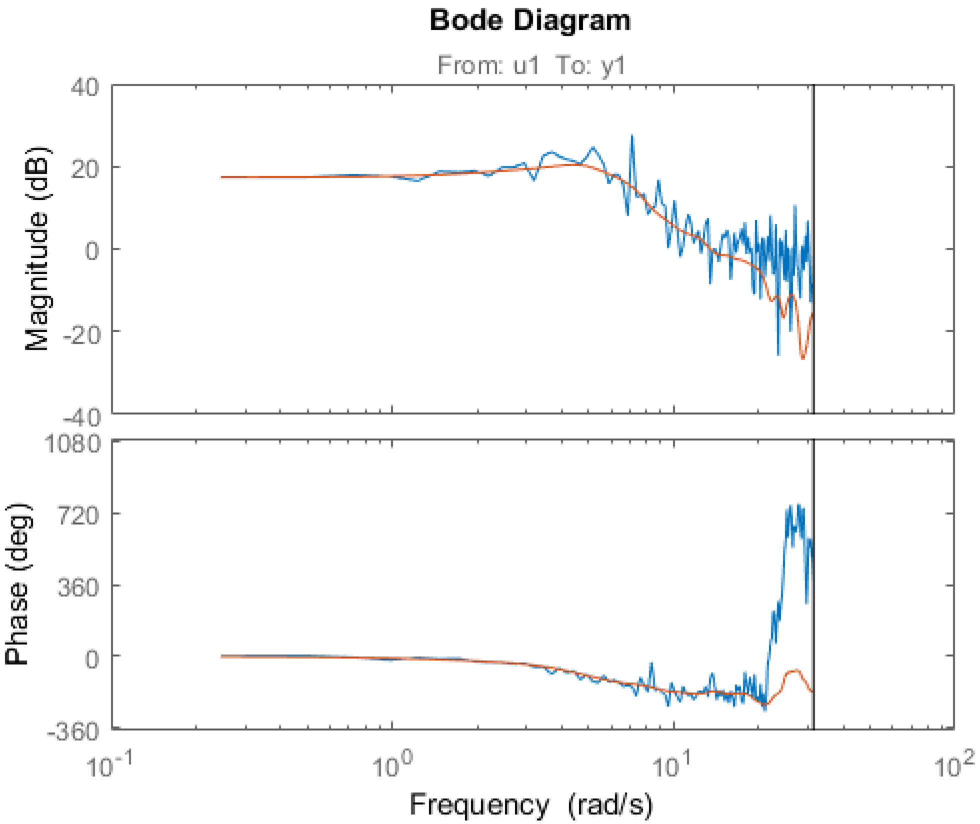
```
load iddata1 z1;
```

Estimate empirical transfer function and smoothed spectral estimate.

```
ge = etfe(z1);
gs = spa(z1);
```

Compare the two models on a Bode plot.

```
bode(ge,gs)
```



Generate Empirical Transfer Function Using Periodic Input

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

Generate a periodic input signal and output signal using simulation.

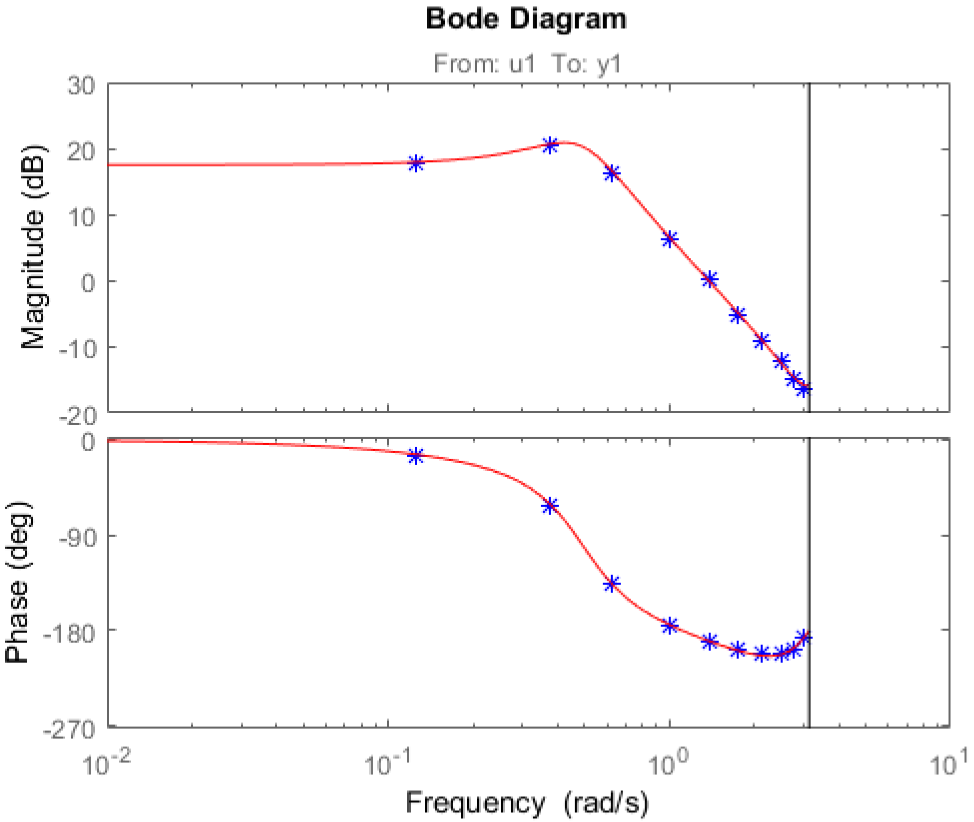
```
m = idpoly([1 -1.5 0.7],[0 1 0.5]);
u = iddata([],idinput([50,1,10],'sine'));
u.Period = 50;
y = sim(m,u);
```

Estimate an empirical transfer function.

```
me = etfe([y u]);
```

Compare the empirical transfer function with the original model.

```
bode(me,'b*',m,'r')
```



Apply Smoothing Operation on Empirical Transfer Function Estimate

Perform a smoothing operation on raw spectral estimates using a Hamming Window and compare the responses.

Load data.

```
load iddata1
```

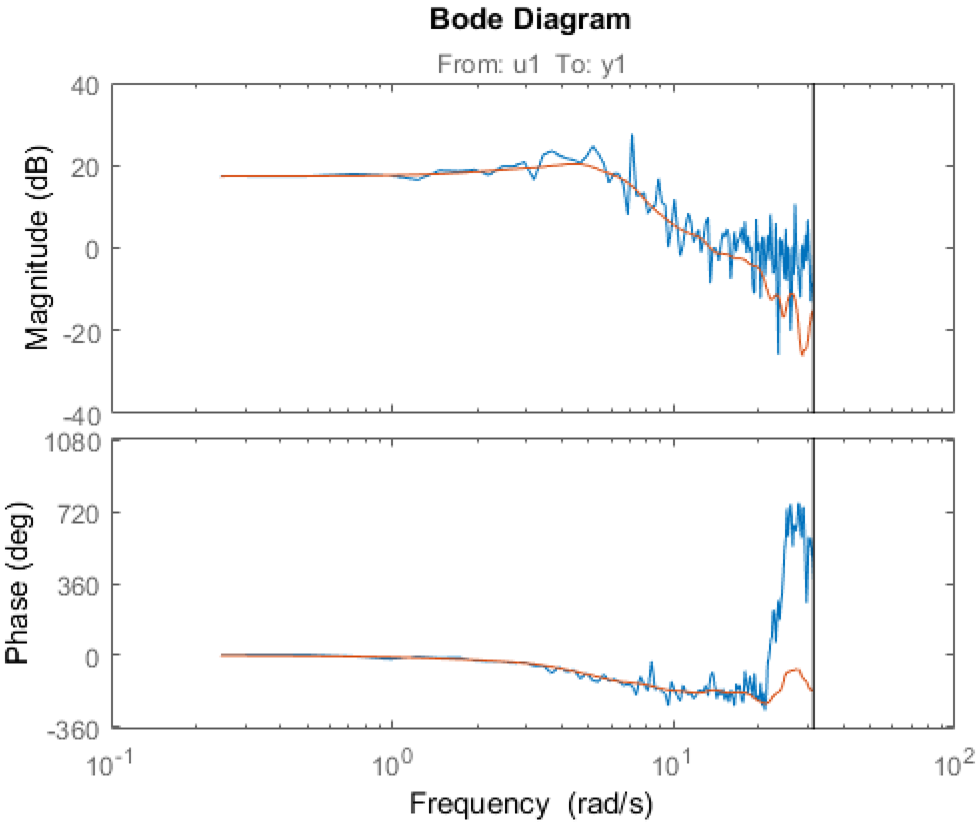
Estimate empirical transfer functions with and without the smoothing operation.

```
ge1 = etfe(z1);
ge2 = etfe(z1,32);
```

Compare the models on a Bode plot.

`ge2` is smoother than `ge1` because of the effect of the smoothing operation.

```
bode(ge1,ge2)
```



Compare Effect of Frequency Spacing on Empirical Transfer Function Estimate

Estimate empirical transfer functions with low- and high-frequency spacings and compare the responses.

Load data.

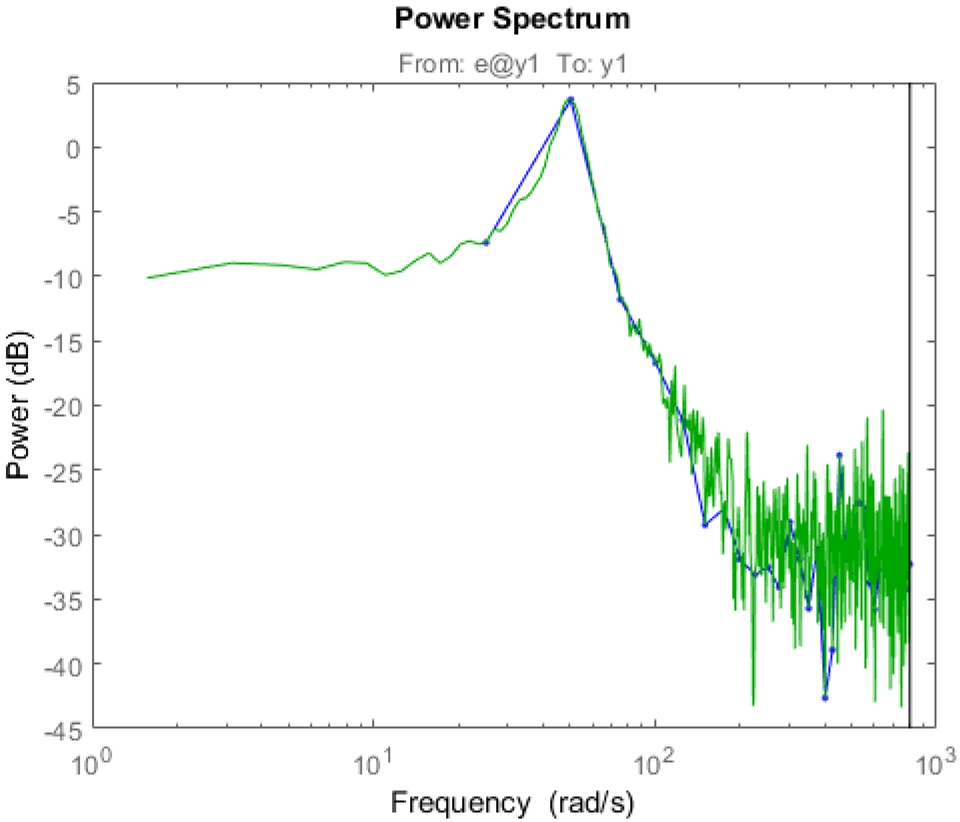
```
load iddata9
```

Estimate empirical transfer functions with low and high frequency spacings.

```
ge1 = etfe(z9,[],32);
ge2 = etfe(z9,[],512);
```

Plot the output power spectrum of the two models.

```
spectrum(ge1,'b.-',ge2,'g')
```



- “Estimate Frequency-Response Models at the Command Line”

Input Arguments

`data` — Estimation data

`iddata`

Estimation data, specified as an `iddata` object. The data can be time- or frequency-domain input/output signals or time-series data.

`M` — Frequency resolution

`[]` (default) | positive scalar

Frequency resolution, specified as a positive scalar.

`N` — Frequency spacing

128 for nonperiodic time-domain data (default) | positive scalar

Frequency spacing, specified as a positive scalar. For frequency-domain data, the default frequency spacing is the spacing inherent in the estimation data.

Output Arguments

`g` — Transfer function estimate

`idfrd`

Transfer function estimate, returned as an `idfrd` model.

Information about the estimation results and options used is stored in the model's `Report` property. `Report` has the following fields:

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Status</code></td><td>Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.</td></tr>
<tr><td><code>Method</code></td><td>Estimation command used.</td></tr>
<tr><td><code>WindowSize</code></td><td>Size of the Hamming window.</td></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation, returned as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Type</code></td><td>Data type.</td></tr>
<tr><td><code>Length</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time.</td></tr>
<tr><td><code>InterSa</code></td><td>Input intersample behavior, returned as one of the following values:
• <code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
• <code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
• <code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.</td></tr>
<tr><td><code>InputOf</code></td><td>Offset removed from time-domain input data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
<tr><td><code>OutputO</code></td><td>Offset removed from time-domain output data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
</table>
</td></tr>
</table>

For more information on using `Report`, see “Estimation Report”.

More About

- “What is a Frequency-Response Model?”

See Also

`bode` | `freqresp` | `idfrd` | `impulseest` | `nyquist` | `spa` | `spafdr` | `spectrum`

Introduced before R2006a

evalfr

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]);
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`

Introduced in R2012a

extendedKalmanFilter

Create extended Kalman filter object for online state estimation

Syntax

```
obj = extendedKalmanFilter(StateTransitionFcn,MeasurementFcn,
InitialState)
obj = extendedKalmanFilter(StateTransitionFcn,MeasurementFcn,
InitialState,Name,Value)
obj = extendedKalmanFilter(Name,Value)
```

Description

`obj = extendedKalmanFilter(StateTransitionFcn,MeasurementFcn, InitialState)` creates an extended Kalman filter object for online state estimation of a discrete-time nonlinear system. `StateTransitionFcn` is a function that calculates the state of the system at time *k*, given the state vector at time *k*-1. `MeasurementFcn` is a function that calculates the output measurement of the system at time *k*, given the state at time *k*. `InitialState` specifies the initial value of the state estimates.

After creating the object, use the `correct` and `predict` commands to update state estimates and state estimation error covariance values using a first-order discrete-time extended Kalman filter algorithm and real-time data.

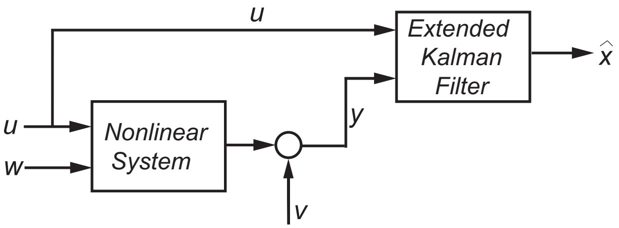
`obj = extendedKalmanFilter(StateTransitionFcn,MeasurementFcn, InitialState,Name,Value)` specifies additional attributes of the extended Kalman filter object using one or more `Name,Value` pair arguments.

`obj = extendedKalmanFilter(Name,Value)` creates an extended Kalman filter object with properties specified using one or more `Name,Value` pair arguments. The properties that you do not specify retain their default value.

Object Description

`extendedKalmanFilter` creates an object for online state estimation of a discrete-time nonlinear system using the first-order discrete-time extended Kalman filter algorithm.

Consider a plant with states *x*, input *u*, output *y*, process noise *w*, and measurement noise *v*. Assume that you can represent the plant as a nonlinear system.



The algorithm computes the state estimates $\hat{x}$ of the nonlinear system using state transition and measurement functions specified by you. The software lets you specify the noise in these functions as additive or nonadditive:

- **Additive Noise Terms** — The state transition and measurements equations have the following form:

  $$x[k] = f(x[k-1], u_s[k-1]) + w[k-1]$$
  $$y[k] = h(x[k], u_m[k]) + v[k]$$

  Here *f* is a nonlinear state transition function that describes the evolution of states `x` from one time step to the next. The nonlinear measurement function *h* relates `x` to the measurements `y` at time step `k`. `w` and `v` are the zero-mean, uncorrelated process and measurement noises, respectively. These functions can also have additional input arguments that are denoted by u_s and u_m in the equations. For example, the additional arguments could be time step `k` or the inputs `u` to the nonlinear system. There can be multiple such arguments.

  Note that the noise terms in both equations are additive. That is, `x(k)` is linearly related to the process noise `w(k-1)`, and `y(k)` is linearly related to the measurement noise `v(k)`.
- **Nonadditive Noise Terms** — The software also supports more complex state transition and measurement functions where the state *x*[*k*] and measurement *y*[*k*] are nonlinear functions of the process noise and measurement noise, respectively. When the noise terms are nonadditive, the state transition and measurements equation have the following form:

$$x[k] = f(x[k-1], w[k-1], u_s[k-1])$$
$$y[k] = h(x[k], v[k], u_m[k])$$

When you perform online state estimation, you first create the nonlinear state transition function f and measurement function h. You then construct the `extendedKalmanFilter` object using these nonlinear functions, and specify whether the noise terms are additive or nonadditive. You can also specify the Jacobians of the state transition and measurement functions. If you do not specify them, the software numerically computes the Jacobians.

After you create the object, you use the `predict` command to predict state estimate at the next time step, and `correct` to correct state estimates using the algorithm and real-time data. For information about the algorithm, see “Extended and Unscented Kalman Filter Algorithms for Online State Estimation”.

You can use the following commands with `extendedKalmanFilter` objects:

Command	**Description**
`correct`	Correct the state and state estimation error covariance at time step k using measured data at time step k.
`predict`	Predict the state and state estimation error covariance at time the next time step.
`clone`	Create another object with the same object property values. Do not create additional objects using syntax `obj2 = obj`. Any changes made to the properties of the new object created in this way (`obj2`) also change the properties of the original object (`obj`).

For `extendedKalmanFilter` object properties, see “Properties” on page 1-290.

Examples

Create Extended Kalman Filter Object for Online State Estimation

To define an extended Kalman filter object for estimating the states of your system, you first write and save the state transition function and measurement function for the system.

In this example, use the previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`. These functions describe a discrete-approximation to a van der Pol oscillator with nonlinearity parameter, mu, equal to 1. The oscillator has two states.

Specify an initial guess for the two states. You specify the guess as an `M`-element row or column vector, where `M` is the number of states.

```
initialStateGuess = [1;0];
```

Create the extended Kalman filter object. Use function handles to provide the state transition and measurement functions to the object.

```
obj = extendedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,initialStateGuess);
```

The object has a default structure where the process and measurement noise are additive.

To estimate the states and state estimation error covariance from the constructed object, use the `correct` and `predict` commands and real-time data.

Specify Process and Measurement Noise Covariances in Extended Kalman Filter Object

Create an extended Kalman filter object for a van der Pol oscillator with two states and one output. Use the previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`. These functions are written for additive process and measurement noise terms. Specify the initial state values for the two states as [2;0].

Since the system has two states and the process noise is additive, the process noise is a 2-element vector and the process noise covariance is a 2-by-2 matrix. Assume there is no cross-correlation between process noise terms, and both the terms have the same variance 0.01. You can specify the process noise covariance as a scalar. The software uses the scalar value to create a 2-by-2 diagonal matrix with 0.01 on the diagonals.

Specify the process noise covariance during object construction.

```
obj = extendedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,[2;0],...
    'ProcessNoise',0.01);
```

Alternatively, you can specify noise covariances after object construction using dot notation. For example, specify the measurement noise covariance as 0.2.

```
obj.MeasurementNoise = 0.2;
```

Since the system has only one output, the measurement noise is a 1-element vector and the `MeasurementNoise` property denotes the variance of the measurement noise.

Specify Jacobians for State and Measurement Functions

Create an extended Kalman filter object for a van der Pol oscillator with two states and one output. Use the previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`. Specify the initial state values for the two states as [2;0].

```
obj = extendedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,[2;0]);
```

The extended Kalman filter algorithm uses Jacobians of the state transition and measurement functions for state estimation. You write and save the Jacoban functions and provide them as function handles to the object. In this example, use the previously written and saved functions `vdpStateJacobianFcn.m` and `vdpMeasurementJacobianFcn.m`.

```
obj.StateTransitionJacobianFcn = @vdpStateJacobianFcn.m;
obj.MeasurementJacobianFcn = @vdpMeasurementJacobianFcn;
```

Note that if you do not specify the Jacobians of the functions, the software numerically computes the Jacobians. This numerical computation may result in increased processing time and numerical inaccuracy of the state estimation.

Specify Nonadditive Measurement Noise in Extended Kalman Filter Object

Create an extended Kalman filter object for a van der Pol oscillator with two states and one output. Assume that the process noise terms in the state transition function are additive. That is, there is a linear relation between the state and process noise. Also assume that the measurement noise terms are nonadditive. That is, there is a nonlinear relation between the measurement and measurement noise.

```
obj = extendedKalmanFilter('HasAdditiveMeasurementNoise',false);
```

Specify the state transition function and measurement functions. Use the previously written and saved functions, `vdpStateFcn.m` and `vdpMeasurementNonAdditiveNoiseFcn.m`.

The state transition function is written assuming the process noise is additive. The measurement function is written assuming the measurement noise is nonadditive.

```
obj.StateTransitionFcn = @vdpStateFcn;
obj.StateTransitionFcn = @vdpMeasurementNonAdditiveNoiseFcn;
```

Specify the initial state values for the two states as [2;0].

```
obj.State = [2;0];
```

You can now use the `correct` and `predict` commands to estimate the state and state estimation error covariance values from the constructed object.

Specify State Transition and Measurement Functions with Additional Inputs

Consider a nonlinear system with input `u` whose state `x` and measurement `y` evolve according to the folowing state transition and measurement equations:

$$x[k] = \sqrt{x[k-1] + u[k-1]} + w[k-1]$$

$$y[k] = x[k] + 2 * u[k] + v[k]^2$$

The process noise `w` of the system is additive while the measurement noise `v` is nonadditive.

Create the state transition function and measurement function for the system. Specify the functions with an additional input `u`.

```
f = @(x,u)(sqrt(x+u));
h = @(x,v,u)(x+2*u+v^2);
```

`f` and `h` are function handles to the anonymous functions that store the state transition and measurement functions, respectively. In the measurement function, because the measurement noise is nonadditive, `v` is also specified as an input. Note that `v` is specified as an input before the additional input `u`.

Create an extended Kalman filter object for estimating the state of the nonlinear system using the specified functions. Specify the initial value of the state as 1, and the measurement noise as nonadditive.

```
obj = extendedKalmanFilter(f,h,1,'HasAdditiveMeasurementNoise',false);
```

Specify the measurement noise covariance.

```
obj.MeasurementNoise = 0.01;
```

You can now estimate the state of the system using the `predict` and `correct` commands. You pass the values of `u` to `predict` and `correct`, which in turn pass them to the state transition and measurement functions, respectively.

Correct the state estimate with measurement y[k]=0.8 and input u[k]=0.2 at time step k.

```
correct(obj,0.8,0.2)
```

Predict the state at next time step, given u[k]=0.2.

```
predict(obj,0.2)
```

- "Nonlinear State Estimation Using Unscented Kalman Filter"
- "Fault Detection Using an Extended Kalman Filter"
- "Generate Code for Online State Estimation in MATLAB"

Input Arguments

`StateTransitionFcn` — State transition function

function handle

State transition function *f*, specified as a function handle. The function calculates the *M*-element state vector of the system at time step *k*, given the state vector at time step *k*-1. *M* is the number of states of the nonlinear system.

You write and save the state transition function for your nonlinear system, and use it to construct the object. For example, if `vdpStateFcn.m` is the state transition function, specify `StateTransitionFcn` as `@vdpStateFcn`. You can also specify `StateTransitionFcn` as a function handle to an anonymous function.

The inputs to the function you write depend on whether you specify the process noise as additive or nonadditive in the `HasAdditiveProcessNoise` property of the object:

- `HasAdditiveProcessNoise` is true — The process noise `w` is additive, and the state transition function specifies how the states evolve as a function of state values at the previous time step:

```
x(k) = f(x(k-1),Us1,...,Usn)
```

  Where `x(k)` is the estimated state at time `k`, and `Us1,...,Usn` are any additional input arguments required by your state transition function, such as system inputs or the sample time. During estimation, you pass these additional arguments to the `predict` command, which in turn passes them to the state transition function.

- `HasAdditiveProcessNoise` is false — The process noise is nonadditive, and the state transition function also specifies how the states evolve as a function of the process noise:

  ```
  x(k) = f(x(k-1),w(k-1),Us1,...,Usn)
  ```

To see an example of a state transition function with additive process noise, type `edit vdpStateFcn` at the command line.

`MeasurementFcn` — Measurement function

function handle

Measurement function *h*, specified as a function handle. The function calculates the *N*-element output measurement vector of the nonlinear system at time step *k*, given the state vector at time step *k*. *N* is the number of measurements of the system. You write and save the measurement function, and use it to construct the object. For example, if `vdpMeasurementFcn.m` is the measurement function, specify `MeasurementFcn` as `@vdpMeasurementFcn`. You can also specify `MeasurementFcn` as a function handle to an anonymous function.

The inputs to the function depend on whether you specify the measurement noise as additive or nonadditive in the `HasAdditiveMeasurementNoise` property of the object:

- `HasAdditiveMeasurementNoise` is true — The measurement noise `v` is additive, and the measurement function specifies how the measurements evolve as a function of state values:

  ```
  y(k) = h(x(k),Um1,...,Umn)
  ```

  Where `y(k)` and `x(k)` are the estimated output and estimated state at time `k`, and `Um1,...,Umn` are any optional input arguments required by your measurement function. For example, if you are using multiple sensors for tracking an object, an additional input could be the sensor position. During estimation, you pass these additional arguments to the `correct` command, which in turn passes them to the measurement function.

- `HasAdditiveMeasurementNoise` is false — The measurement noise is nonadditive, and the measurement function also specifies how the output measurement evolves as a function of the measurement noise:

  ```
  y(k) = h(x(k),v(k),Um1,...,Umn)
  ```

To see an example of a measurement function with additive process noise, type `edit vdpMeasurementFcn` at the command line. To see an example of a measurement function with nonadditive process noise, type `edit vdpMeasurementNonAdditiveNoiseFcn`.

`InitialState` — Initial state estimate value

vector

Initial state estimate value, specified as an *M*-element vector, where *M* is the number of states in the system. Specify the initial state values based on your knowledge of the system.

The specified value is stored in the `State` property of the object. If you specify `InitialState` as a column vector, then `State` is also a column vector, and the `predict` and `correct` commands return state estimates as a column vector. Otherwise, a row vector is returned.

If you want a filter with single-precision floating-point variables, specify `InitialState` as a single-precision vector variable. For example, for a two-state system with state transition and measurement functions `vdpStateFcn.m` and `vdpMeasurementFcn.m`, create the extended Kalman filter object with initial state estimates `[1;2]` as follows:

```
obj = extendedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,single([1;2]))
```

Data Types: `double` | `single`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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 properties of `extendedKalmanFilter` object during object creation. For example, to create an extended Kalman filter object and specify the process noise covariance as 0.01:

```
obj = extendedKalmanFilter(StateTransitionFcn,MeasurementFcn,InitialState,'ProcessNois
```

Properties

`extendedKalmanFilter` object properties are of three types:

- Tunable properties that you can specify multiple times, either during object construction using `Name,Value` arguments, or any time afterward during state estimation. After object creation, use dot notation to modify the tunable properties.

  ```
  obj = extendedKalmanFilter(StateTransitionFcn,MeasurementFcn,InitialState);
  obj.ProcessNoise = 0.01;
  ```

  The tunable properties are `State`, `StateCovariance`, `ProcessNoise`, and `MeasurementNoise`.
- Nontunable properties that you can specify once, either during object construction or afterward using dot notion. Specify these properties before state estimation using `correct` and `predict`. The `StateTransitionFcn`, `MeasurementFcn`, `StateTransitionJacobianFcn`, and `MeasurementJacobianFcn` properties belong to this category.
- Nontunable properties that you must specify during object construction. The `HasAdditiveProcessNoise` and `HasAdditiveMeasurementNoise` properties belong to this category.

`HasAdditiveMeasurementNoise` — Measurement noise characteristics

`true` (default) | `false`

Measurement noise characteristics, specified as one of the following values:

- `true` — Measurement noise `v` is additive. The measurement function *h* that is specified in `MeasurementFcn` has the following form:

  ```
  y(k) = h(x(k),Um1,...,Umn)
  ```

  Where `y(k)` and `x(k)` are the estimated output and estimated state at time `k`, and `Um1,...,Umn` are any optional input arguments required by your measurement function.
- `false` — Measurement noise is nonadditive. The measurement function specifies how the output measurement evolves as a function of the state *and* measurement noise:

```
y(k) = h(x(k),v(k),Um1,...,Umn)
```

`HasAdditiveMeasurementNoise` is a nontunable property, and you can specify it only during object construction. You cannot change it using dot notation.

`HasAdditiveProcessNoise` — Process noise characteristics

`true` (default) | `false`

Process noise characteristics, specified as one of the following values:

- `true` — Process noise `w` is additive. The state transition function *f* specified in `StateTransitionFcn` has the following form:

  ```
  x(k) = f(x(k-1),Us1,...,Usn)
  ```

  Where `x(k)` is the estimated state at time `k`, and `Us1,...,Usn` are any additional input arguments required by your state transition function.
- `false` — Process noise is nonadditive. The state transition function specifies how the states evolve as a function of the state *and* process noise at the previous time step:

  ```
  x(k) = f(x(k-1),w(k-1),Us1,...,Usn)
  ```

`HasAdditiveProcessNoise` is a nontunable property, and you can specify it only during object construction. You cannot change it using dot notation.

`MeasurementFcn` — Measurement function

`[]` (default) | function handle

Measurement function h, specified as a function handle. The function calculates the N-element output measurement vector of the nonlinear system at time step k, given the state vector at time step k. N is the number of measurements of the system. You write and save the measurement function and use it to construct the object. For example, if `vdpMeasurementFcn.m` is the measurement function, specify `MeasurementFcn` as `@vdpMeasurementFcn`. You can also specify `MeasurementFcn` as a function handle to an anonymous function.

The inputs to the function depend on whether you specify the measurement noise as additive or nonadditive in the `HasAdditiveMeasurementNoise` property of the object:

- `HasAdditiveMeasurementNoise` is true — The measurement noise `v` is additive, and the measurement function specifies how the measurements evolve as a function of state values:

  `y(k) = h(x(k),Um1,...,Umn)`

  Where `y(k)` and `x(k)` are the estimated output and estimated state at time `k`, and `Um1,...,Umn` are any optional input arguments required by your measurement function. For example, if you are using multiple sensors for tracking an object, an additional input could be the sensor position. During estimation, you pass these additional arguments to the `correct` command which in turn passes them to the measurement function.

- `HasAdditiveMeasurementNoise` is false — The measurement noise is nonadditive, and the measurement function also specifies how the output measurement evolves as a function of the measurement noise:

  `y(k) = h(x(k),v(k),Um1,...,Umn)`

To see an example of a measurement function with additive process noise, type `edit vdpMeasurementFcn` at the command line. To see an example of a measurement function with nonadditive process noise, type `edit vdpMeasurementNonAdditiveNoiseFcn`.

`MeasurementFcn` is a nontunable property. You can specify it once before using the `correct` command either during object construction or using dot notation after object construction. You cannot change it after using the `correct` command.

`MeasurementJacobianFcn` — Jacobian of measurement function

[ ] (default) | function handle

Jacobian of measurement function *h*, specified as one of the following:

- `[]` — The Jacobian is numerically computed at every call to the `correct` command. This may increase processing time and numerical inaccuracy of the state estimation.
- function handle — You write and save the Jacobian function and specify the handle to the function. For example, if `vdpMeasurementJacobianFcn.m` is the Jacobian function, specify `MeasurementJacobianFcn` as `@vdpMeasurementJacobianFcn`.

  The function calculates the partial derivatives of the measurement function with respect to the states and measurement noise. The number of inputs to the Jacobian function must equal the number of inputs to the measurement function and must be specified in the same order in both functions. The number of outputs of the Jacobian function depends on the `HasAdditiveMeasurementNoise` property:

- `HasAdditiveMeasurementNoise` is true — The function calculates the partial derivatives of the measurement function with respect to the states ($\partial h / \partial x$). The output is as an N-by-M Jacobian matrix, where N is the number of measurements of the system and M is the number of states.
- `HasAdditiveMeasurementNoise` is false — The function also returns a second output that is the partial derivative of the measurement function with respect to the measurement noise terms ($\partial h / \partial v$). The second output is returned as an N-by-V Jacobian matrix, where V is the number of measurement noise terms.

To see an example of a Jacobian function for additive measurement noise, type `edit vdpMeasurementJacobianFcn` at the command line.

`MeasurementJacobianFcn` is a nontunable property. You can specify it once before using the `correct` command either during object construction or using dot notation after object construction. You cannot change it after using the `correct` command.

`MeasurementNoise` — Measurement noise covariance

1 (default) | scalar | matrix

Measurement noise covariance, specified as a scalar or matrix depending on the value of the `HasAdditiveMeasurementNoise` property:

- `HasAdditiveMeasurementNoise` is true — Specify the covariance as a scalar or an N-by-N matrix, where N is the number of measurements of the system. Specify a scalar if there is no cross-correlation between measurement noise terms and all the terms have the same variance. The software uses the scalar value to create an N-by-N diagonal matrix.
- `HasAdditiveMeasurementNoise` is false — Specify the covariance as a V-by-V matrix, where V is the number of measurement noise terms. `MeasurementNoise` must be specified before using `correct`. After you specify `MeasurementNoise` as a matrix for the first time, to then change `MeasurementNoise` you can also specify it as a scalar. Specify as a scalar if there is no cross-correlation between the measurement noise terms and all the terms have the same variance. The software extends the scalar to a V-by-V diagonal matrix with the scalar on the diagonals.

`MeasurementNoise` is a tunable property. You can change it using dot notation.

`ProcessNoise` — Process noise covariance

1 (default) | scalar | matrix

Process noise covariance, specified as a scalar or matrix depending on the value of the `HasAdditiveProcessNoise` property:

- `HasAdditiveProcessNoise` is true — Specify the covariance as a scalar or an *M*-by-*M* matrix, where *M* is the number of states of the system. Specify a scalar if there is no cross-correlation between process noise terms, and all the terms have the same variance. The software uses the scalar value to create an *M*-by-*M* diagonal matrix.
- `HasAdditiveProcessNoise` is false — Specify the covariance as a *W*-by-*W* matrix, where *W* is the number of process noise terms. `ProcessNoise` must be specified before using `predict`. After you specify `ProcessNoise` as a matrix for the first time, to then change `ProcessNoise` you can also specify it as a scalar. Specify as a scalar if there is no cross-correlation between the process noise terms and all the terms have the same variance. The software extends the scalar to a *W*-by-*W* diagonal matrix.

`ProcessNoise` is a tunable property. You can change it using dot notation.

`State` — State of nonlinear system

[ ] (default) | vector

State of the nonlinear system, specified as a vector of size *M*, where *M* is the number of states of the system.

When you use the `predict` command, `State` is updated with the predicted value at time step *k* using the state value at time step *k*–1. When you use the `correct` command, `State` is updated with the estimated value at time step *k* using measured data at time step *k*.

The initial value of `State` is the value you specify in the `InitialState` input argument during object creation. If you specify `InitialState` as a column vector, then `State` is also a column vector, and the `predict` and `correct` commands return state estimates as a column vector. Otherwise, a row vector is returned. If you want a filter with single-precision floating-point variables, you must specify `State` as a single-precision variable during object construction using the `InitialState` input argument.

`State` is a tunable property. You can change it using dot notation.

`StateCovariance` — State estimation error covariance

1 (default) | scalar | matrix

State estimation error covariance, specified as a scalar or an *M*-by-*M* matrix, where *M* is the number of states of the system. If you specify a scalar, the software uses the scalar value to create an *M*-by-*M* diagonal matrix.

Specify a high value for the covariance when you do not have confidence in the initial state values that you specify in the `InitialState` input argument.

When you use the `predict` command, `StateCovariance` is updated with the predicted value at time step *k* using the state value at time step *k*–1. When you use the `correct` command, `StateCovariance` is updated with the estimated value at time step *k* using measured data at time step *k*.

`StateCovariance` is a tunable property. You can change it using dot notation after using the `correct` or `predict` commands.

`StateTransitionFcn` — State transition function

[ ] (default) | function handle

State transition function *f*, specified as a function handle. The function calculates the *M*-element state vector of the system at time step *k*, given the state vector at time step *k*-1. *M* is the number of states of the nonlinear system.

You write and save the state transition function for your nonlinear system and use it to construct the object. For example, if `vdpStateFcn.m` is the state transition function, specify `StateTransitionFcn` as `@vdpStateFcn`. You can also specify `StateTransitionFcn` as a function handle to an anonymous function.

The inputs to the function you write depend on whether you specify the process noise as additive or nonadditive in the `HasAdditiveProcessNoise` property of the object:

- `HasAdditiveProcessNoise` is true — The process noise `w` is additive, and the state transition function specifies how the states evolve as a function of state values at previous time step:

  ```
  x(k) = f(x(k-1),Us1,...,Usn)
  ```

  Where `x(k)` is the estimated state at time `k`, and `Us1,...,Usn` are any additional input arguments required by your state transition function, such as system inputs or the sample time. During estimation, you pass these additional arguments to the `predict` command, which in turn passes them to the state transition function.
- `HasAdditiveProcessNoise` is false — The process noise is nonadditive, and the state transition function also specifies how the states evolve as a function of the process noise:

  ```
  x(k) = f(x(k-1),w(k-1),Us1,...,Usn)
  ```

To see an example of a state transition function with additive process noise, type `edit vdpStateFcn` at the command line.

`StateTransitionFcn` is a nontunable property. You can specify it once before using the `predict` command either during object construction or using dot notation after object construction. You cannot change it after using the `predict` command.

`StateTransitionJacobianFcn` — Jacobian of state transition function

[ ] (default) | function handle

Jacobian of state transition function *f*, specified as one of the following:

- `[]` — The Jacobian is numerically computed at every call to the `predict` command. This may increase processing time and numerical inaccuracy of the state estimation.
- function handle — You write and save the Jacobian function and specify the handle to the function. For example, if `vdpStateJacobianFcn.m` is the Jacobian function, specify `StateTransitionJacobianFcn` as `@vdpStateJacobianFcn`.

  The function calculates the partial derivatives of the state transition function with respect to the states and process noise. The number of inputs to the Jacobian function must equal the number of inputs of the measurement function and must be specified in the same order in both functions. The number of outputs of the function depends on the `HasAdditiveProcessNoise` property:

  - `HasAdditiveProcessNoise` is true — The function calculates the partial derivative of the state transition function with respect to the states ($\partial f / \partial x$). The output is an M-by-M Jacobian matrix, where M is the number of states.
  - `HasAdditiveProcessNoise` is false — The function must also return a second output that is the partial derivative of the state transition function with respect to the measurement noise terms ($\partial f / \partial w$). The second output is returned as an M-by-W Jacobian matrix, where W is the number of process noise terms.

The extended Kalman filter algorithm uses the Jacobian to compute the state estimation error covariance.

To see an example of a Jacobian function for additive process noise, type `edit vdpStateJacobianFcn` at the command line.

`StateTransitionJacobianFcn` is a nontunable property. You can specify it once before using the `predict` command either during object construction or using dot notation after object construction. You cannot change it after using the `predict` command.

Output Arguments

`obj` — Extended Kalman filter object for online state estimation

`extendedKalmanFilter` object

Extended Kalman filter object for online state estimation, returned as an `extendedKalmanFilter` object. This object is created using the specified properties. Use the `correct` and `predict` commands to estimate the state and state estimation error covariance using the extended Kalman filter algorithm.

When you use `predict`, `obj.State` and `obj.StateCovariance` are updated with the predicted value at time step *k* using the state value at time step *k*–1. When you use `correct`, `obj.State` and `obj.StateCovariance` are updated with the estimated values at time step *k* using measured data at time step *k*.

More About

- "What Is Online Estimation?"
- "Extended and Unscented Kalman Filter Algorithms for Online State Estimation"
- "Validate Online State Estimation at the Command Line"
- "Troubleshoot Online State Estimation"

See Also

Functions

`clone` | `correct` | `predict` | `unscentedKalmanFilter`

Blocks

`Kalman Filter`

Introduced in R2016b

evaluate

Value of nonlinearity estimator at given input

Syntax

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

Arguments

`nl`

Nonlinearity estimator object.

`x`

Value at which to evaluate the nonlinearity.

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

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

Description

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

Examples

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

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

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

See Also

`idnlarx` | `idnlhw`

Introduced in R2007a

fcat

Concatenate FRD models along frequency dimension

Syntax

```
sys = fcat(sys1,sys2,...)
```

Description

`sys = fcat(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

`idfrd` | `fdel` | `fselect` | `interp` | `frd`

Introduced before R2006a

fdel

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`.

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.

More About

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 `fcat`. `fcat` 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`.

See Also

`idfrd` | `fcat` | `fselect` | `frd`

Introduced in R2012a

feedback

Identify possible feedback data

Syntax

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

Description

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

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

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

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

See Also

`advice` | `iddata`

Introduced before R2006a

fft

Transform `iddata` object to frequency domain data

Syntax

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

Description

`Datf = fft(Data)` transforms time-domain data to frequency domain data. If `Data` is a time-domain `iddata` object with real-valued signals and with constant sample time `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.

`Datf = fft(Data,N)` specifies the transformation length. 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.

`Datf = fft(Data,N,'complex')` specifies to include negative frequencies. 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`

Introduced in R2007a

idnlarx/findop

Compute operating point for Nonlinear ARX model

Syntax

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

`[X,U] = findop(sys,spec)`

`[X,U] = findop( ___ ,Options)`

`[X,U,Report] = findop( ___ )`

```
[X,U] = findop(sys,'snapshot',T,Uin)
[X,U] = findop(sys,'snapshot',T,Uin,X0)
```

Description

`[X,U] = findop(sys,'steady',InputLevel,OutputLevel)` returns the operating-point state values, `X`, and input values, `U`, for the`idnlarx` model, `sys`, using steady-state input and output specifications.

`[X,U] = findop(sys,spec)` returns the steady-state operating point for `sys` using the operating-point specification, `spec`.

`[X,U] = findop( ___ ,Options)` specifies optimization search options for all of the previous syntaxes.

`[X,U,Report] = findop( ___ )` returns a summary report on the optimization search results for all of the previous syntaxes.

`[X,U] = findop(sys,'snapshot',T,Uin)` returns the operating point for `sys` at a simulation snapshot at time, `T`, using the specified input, `Uin`. The initial states of `sys` are assumed to be zero.

`[X,U] = findop(sys,'snapshot',T,Uin,X0)` specifies the initial states of the simulation.

Examples

Find Steady-State Nonlinear ARX Operating Point Using Default Specifications

Estimate a nonlinear ARX model.

```
load iddata6;
M = nlarx(z6,[4 3 1]);
```

Find the steady-state operating point where the input level is fixed to `1` and the output is unknown.

```
[X,U] = findop(M,'steady',1,NaN);
```

Find Nonlinear ARX Operating Point Using Additional Specifications

Estimate a nonlinear ARX model.

```
load iddata7;
M = nlarx(z7,[4 3*ones(1,2) 2*ones(1,2)]);
```

Create a default operating point specification object.

```
spec = operspec(M);
```

Set the values for the input signals.

```
spec.Input.Value(1) = -1;
spec.Input.Value(2) = 1;
```

Set the maximum and minimum values for the output signal.

```
spec.Output.Max = 10;
spec.Output.Min = -10;
```

Find the steady-state operating point using the given specifications.

```
[X,U] = findop(M,spec);
```

Find Nonlinear ARX Operating Point Using Custom Options

Estimate a nonlinear ARX model.

```
load iddata6;
M = nlarx(z6,[4 3 2]);
```

Create a default `findopOptions` option set.

```
opt = findopOptions(M);
```

Modify the option set to specify a steepest descent gradient search method with a maximum of 50 iterations.

```
opt.SearchMethod = 'grad';
opt.SearchOption.MaxIter = 50;
```

Find the steady-state operating point using the specified options.

```
[X,U] = findop(M,'steady',1,1,opt);
```

Retrieve Nonlinear ARX Operating Point Search Report

Estimate a nonlinear ARX model.

```
load iddata7;
M = nlarx(z7,[4 3*ones(1,2) 2*ones(1,2)]);
```

Find the steady-state operating point where input 1 is set to `1` and input 2 is unrestricted. The initial guess for the output value is `2`.

```
[X,U,R] = findop(M,'steady',[1 NaN],2);
```

Display the summary report.

```
disp(R);
```

```
            SearchMethod: 'auto'
                 WhyStop: 'Near (local) minimum, (norm(g) < tol)'
              Iterations: 20
               FinalCost: 1.9722e-31
    FirstOrderOptimality: 2.4434e-16
            SignalLevels: [1×1 struct]
```

Find Nonlinear ARX Simulation Snapshot Using Default Initial States

Load the estimation data and estimate a nonlinear ARX model.

```
load twotankdata;
z = iddata(y,u,1);
M = nlarx(z,[4 3 1]);
```

Find the simulation snapshot after 10 seconds, assuming initial states of zero.

```
[X,U] = findop(M,'snapshot',10,z);
```

Find Nonlinear ARX Simulation Snapshot Using Initial State Specifications

Load the estimation data and estimate a nonlinear ARX model.

```
load twotankdata;
z = iddata(y,u,1);
M = nlarx(z,[4 3 1]);
```

Create an initial state vector. The first four states correspond to delayed output values and the final three states correspond to delayed inputs.

```
X0 = [2;2;2;2;5;5;5];
```

Find the simulation snapshot after 10 seconds using the specified initial states.

```
[X,U] = findop(M,'snapshot',10,z,X0);
```

Input Arguments

`sys` — Nonlinear ARX model

`idnlarx` object

Nonlinear ARX model, specified as an `idnlarx` object.

`InputLevel` — Steady-state input level

vector

Steady-state input level for computing the operating point, specified as a vector. The length of `InputLevel` must equal the number of inputs specified in `sys`.

The optimization algorithm assumes that finite values in `InputLevel` are fixed input values. Use `NaN` to specify unknown input signals with initial guesses of `0`. The minimum and maximum bounds for all inputs have default values of `-Inf` and `+Inf` respectively.

`OutputLevel` — Steady-state output level

vector

Steady-state output level for computing the operating point, specified as a vector. The length of `OutputLevel` must equal the number of outputs specified in `sys`.

The values in `OutputLevel` indicate initial guesses for the optimization algorithm. Use `NaN` to specify unknown output signals with initial guesses of `0`. The minimum and maximum bounds for all outputs have default values of `-Inf` and `+Inf` respectively.

`spec` — Operating-point specifications

`operspec` object

Operating-point specifications, such as minimum and maximum input/output constraints and known inputs, specified as an`operspec` object.

`T` — Operating point snapshot time

positive scalar

Operating point snapshot time, specified as a positive scalar. The value of `T` must be in the range [T_0, $N*T_s$], where N is the number of input samples, T_s is the sample time and T_0 is the input start time (`Uin.Tstart`).

`Uin` — Snapshot simulation input

`iddata` object | matrix

Snapshot simulation input, specified as one of the following:

- Time-domain `iddata` object with a sample time and input size that matches `sys`.
- Matrix with as many columns as there are input channels. If the matrix has N rows, the input data is assumed to correspond to the time vector `(1:N)*sys.Ts`.

`X0` — Initial states

column vector

Initial states of the simulation, specified as a column vector with size equal to the number of states in `sys`. `X0` provides the initial conditions at the time corresponding to the first input sample (`Uin.Start`, if `Uin` is an `iddata` object, or `sys.Ts` if `Uin` is a double matrix).

For more information about the states of an `idnlarx` model, see “Definition of idnlarx States” on page 1-528.

`Options` — Operating point search options

`findopOptions` option set

Operating point search options, specified as a `findopOptions` option set.

Output Arguments

`X` — Operating point state values

column vector

Operating point state values, returned as a column vector of length equal to the number of model states.

`U` — Operating point input values

column vector

Operating point input values, returned as a column vector of length equal to the number of inputs.

`Report` — Search result summary

structure

Search result summary report, returned as a structure with the following fields:

Field	Description
`SearchMet`	Search method used for iterative parameter estimation. See `SearchMethod` in `findopOptions` for more information.
`WhyStop`	Search algorithm termination condition.
`Iteration`	Number of estimation iterations performed.
`FinalCost`	Final value of the minimization objective function (sum of the squared errors).
`FirstOrde`	∞ -norm of the search gradient vector when the search algorithm terminates.
`SignalLev`	Structure containing the fields `Input` and `Output`, which are the operating point input and output signal levels respectively.

More About

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

To compute the steady-state operating point, call `findop` using either of the following syntaxes:

```
[X,U] = findop(sys,'steady',InputLevel,OutputLevel)
[X,U] = findop(sys,spec)
```

To compute the states, `X`, and the input, `U`, of the steady-state operating point, `findop` minimizes the norm of the error $e(t) = y(t)-f(x(t), u(t))$, where:

- f is the nonlinearity estimator.
- $u(t)$ is the input.
- $x(t)$ is the model state.
- $y(t)$ is the model output.

You can specify the search algorithm and search options using the `findopOptions` option set.

The algorithm uses the following independent variables for minimization:

- Unknown (unspecified) input signal levels
- Output signal levels

Because `idnlarx` model states are delayed samples of the input and output variables, the state values 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 1-528.

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.

See Also

`data2state` | `findopOptions` | `idnlarx` | `idnlarx/operspec` | `idnlhw/findop` | `sim`

Introduced in R2008a

idnlhw/findop

Compute operating point for Hammerstein-Wiener model

Syntax

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

`[X,U] = findop(sys,spec)`

`[X,U] = findop(___,Options)`

`[X,U,Report] = findop(___)`

```
[X,U] = findop(sys,'snapshot',T,Uin)
[X,U] = findop(sys,'snapshot',T,Uin,X0)
```

Description

`[X,U] = findop(sys,'steady',InputLevel,OutputLevel)` returns the operating-point state values, `X`, and input values, `U`, for the `idnlhw` model, `sys`, using steady-state input and output specifications.

`[X,U] = findop(sys,spec)` returns the steady-state operating point for `sys` using the operating point specification in `spec`.

`[X,U] = findop(___,Options)` specifies optimization search options for all of the previous syntaxes.

`[X,U,Report] = findop(___)` returns a summary report on the optimization search results for all of the previous syntaxes.

`[X,U] = findop(sys,'snapshot',T,Uin)` returns the operating point for `sys` at a simulation snapshot at time, `T`, using the specified input, `Uin`. The initial states of `sys` are assumed to be zero.

`[X,U] = findop(sys,'snapshot',T,Uin,X0)` specifies the initial states of the simulation.

Examples

Find Steady-State Hammerstein-Wiener Operating Point Using Default Specifications

Load the estimation data and estimate a Hammerstein-Wiener model.

```
load twotankdata;
z = iddata(y,u,1);
M = nlhw(z,[5 1 3]);
```

Find the steady-state operating point where the input level is set to 1 and the output is unknown.

```
[X,U] = findop(M,'steady',1,NaN);
```

Find Hammerstein-Wiener Operating Point Using Additional Specifications

Estimate a Hammerstein-Wiener model.

```
load iddata7;
orders = [4*ones(1,2) 2*ones(1,2) 3*ones(1,2)];
M = nlhw(z7,orders,'unitgain','pwlinear');
```

Create a default operating point specification object.

```
spec = operspec(M);
```

Set the values for the input signals.

```
spec.Input.Value(1) = -1;
spec.Input.Value(2) = 1;
```

Set the maximum and minimum values for the output signal.

```
spec.Output.Max = 10;
spec.Output.Min = -10;
```

Find the steady-state operating point using the given specifications.

```
[X,U] = findop(M,spec);
```

Find Hammerstein-Wiener Operating Point Using Custom Options

Load the estimation data and estimate a Hammerstein-Wiener model.

```
load twotankdata;
z = iddata(y,u,1);
M = nlhw(z,[5 1 3]);
```

Create a default `findopOptions` option set.

```
opt = findopOptions(M);
```

Modify the option set to specify a steepest descent gradient search method with a maximum of 50 iterations.

```
opt.SearchMethod = 'grad';
opt.SearchOption.MaxIter = 50;
```

Find the steady-state operating point using the specified options.

```
[X,U] = findop(M,'steady',1,NaN,opt);
```

Retrieve Hammerstein-Wiener Operating Point Search Report

Load the estimation data and estimate a Hammerstein-Wiener model.

```
load iddata7;
orders = [4*ones(1,2) 2*ones(1,2) 3*ones(1,2)];
M = nlhw(z7,orders,'unitgain','pwlinear');
```

Find the steady-state operating point where input 1 is set to `1` and input 2 is unrestricted. The initial guess for the output value is `2`.

```
[X,U,R] = findop(M,'steady',[1 NaN],2);
```

Display the summary report.

```
disp(R);
```

```
            SearchMethod: 'auto'
                 WhyStop: 'Near (local) minimum, (norm(g) < tol)'
              Iterations: 3
               FinalCost: 1.9722e-31
    FirstOrderOptimality: 1.6481e-16
            SignalLevels: [1×1 struct]
```

Find Hammerstein-Wiener Simulation Snapshot Using Default Initial States

Load the estimation data estimate a Hammerstein-Wiener model.

```
load twotankdata;
z = iddata(y,u,1);
M = nlhw(z,[5 1 3]);
```

Find the simulation snapshot after 10 seconds, assuming initial states of zero.

```
[X,U] = findop(M,'snapshot',10,z);
```

Find Hammerstein-Wiener Simulation Snapshot Using Initial State Specifications

Load the estimation data and estimate a Hammerstein-Wiener model.

```
load twotankdata
z = iddata(y,u,1);
M = nlhw(z,[5 1 3]);
```

Create an initial state vector.

```
X0 = [10;10;5;5;1;1;0];
```

Find the simulation snapshot after 10 seconds using the specified initial states.

```
[X,U] = findop(M,'snapshot',10,z,X0);
```

Input Arguments

`sys` — Hammerstein-Wiener model

`idnlhw` object

Hammerstein-Wiener model, specified as an `idnlhw` object.

`InputLevel` — Steady-state input level

vector

Steady-state input level for computing the operating point, specified as a vector. The length of `InputLevel` must equal the number of inputs specified in `sys`.

The optimization algorithm assumes that finite values in `InputLevel` are fixed input values. Use `NaN` to specify unknown input signals with initial guesses of `0`. The minimum and maximum bounds for all inputs have default values of `-Inf` and `+Inf` respectively.

`OutputLevel` — Steady-state output level

vector

Steady-state output level for computing the operating point, specified as a vector. The length of `OutputLevel` must equal the number of outputs specified in `sys`.

The values in `OutputLevel` indicate initial guesses for the optimization algorithm. Use `NaN` to specify unknown output signals with initial guesses of `0`. The minimum and maximum bounds for all outputs have default values of `-Inf` and `+Inf` respectively.

`spec` — Operating-point specifications

`operspec` object

Operating-point specifications, such as minimum and maximum input/output constraints and known inputs, specified as an `operspec` object.

`T` — Operating point snapshot time

positive scalar

Operating point snapshot time, specified as a positive scalar. The value of `T` must be in the range $[T_0, N*T_s]$, where N is the number of input samples, T_s is the sample time and T_0 is the input start time (`Uin.Tstart`).

`Uin` — Snapshot simulation input

`iddata` object | matrix

Snapshot simulation input, specified as one of the following:

- Time-domain `iddata` object with a sample time and input size that matches `sys`.
- Matrix with as many columns as there are input channels. If the matrix has N rows, the input data is assumed to correspond to the time vector `(1:N)*sys.Ts`.

`X0` — Initial states

column vector

Initial states of the simulation, specified as a column vector with length equal to the number of states in `sys`. `X0` provides the initial conditions at the time corresponding to the first input sample (`Uin.Start`, if `Uin` is an `iddata` object, or `sys.Ts` if `Uin` is a double matrix).

For more information about the states of an `idnlhw` model, see “Definition of `idnlhw` States” on page 1-567.

`Options` — Operating point search options

`findopOptions` option set

Operating point search options, specified as a `findopOptions` option set.

Output Arguments

`X` — Operating point state values

column vector

Operating point state values, returned as a column vector of length equal to the number of model states.

`U` — Operating point input values

column vector

Operating point input values, returned as a column vector of length equal to the number of inputs.

`Report` — Search result summary

structure

Search result summary report, returned as a structure with the following fields:

Field	Description
`SearchMet`	Search method used for iterative parameter estimation. See `SearchMethod` in `findopOptions` for more information.
`WhyStop`	Search algorithm termination condition.
`Iteration`	Number of estimation iterations performed.
`FinalCost`	Final value of the minimization objective function (sum of the squared errors).
`FirstOrde`	∞ -norm of the search gradient vector when the search algorithm terminates.
`SignalLev`	Structure containing the fields `Input` and `Output`, which are the operating point input and output signal levels respectively.

More About

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

To compute the steady-state operating point, call `findop` using either of the following syntaxes:

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

```
[X,U] = findop(sys,spec)
```

`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 $X = \mathrm{inv}(I-A)*B*f(U)$, 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: $x = \mathrm{inv}(I-A)*B*f(U)$, where A and B are state-space parameters corresponding to the linear model $L(x,u)$. Thus, substituting $x = \mathrm{inv}(I-A)*B*f(U)$ 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] = ssdata(model.LinearModel)`.

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

See Also

`findopOptions` | `idnlarx/findop` | `idnlhw` | `idnlhw/operspec` | `sim`

Introduced in R2008a

findopOptions

Option set for `findop`

Syntax

```
opt = findopOptions(model)
opt = findopOptions(model,Name,Value)
```

Description

`opt = findopOptions(model)` creates a default option set for computing the operating point of a specified nonlinear ARX or Hammerstein-Wiener model. Use dot notation to modify this option set for your specific application. Options that you do not modify retain their default values.

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

Examples

Create Default Option Set for Operating Point Search

Create a default option set for `findop` using an `idnlarx` model

```
opt = findopOptions(idnlarx);
```

Create and Modify Default Operating Point Search Options

Create a default option set for `findop` using an `idnlhw` model.

```
opt = findopOptions(idnlhw);
```

Use dot notation to specify a subspace Gauss-Newton least squares search with a maximum of 25 iterations.

```
opt.SearchMethod = 'gn';
```

```
opt.SearchOption.MaxIter = 25;
```

Specify Options for Operating Point Search

Create an option set for `findop` using an `idnlarx` model. Specify a steepest descent least squares search with default search options.

```
opt = findopOptions(idnlarx,'SearchMethod','grad');
```

Input Arguments

`model` — Estimated nonlinear model

`idnlarx` model | `idnlhw` model

Estimated nonlinear model, specified as one of the following:

- `idnlarx` model
- `idnlhw` model

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example: `'SearchMethod','grad'` specifies a steepest descent least squares search method

`'SearchMethod'` — Numerical search method used for iterative parameter estimation

`'auto'` (default) | `'gn'` | `'gna'` | `'lm'` | `'grad'` | `'lsqnonlin'`

Numerical search method used for iterative parameter estimation, specified as the comma-separated pair consisting of `'SearchMethod'` and one of the following:

- `'auto'` — A combination of the line search algorithms, `'gn'`, `'lm'`, `'gna'`, and `'grad'` methods is tried at each iteration. The descent direction leading to the largest reduction in estimation cost is used.
- `'gn'` — Subspace Gauss-Newton least squares search. Singular values of the Jacobian matrix less than `GnPinvConst*eps*max(size(J))*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'` — Adaptive subspace Gauss-Newton search. Eigenvalues less than `gamma*max(sv)` of the Hessian are ignored, 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 five bisections. This value is decreased by the factor `2*LMStep` each time a search is successful without any bisections.
- `'lm'` — Levenberg-Marquardt least squares search, where the next parameter value is `-pinv(H+d*I)*grad` from the previous one. *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.
- `'grad'` — Steepest descent least squares search.
- `'lsqnonlin'` — Trust region reflective algorithm provided by Optimization Toolbox. See `lsqnonlin` for more information.

`'SearchOption'` — Option set for the search algorithm

search option set

Option set for the search algorithm, specified as the comma-separated pair consisting of `'SearchOption'` and a search option set with fields that depend on the value of `SearchMethod`.

`SearchOption` Structure When `SearchMethod` Is Specified as `'gn'`, `'gna'`, `'lm'`, `'grad'`, or `'auto'`

Field Name	Description	Default
`Toleran`	Minimum percentage difference between the current value of the loss function and its expected improvement after the next iteration, specified as a positive scalar. When the percentage of expected improvement is less than `Tolerance`, the iterations stop. The estimate of the expected loss-function improvement at the next iteration is based on the Gauss-Newton vector computed for the current parameter value.	`0.01`
`MaxIter`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations	`20`

<table>
<tr><th>Field Name</th><th colspan="2">Description</th><th>Default</th></tr>
<tr><td></td><td colspan="2">stop when `MaxIter` is reached or another stopping criterion is satisfied, such as `Tolerance`.

Setting `MaxIter = 0` returns the result of the start-up procedure.

Use `sys.Report.Termination.Iterations` to get the actual number of iterations during an estimation, where *sys* is an `idtf` model.</td><td></td></tr>
<tr><td>`Advance`</td><td colspan="3">Advanced search settings, specified as a structure with the following fields:
<table>
<tr><th>Field Name</th><th>Description</th><th>Default</th></tr>
<tr><td>`GnPinvCons`</td><td>Jacobian matrix singular value threshold, specified as a positive scalar. Singular values of the Jacobian matrix that are smaller than `GnPinvConst*max(size(J)*norm(J)*eps)` are discarded when computing the search direction. Applicable when `SearchMethod` is `'gn'`.</td><td>`10000`</td></tr>
<tr><td>`InitGnaTol`</td><td>Initial value of *gamma*, specified as a positive scalar. Applicable when `SearchMethod` is `'gna'`.</td><td>`0.0001`</td></tr>
<tr><td>`LMStartVal`</td><td>Starting value of search-direction length *d* in the Levenberg-Marquardt method, specified as a positive scalar. Applicable when `SearchMethod` is `'lm'`.</td><td>`0.001`</td></tr>
<tr><td>`LMStep`</td><td>Size of the Levenberg-Marquardt step, specified as a positive integer. 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'`.</td><td>`2`</td></tr>
<tr><td>`MaxBisecti`</td><td>Maximum number of bisections used for line search along the search direction, specified as a positive integer.</td><td>`25`</td></tr>
<tr><td>`MaxFunEval`</td><td>Maximum number of calls to the model file, specified as a positive integer. Iterations stop if</td><td>`Inf`</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Field Name</th><th>Description</th><th>Default</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field Name</th><th>Description</th><th>Default</th></tr>
<tr><td></td><td>the number of calls to the model file exceeds this value.</td><td></td></tr>
<tr><td><code>MinParChan</code></td><td>Smallest parameter update allowed per iteration, specified as a nonnegative scalar.</td><td><code>0</code></td></tr>
<tr><td><code>RelImprove</code></td><td>Relative improvement threshold, specified as a nonnegative scalar. Iterations stop if the relative improvement of the criterion function is less than this value.</td><td><code>0</code></td></tr>
<tr><td><code>StepReduct</code></td><td>Step reduction factor, specified as a positive scalar that is greater than 1. The suggested parameter update is reduced by the factor <code>StepReduction</code> after each try. This reduction continues until <code>MaxBisections</code> tries are completed or a lower value of the criterion function is obtained.
<code>StepReduction</code> is not applicable for <code>SearchMethod</code> <code>'lm'</code> (Levenberg-Marquardt method).</td><td><code>2</code></td></tr>
</table>
</td><td></td></tr>
</table>

`SearchOption` Structure When `SearchMethod` Is Specified as `'lsqnonlin'`

Field Name	Description	Default
`TolFun`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar. The value of `TolFun` is the same as that of `opt.SearchOption.Advanced.TolFun`.	`1e-5`
`TolX`	Termination tolerance on the estimated parameter values, specified as a positive scalar. The value of `TolX` is the same as that of `opt.SearchOption.Advanced.TolX`.	`1e-6`

Field Name	Description	Default
`MaxIter`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIter` is reached or another stopping criterion is satisfied, such as `TolFun`. The value of `MaxIter` is the same as that of `opt.SearchOption.Advanced.MaxIter`.	`20`
`Advance`	Advanced search settings, specified as an option set for `lsqnonlin`. For more information, see the Optimization Options table in “Optimization Options”.	Use `optimset('lsqnonlin')` to create a default option set.

To specify field values in `SearchOption`, create a default `findopOptions` set and modify the fields using dot notation. Any fields that you do not modify retain their default values.

```
opt = findopOptions;
opt.SearchOption.MaxIter = 15;
opt.SearchOption.Advanced.RelImprovement = 0.5;
```

Output Arguments

`opt` — Option set for `findop` command

`findopOptions` object

Option set for `findop` command, returned as a `findopOptions` object.

See Also

`idnlarx/findop` | `idnlhw/findop`

Introduced in R2015a

findstates

Estimate initial states of model

Syntax

```
x0 = findstates(sys,Data)
x0 = findstates(sys,Data,Horizon)
x0 = findstates(sys,Data,Horizon,Options)

[x0,Report]= findstates( ___ )
```

Description

`x0 = findstates(sys,Data)` estimates the initial states, `x0`, of an identified model `sys`, to maximize the fit between the model response and the output signal in the estimation data.

`x0 = findstates(sys,Data,Horizon)` specifies the prediction horizon for computing the response of `sys`.

`x0 = findstates(sys,Data,Horizon,Options)` specifies additional options for computation of `x0`.

`[x0,Report]= findstates( ___ )` delivers a report on the initial state estimation. `Report` is returned with any of the previous syntaxes.

Examples

Estimate Initial States of a Model

Create a nonlinear grey-box model.

```
FileName = 'dcmotor_m';
Order = [2 1 2];
```

```
Parameters = [0.24365;0.24964];
nlgr = idnlgrey(FileName,Order,Parameters);
```

The model is a linear DC motor with one input (voltage), and two outputs (angular position and angular velocity). The structure of the model is specified by dcmotor_m.m file.

Load the estimation data.

```
load(fullfile(matlabroot,'toolbox','ident',...
     'iddemos','data','dcmotordata'));
z = iddata(y,u,0.1);
```

Estimate the initial states.

```
X0 = findstates(nlgr,z,Inf);
```

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.

Load sample data.

```
load iddata1 z1;
```

Estimate a linear model from the data.

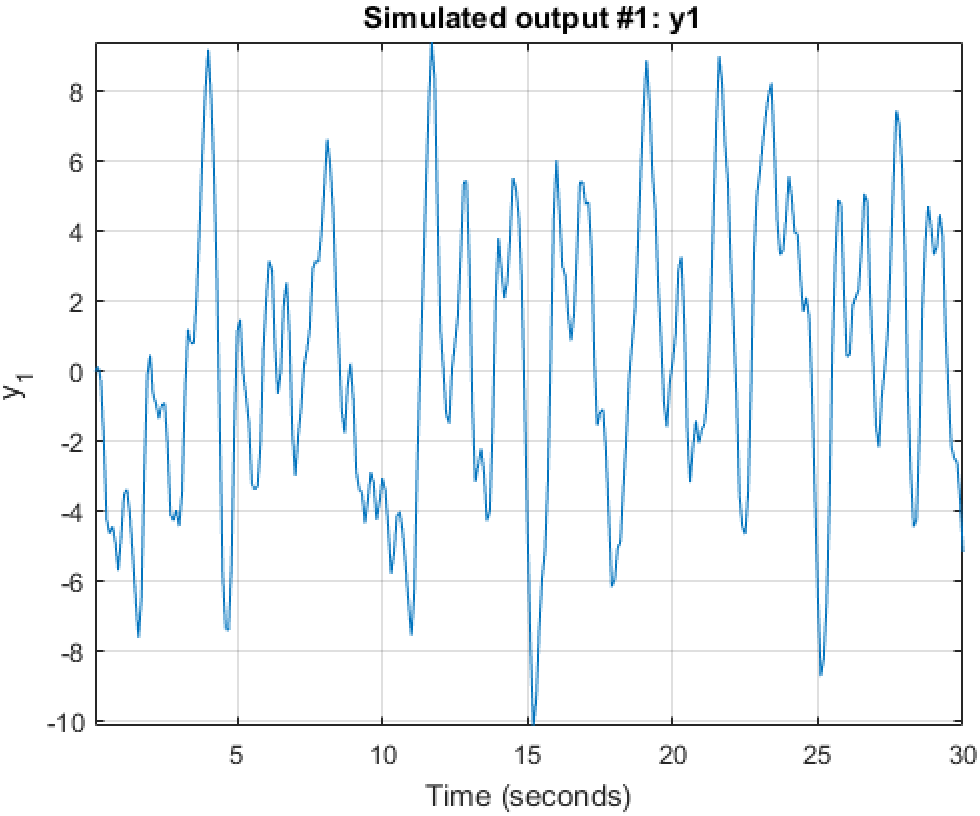
```
model = ssest(z1,2);
```

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(:,[],:),opt);
```



Selectively Estimate Initial States of a Model

Estimate the initial states of a model selectively by fixing the first state and allowing the second state of the model to be estimated.

Create a nonlinear grey-box model.

```
FileName = 'dcmotor_m';
Order = [2 1 2];
Parameters = [0.24365;0.24964];
nlgr = idnlgrey(FileName,Order,Parameters);
```

The model is a linear DC motor with one input (voltage), and two outputs (angular position and angular velocity). The structure of the model is specified by `dcmotor_m.m` file.

Load the estimation data.

```
load(fullfile(matlabroot,'toolbox','ident',...
     'iddemos','data','dcmotordata'));
z = iddata(y,u,0.1);
```

Hold the first state fixed at zero, and estimate the value of the second.

```
x0spec = idpar('x0',[0;0]);
x0spec.Free(1) = false;
opt = findstatesOptions;
opt.InitialState = x0spec;
[X0,Report] = findstates(nlgr,z,Inf,opt)
```

```
X0 =

         0
    0.0061


Report =

         Status: 'Estimated by simulation error minimization'
         Method: 'lsqnonlin'
     Covariance: [2×2 double]
       DataUsed: [1×1 struct]
    Termination: [1×1 struct]
```

Estimate Initial States by Specifying an Initial State Vector

Create a nonlinear grey-box model.

```
FileName = 'dcmotor_m';
Order = [2 1 2];
Parameters = [0.24365;0.24964];
nlgr = idnlgrey(FileName,Order,Parameters);
```

The model is a linear DC motor with one input (voltage), and two outputs (angular position and angular velocity). The structure of the model is specified by `dcmotor_m.m` file.

Load the estimation data.

```
load(fullfile(matlabroot,'toolbox','ident',...
     'iddemos','data','dcmotordata'));
z = iddata(y,u,0.1);
```

Specify an initial guess for the initial states.

```
x0spec = idpar('x0',[10;10]);
```

`x0spec.Free` is true by default

Estimate the initial states

```
opt = findstatesOptions;
opt.InitialState = x0spec;
x0 = findstates(nlgr,z,Inf,opt)


x0 =

    0.0362
   -0.1322
```

Estimate Initial States Using Multi-Experiment Data

Create a nonlinear grey-box model.

```
FileName = 'dcmotor_m';
Order = [2 1 2];
Parameters = [0.24365;0.24964];
nlgr = idnlgrey(FileName,Order,Parameters);
set(nlgr, 'InputName','Voltage','OutputName', ...
  {'Angular position','Angular velocity'});
```

The model is a linear DC motor with one input (voltage), and two outputs (angular position and angular velocity). The structure of the model is specified by `dcmotor_m.m` file.

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'});
```

Create a three-experiment data set.

```
z3 = merge(z,z,z);
```

Choose experiment for estimating the initial states:

- Estimate initial state 1 for experiments 1 and 3
- Estimate initial state 2 for experiment 1

The fixed initial states have zero values.

```
x0spec = idpar('x0',zeros(2,3));
x0spec.Free(1,2) = false;
x0spec.Free(2,[2 3]) = false;
opt = findstatesOptions;
opt.InitialState = x0spec;
```

Estimate the initial states

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

Input Arguments

`sys` — Identified model

`idss` object | `idgrey` object | `idnlarx` object | `idnlhw` object | `idnlgrey` object

Identified model whose initial states are estimated, represented as a linear state-space (`idss` or `idgrey`) or nonlinear model (`idnlarx`, `idnlhw`, or `idnlgrey`).

`Data` — Estimation data

`iddata` object

Estimation data, specified as an `iddata` object with input/output dimensions that match `sys`.

If `sys` is a linear model, `Data` can be a frequency-domain `iddata` object. For easier interpretation of initial conditions, make the frequency vector of `Data` be symmetric about the origin. For converting time-domain data into frequency-domain data, use `fft` with `'compl'` input argument, and ensure that there is sufficient zero padding. Scale your data appropriately when you compare `x0` between the time-domain and frequency-domain. Since for an N-point fft, the input/output signals are scaled by `1/sqrt(N)`, the estimated `x0` vector is also scaled by this factor.

`Horizon` — Prediction horizon for computing model response

`1` (default) | positive integer between `1` and `Inf`

Prediction horizon for computing the response of `sys`, specified as a positive integer between `1` and `Inf`. The most common values used are:

- `Horizon = 1` — Minimizes the 1-step prediction error. The 1–step ahead prediction response of `sys` is compared to the output signals in `Data` to determine `x0`. See `predict` for more information.
- `Horizon = Inf` — Minimizes the simulation error. The difference between measured output, `Data.y`, and simulated response of `sys` to the measured input data, `Data.u` is minimized. See `sim` for more information.

Specify `Horizon` as any positive integer between 1 and `Inf`, with the following restrictions:

Scenario	`Horizon`
Continuous-time model with time-domain data	`1` or `Inf`
Continuous-time frequency-domain data (`data.Ts` = 0)	`Inf`
Output Error models (trivial noise component): • Nonlinear grey-box (`idnlgrey`) • Hammerstein-Wiener (`idnlhw`) • Linear state-space with disturbance matrix, `K` = `0`	Irrelevant — Any value of `Horizon` returns the same answer for `x0`
Nonlinear ARX (`idnlarx`)	`1` or `Inf`

`Options` — Estimation options for `findstates`

`findstates` Option set

Estimation options for `findstates`, specified as an option set created using `findstatesOptions`

Output Arguments

`x0` — Estimated initial states

vector | matrix

Estimated initial states of model `sys`, returned as a vector or matrix. For multi-experiment data, `x0` is a matrix with one column for each experiment.

`Report` — Initial state estimation information

structure

Initial state estimation information, returned as a structure. `Report` contains information about the data used, state covariance, and results of any numerical optimization performed to search for the initial states. `Report` has the following fields:

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Status</code></td><td>Summary of how the initial state were estimated.</td></tr>
<tr><td><code>Method</code></td><td>Search method used.</td></tr>
<tr><td><code>Covarianc</code></td><td>Covariance of state estimates, returned as a <i>Ns</i>-by-<i>Ns</i> matrix, where <i>Ns</i> is the number of states.</td></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation, returned as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
<tr><td><code>Type</code></td><td>Data type.</td></tr>
<tr><td><code>Length</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time.</td></tr>
<tr><td><code>InterSam</code></td><td>Input intersample behavior, returned as one of the following values:

<code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
<code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
<code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.
</td></tr>
<tr><td><code>InputOff</code></td><td>Offset removed from time-domain input data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
<tr><td><code>OutputOf</code></td><td>Offset removed from time-domain output data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Terminati</code></td><td>Termination conditions for the iterative search used for initial state estimation of nonlinear models. Structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>WhyStop</code></td><td>Reason for terminating the numerical search.</td></tr>
<tr><td><code>Iteratio</code></td><td>Number of search iterations performed by the estimation algorithm.</td></tr>
<tr><td><code>FirstOrd</code></td><td>∞ -norm of the gradient search vector when the search algorithm terminates.</td></tr>
<tr><td><code>FcnCount</code></td><td>Number of times the objective function was called.</td></tr>
<tr><td><code>UpdateNo</code></td><td>Norm of the gradient search vector in the last iteration. Omitted when the search method is <code>'lsqnonlin'</code>.</td></tr>
<tr><td><code>LastImpr</code></td><td>Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is <code>'lsqnonlin'</code>.</td></tr>
<tr><td><code>Algorith</code></td><td>Algorithm used by <code>'lsqnonlin'</code> search method. Omitted when other search methods are used.</td></tr>
</table>
<code>Termination</code> is empty for linear models.</td></tr>
</table>

See Also

`findstatesOptions` | `predict` | `sim`

Introduced in R2015a

findstatesOptions

Option set for `findstates`

Syntax

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

Description

`opt = findstatesOptions` creates the default option set for `findstates`. Use dot notation to customize the option set, if needed.

`opt = findstatesOptions(Name,Value)` creates an option set with options specified by one or more `Name,Value` pair arguments. The options that you do not specify retain their default value.

Examples

Identify Initial States Using Option Set

Create an option set for `findstates` by configuring a specification object for the initial states.

Identify a fourth-order state-space model from data.

```
load iddata8 z8;
sys = ssest(z8,4);
```

`z8` is an `iddata` object containing time-domain system response data. `sys` is a fourth-order `idss` model that is identified from the data.

Configure a specification object for the initial states of the model.

```
x0obj = idpar([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. 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 the model.

```
opt = findstatesOptions;
opt.InitialState = x0obj;
```

Identify the initial states of the model.

```
x0_estimated = findstates(sys,z8,Inf,opt);
```

Specify Option Set for Initial States Estimation

Create an option set for `findstates` where:

- Initial states are estimated such that the norm of prediction error is minimized. The initial values of the states corresponding to nonzero delays are also estimated.
- Adaptive subspace Gauss-Newton search is used for estimation.

```
opt = findstatesOptions('InitialState','d','SearchMethod','gna');
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example: `findstatesOptions('InitialState','d')`

`'InitialState'` — Estimation of initial states

`'e'` (default) | `'d'` | vector or matrix | `idpar` object `x0obj`

Estimation of initial states, specified as the comma-separated pair consisting of `'InitialState'` and one of the following:

- `'e'` — The initial states are estimated such that the norm of prediction error is minimized.
- `'d'` — Similar to `'e'`, but absorbs nonzero delays into the model coefficients. The delays are first converted to explicit model states, and the initial values of those states are also estimated and returned.

  Use this option for discrete-time linear models only.
- `Vector or Matrix` — Initial guess for state values, when using nonlinear models. Specify a column vector of length equal to the number of states. For multi-experiment data, use a matrix with `Ne` columns, where `Ne` is the number of experiments.

  Use this option for nonlinear models only.
- `x0obj` — Specification object created using `idpar`. Use `x0obj` to impose constraints on the initial states by fixing their value or specifying minimum or maximum bounds.

  Use `x0obj` only for nonlinear grey-box models and linear state-space models (`idss` or `idgrey`). This option is applicable only for prediction horizon equal to `1` or `Inf`. See `findstates` for more details about the prediction horizon.

`'InputOffset'` — Removal of offset from time-domain input data during estimation

[ ] (default) | vector of positive integers | matrix

Removal of offset from time-domain input data during estimation, specified as the comma-separated pair consisting of `'InputOffset'` and one of the following:

- A column vector of positive integers of length *Nu*, where *Nu* is the number of inputs.
- `[]` — Indicates no offset.
- *Nu*-by-*Ne* matrix — For multi-experiment data, specify `InputOffset` as an *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.

`'OutputOffset'` — Removal of offset from time-domain output data during estimation

[ ] (default) | vector | matrix

Removal of offset from time-domain output data during estimation, specified as the comma-separated pair consisting of `'OutputOffset'` and one of the following:

- A column vector of length *Ny*, where *Ny* is the number of outputs.
- `[]` — Indicates no offset.
- *Ny*-by-*Ne* matrix — For multi-experiment 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.

`'OutputWeight'` — Weighting of prediction errors when using multi-output data

`[]` (default) | `'noise'` | matrix

Weighting of prediction errors when using multi-output data, specified as the comma-separated pair consisting of `'OutputWeight'` and one of the following:

- `[]` — No weighting is used. Specifying as `[]` is the same as `eye(Ny)`, where *Ny* is the number of outputs.
- `'noise'` — Inverse of the noise variance stored with the model is used for weighting during estimation of initial states.
- Positive semidefinite matrix, `W`, of size *Ny*-by-*Ny* — This weighting minimizes `trace(E'*E*W)` for estimation of initial states, where `E` is the matrix of prediction errors.

`'SearchMethod'` — Numerical search method used for iterative parameter estimation

`'auto'` (default) | `'gn'` | `'gna'` | `'lm'` | `'grad'` | `'lsqnonlin'`

Numerical search method used for iterative parameter estimation, specified as the comma-separated pair consisting of `'SearchMethod'` and one of the following:

- `'auto'` — A combination of the line search algorithms, `'gn'`, `'lm'`, `'gna'`, and `'grad'` methods is tried at each iteration. The descent direction leading to the largest reduction in estimation cost is used.
- `'gn'` — Subspace Gauss-Newton least squares search. Singular values of the Jacobian matrix less than `GnPinvConst*eps*max(size(J))*norm(J)` are discarded when computing the search direction. J is the Jacobian matrix. The Hessian matrix is approximated by J^TJ. If there is no improvement in this direction, the function tries the gradient direction.
- `'gna'` — Adaptive subspace Gauss-Newton search. Eigenvalues less than `gamma*max(sv)` of the Hessian are ignored, where *sv* contains the singular values of the Hessian. The Gauss-Newton direction is computed in the remaining subspace. *gamma* has the initial value `InitGnaTol` (see `Advanced` in `'SearchOption'` 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 five bisections. This value is decreased by the factor `2*LMStep` each time a search is successful without any bisections.

- `'lm'` — Levenberg-Marquardt least squares search, where the next parameter value is `-pinv(H+d*I)*grad` from the previous one. *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.
- `'grad'` — Steepest descent least squares search.
- `'lsqnonlin'` — Trust region reflective algorithm provided by Optimization Toolbox. This method cannot be used with the `'OutputWeight'` option`'noise'`. See `lsqnonlin` for more information.

`'SearchOption'` — Option set for the search algorithm

search option set

Option set for the search algorithm, specified as the comma-separated pair consisting of `'SearchOption'` and a search option set with fields that depend on the value of `SearchMethod`.

`SearchOption` Structure When `SearchMethod` Is Specified as `'gn'`, `'gna'`, `'lm'`, `'grad'`, or `'auto'`

Field Name	Description	Default
`Toleran`	Minimum percentage difference between the current value of the loss function and its expected improvement after the next iteration, specified as a positive scalar. When the percentage of expected improvement is less than `Tolerance`, the iterations stop. The estimate of the expected loss-function improvement at the next iteration is based on the Gauss-Newton vector computed for the current parameter value.	`0.01`
`MaxIter`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIter` is reached or another stopping criterion is satisfied, such as `Tolerance`. Setting `MaxIter = 0` returns the result of the start-up procedure.	`20`

<table>
<tr><th>Field Name</th><th>Description</th><th>Default</th></tr>
<tr><td></td><td>Use `sys.Report.Termination.Iterations` to get the actual number of iterations during an estimation, where *sys* is an `idtf` model.</td><td></td></tr>
<tr><td>`Advance`</td><td colspan="2">Advanced search settings, specified as a structure with the following fields:
<table>
<tr><th>Field Name</th><th>Description</th><th>Default</th></tr>
<tr><td>`GnPinvCons`</td><td>Jacobian matrix singular value threshold, specified as a positive scalar. Singular values of the Jacobian matrix that are smaller than `GnPinvConst*max(size(J)*norm(J)*eps)` are discarded when computing the search direction. Applicable when `SearchMethod` is `'gn'`.</td><td>`10000`</td></tr>
<tr><td>`InitGnaTol`</td><td>Initial value of *gamma*, specified as a positive scalar. Applicable when `SearchMethod` is `'gna'`.</td><td>`0.0001`</td></tr>
<tr><td>`LMStartVal`</td><td>Starting value of search-direction length *d* in the Levenberg-Marquardt method, specified as a positive scalar. Applicable when `SearchMethod` is `'lm'`.</td><td>`0.001`</td></tr>
<tr><td>`LMStep`</td><td>Size of the Levenberg-Marquardt step, specified as a positive integer. 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'`.</td><td>`2`</td></tr>
<tr><td>`MaxBisecti`</td><td>Maximum number of bisections used for line search along the search direction, specified as a positive integer.</td><td>`25`</td></tr>
<tr><td>`MaxFunEval`</td><td>Maximum number of calls to the model file, specified as a positive integer. Iterations stop if the number of calls to the model file exceeds this value.</td><td>`Inf`</td></tr>
<tr><td>`MinParChan`</td><td>Smallest parameter update allowed per iteration, specified as a nonnegative scalar.</td><td>`0`</td></tr>
<tr><td>`RelImprove`</td><td>Relative improvement threshold, specified as a nonnegative scalar. Iterations stop if the relative</td><td>`0`</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Field Name</th><th colspan="2">Description</th><th>Default</th></tr>
<tr><td></td><td colspan="3">
<table>
<tr><th>Field Name</th><th>Description</th><th>Default</th></tr>
<tr><td></td><td>improvement of the criterion function is less than this value.</td><td></td></tr>
<tr><td><code>StepReduct</code></td><td>Step reduction factor, specified as a positive scalar that is greater than 1. The suggested parameter update is reduced by the factor <code>StepReduction</code> after each try. This reduction continues until <code>MaxBisections</code> tries are completed or a lower value of the criterion function is obtained.
<code>StepReduction</code> is not applicable for <code>SearchMethod</code> <code>'lm'</code> (Levenberg-Marquardt method).</td><td><code>2</code></td></tr>
</table>
</td></tr>
</table>

`SearchOption` Structure When `SearchMethod` Is Specified as `'lsqnonlin'`

Field Name	Description	Default
`TolFun`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar. The value of `TolFun` is the same as that of `opt.SearchOption.Advanced.TolFun`.	`1e-5`
`TolX`	Termination tolerance on the estimated parameter values, specified as a positive scalar. The value of `TolX` is the same as that of `opt.SearchOption.Advanced.TolX`.	`1e-6`
`MaxIter`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIter` is reached or another stopping criterion is satisfied, such as `TolFun`.	`20`

Field Name	Description	Default
	The value of `MaxIter` is the same as that of `opt.SearchOption.Advanced.MaxIter`.	
`Advance`	Advanced search settings, specified as an option set for `lsqnonlin`. For more information, see the Optimization Options table in “Optimization Options”.	Use `optimset('lsqnonlin')` to create a default option set.

To specify field values in `SearchOption`, create a default `findstatesOptions` set and modify the fields using dot notation. Any fields that you do not modify retain their default values.

```
opt = findstatesOptions;
opt.SearchOption.Tolerance = 0.02;
opt.SearchOption.Advanced.MaxBisections = 30;
```

Output Arguments

`opt` — Option set for `findstates`

`findstatesOptions` option set

Option set for `findstates`, returned as an `findstatesOptions` option set.

See Also

`findstates` | `idpar`

Introduced in R2012a

fnorm

Pointwise peak gain of FRD model

Syntax

```
fnrm = fnorm(sys)
fnrm = fnorm(sys,ntype)
```

Description

`fnrm = 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 `fnrm` is an FRD object containing the peak gain across frequencies.

`fnrm = 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`

Introduced in R2006a

forecast

Forecast identified model output

Syntax

```
yf = forecast(sys,PastData,K)
yf = forecast(sys,PastData,K,FutureInputs)

yf = forecast( ___ ,opts)

[yf,x0,sysf] = forecast( ___ )
[yf,x0,sysf,yf_sd,x,x_sd] = forecast( ___ )

forecast(sys,PastData,K, ___ )
forecast(sys,Linespec,PastData,K, ___ )
forecast(sys1,...,sysN,PastData,K, ___ )
forecast(sys1,Linespec1,...,sysN,LinespecN,PastData,K, ___ )
```

Description

`yf = forecast(sys,PastData,K)` forecasts the output of an identified time series model `sys`, `K` steps into the future using past measured data, `PastData`.

`forecast` performs prediction into the future, in a time range beyond the last instant of measured data. In contrast, the `predict` command predicts the response of an identified model over the time span of measured data. Use `predict` to determine if the predicted result matches the observed response of an estimated model. If `sys` is a good prediction model, consider using it with `forecast`.

`yf = forecast(sys,PastData,K,FutureInputs)` uses the future values of the inputs, `FutureInputs`, to forecast the response of an identified model with input channels.

`yf = forecast( ___ ,opts)` uses the option set, `opts`, to specify additional forecast options. Use `opts` with any of the previous input argument combinations.

`[yf,x0,sysf] = forecast( ___ )` also returns the estimated values for initial states, `x0`, and a forecasting model, `sysf`, and can include any of the previous input argument combinations.

`[yf,x0,sysf,yf_sd,x,x_sd] = forecast( ___ )` also returns estimated standard deviation of the output, `yf_sd`, state trajectory, `x`, and standard deviation of the trajectory, `x_sd`. Use with any of the previous input argument combinations.

`forecast(sys,PastData,K, ___ )` plots the forecasted output. Use with any of the previous input argument combinations.

To change display options, right-click the plot to access the context menu. For example, to view the estimated standard deviation of the forecasted output, select **Confidence Region** from the context menu. For more details about the menu, see “Tips” on page 1-363.

`forecast(sys,Linespec,PastData,K, ___ )` uses `Linespec` to specify the line type, marker symbol, and color.

`forecast(sys1,...,sysN,PastData,K, ___ )` plots the forecasted outputs for multiple identified models. `forecast` automatically chooses colors and line styles.

`forecast(sys1,Linespec1,...,sysN,LinespecN,PastData,K, ___ )` uses the line type, marker symbol, and color specified for each system.

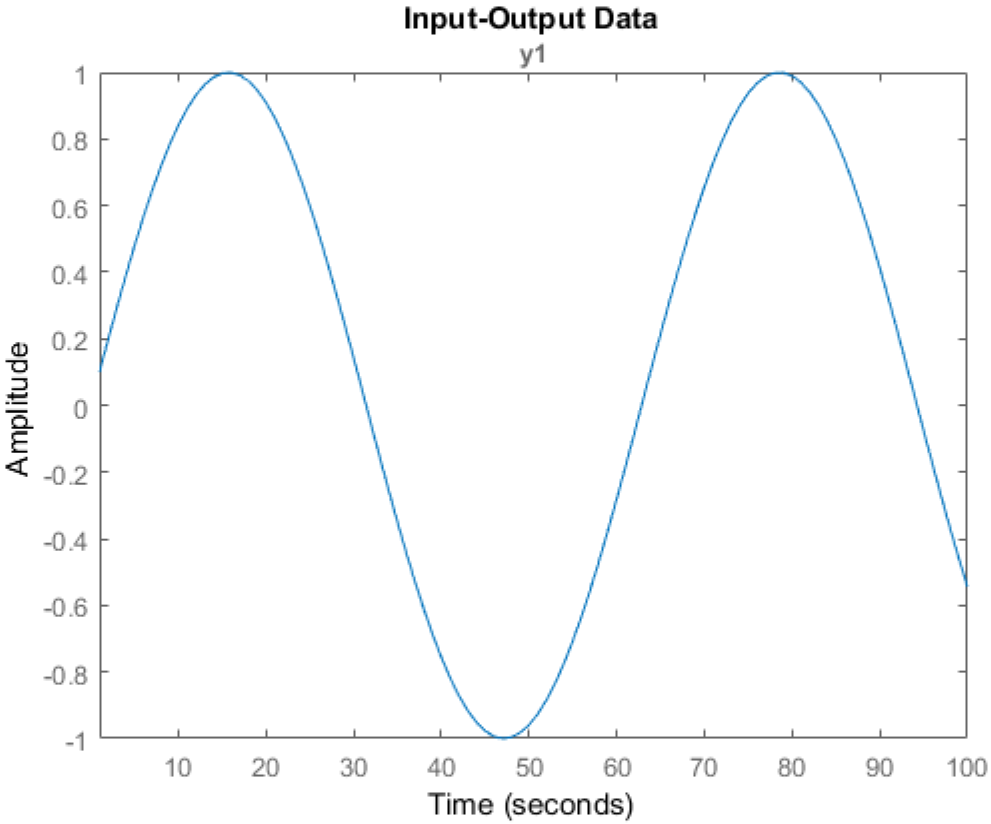
Examples

Forecast Future Values of a Sinusoidal Signal

Forecast the values of a sinusoidal signal using an AR model.

Generate and plot data.

```
data = iddata(sin(0.1*[1:100])',[]);
plot(data)
```



Fit an AR model to the sine wave.

```
sys = ar(data,2);
```

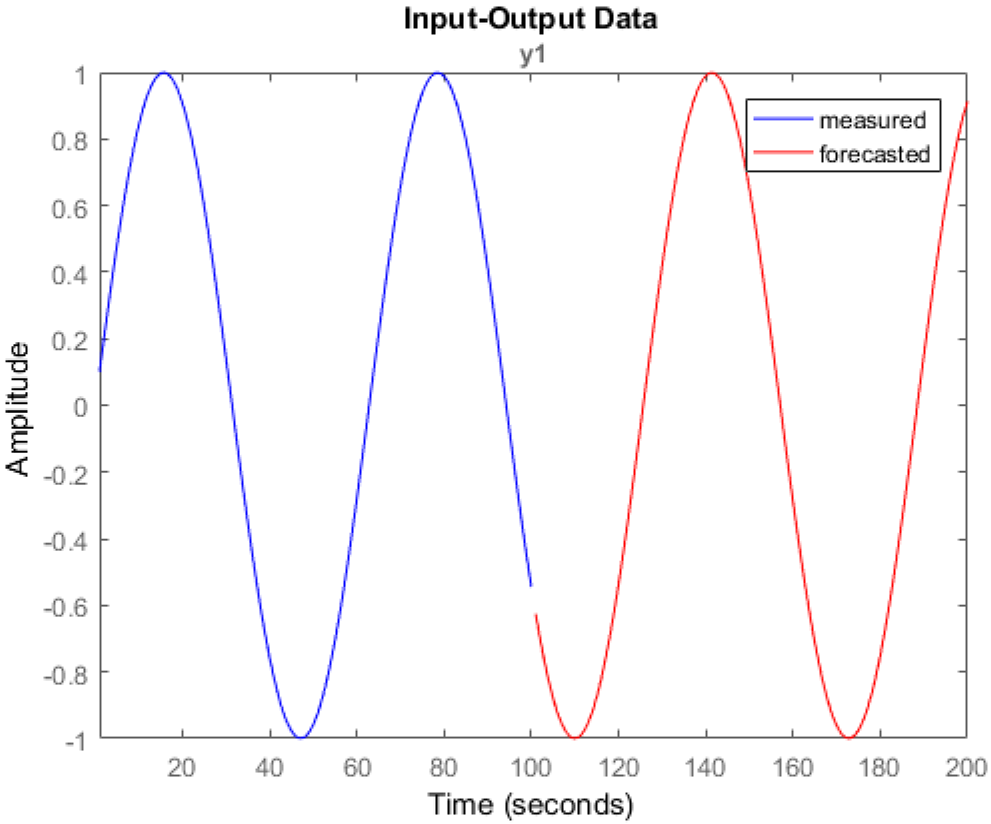
Forecast the values into the future for a given time horizon.

```
K = 100;
p = forecast(sys,data,K);
```

`K` specifies the forecasting time horizon as 100 samples. `p` is the forecasted model response.

Plot the forecasted data.

```
plot(data,'b',p,'r'), legend('measured','forecasted')
```



Alternatively, plot the forecasted output using the syntax `forecast(sys,data,K)`.

Forecast Response of Time Series Model

Obtain past data, and identify a time series model.

```
load iddata9 z9
past_data = z9.OutputData(1:50);
model = ar(z9,4);
```

`z9` is an `iddata` object that contains measured output only.

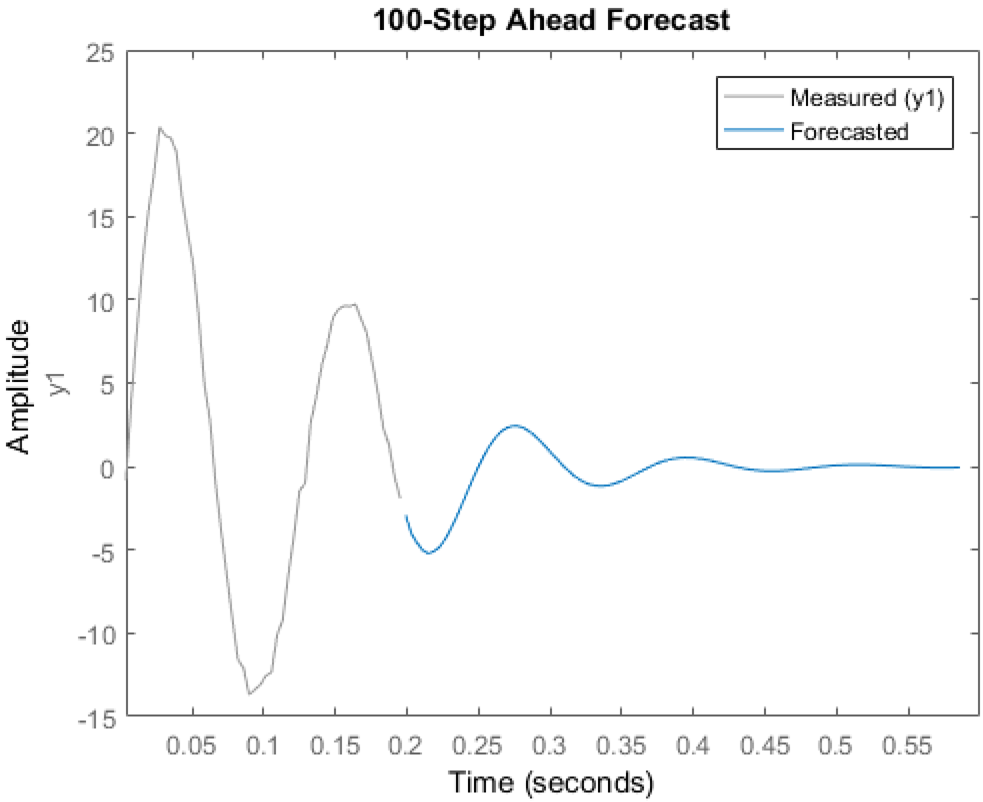
`model` is an `idpoly` time series model.

Specify initial conditions for forecasting.

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

Plot the forecasted system response for a given time horizon.

```
K = 100;
forecast(model,past_data,K,opt);
legend('Measured','Forecasted')
```



Plot Forecasted Output With Specified Line Type

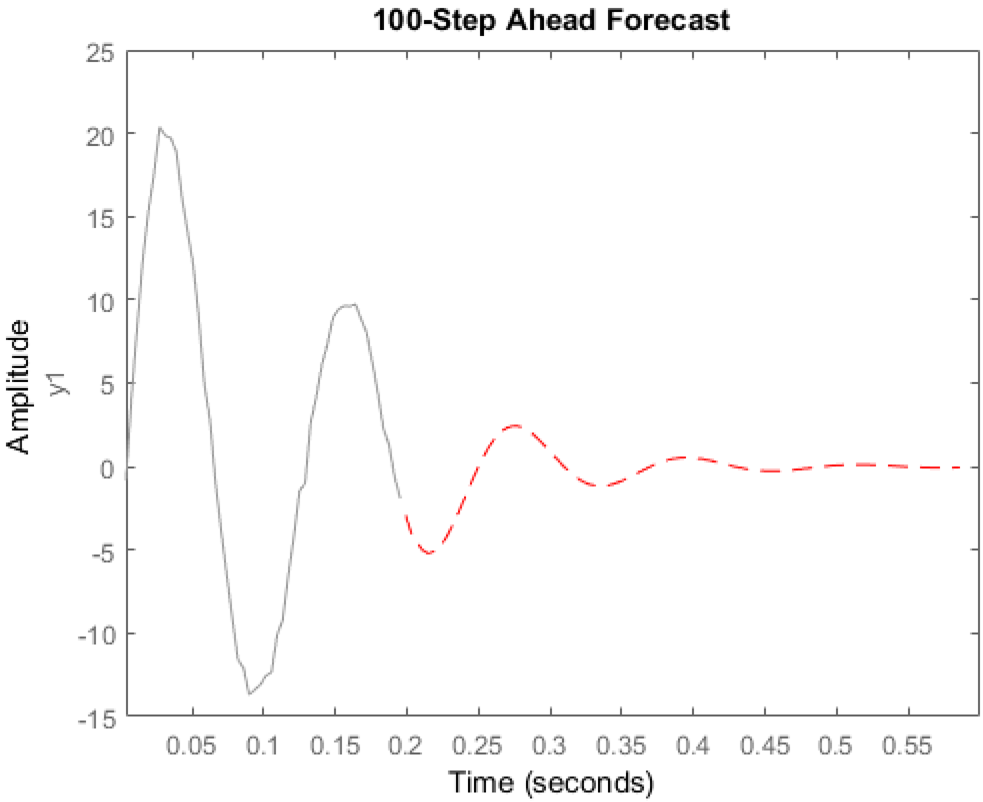
Obtain past data, and identify a time series model.

```
load iddata9 z9
past_data = z9.OutputData(1:50);
model = ar(z9,4);
```

`z9` is an `iddata` object that contains measured output only.

Plot the forecasted system response for a given time horizon as a red dashed line.

```
K = 100;
forecast(model,'r--',past_data,K);
```



The plot also displays the past data by default. To change display options, right-click the plot to access the context menu. For example, to view the estimated standard deviation of the forecasted output, select **ConfidenceRegion** from the context menu. To specify number of standard deviations to plot, double-click the plot and open the Property Editor dialog box. In the dialog box, in the **Options** tab, specify the number of standard deviations in **Confidence Region for Identified Models**. The default value is `1` standard deviation.

Forecast Model Response for Known Future Inputs

Obtain past data, future inputs, and an identified linear model.

```
load iddata1 z1
z1 = iddata(cumsum(z1.y),cumsum(z1.u),z1.Ts,'InterSample','foh');
past_data = z1(1:100);
future_inputs = z1.u(101:end);
sys = polyest(z1,[2 2 2 0 0 1],'IntegrateNoise',true);
```

`z1` is an `iddata` object that contains integrated data. `sys` is an `idpoly` model. `past_data` contains the first 100 data points of `z1`.

`future_inputs` contains the last 200 data points of `z1`.

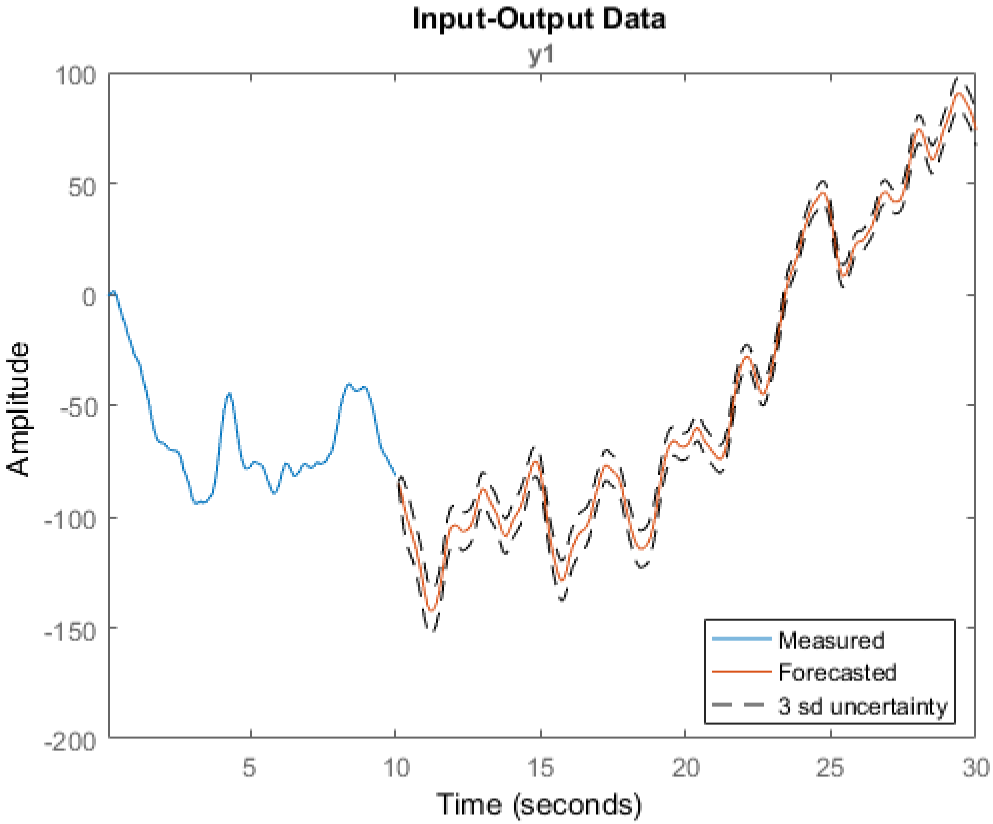
Forecast the system response into the future for a given time horizon and future inputs.

```
K = 200;
[yf,x0,sysf,yf_sd,x,x_sd] = forecast(sys,past_data,K,future_inputs);
```

`yf` is the forecasted model response, and `yf_sd` is the standard deviation of the output. `x0` is the estimated value for initial states, and `sysf` is the forecasting state-space model. Also returned are the state trajectory, `x`, and standard deviation of the trajectory, `x_sd`.

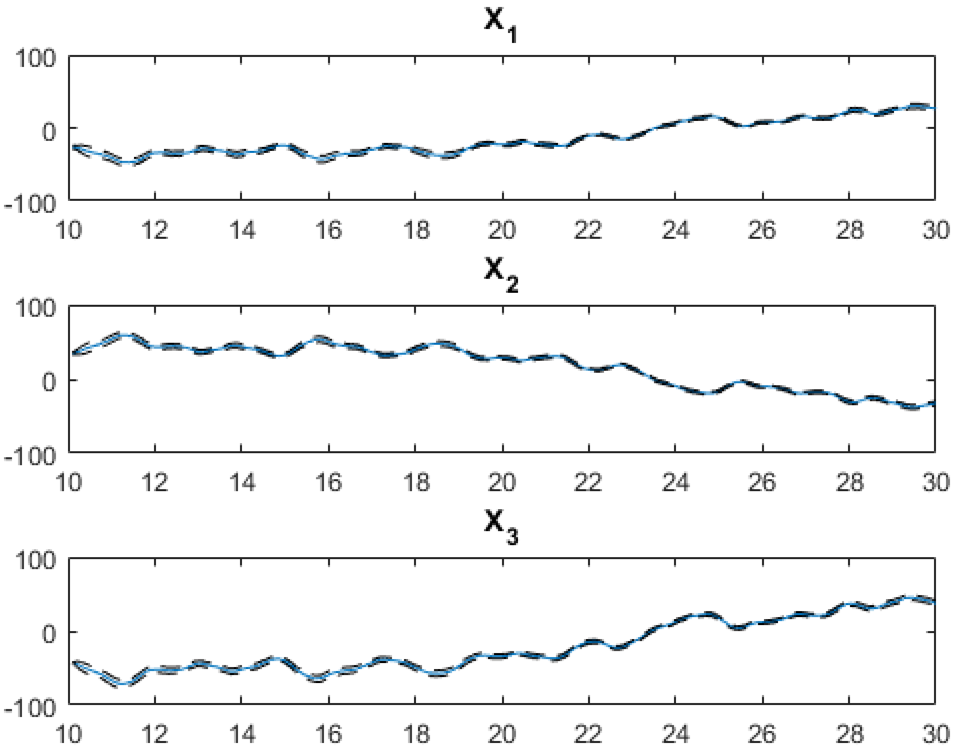
Plot the forecasted response.

```
UpperBound = iddata(yf.OutputData+3*yf_sd,[],yf.Ts,'Tstart',yf.Tstart);
LowerBound = iddata(yf.OutputData-3*yf_sd,[],yf.Ts,'Tstart',yf.Tstart);
plot(past_data(:,:,[]),yf(:,:,[]),UpperBound,'k--',LowerBound,'k--')
legend({'Measured','Forecasted','3 sd uncertainty'},'Location','best')
```



Plot the state trajectory.

```
t = z1.SamplingInstants(101:end);
subplot(3,1,1)
plot(t,x(:,1),t,x(:,1)+3*x_sd(:,1),'k--',t,x(:,1)-3*x_sd(:,1),'k--')
title('X_1')

subplot(3,1,2)
plot(t, x(:,2),t,x(:,2)+3*x_sd(:,2),'k--',t, x(:,2)-3*x_sd(:,2),'k--')
title('X_2')

subplot(3,1,3)
plot(t,x(:,3),t,x(:,3)+3*x_sd(:,3),'k--',t, x(:,3)-3*x_sd(:,3),'k--')
title('X_3')
```



The response uncertainty does not grow over the forecasting time span because of the specification of future inputs.

Forecast Response of Multi-Output Nonlinear Time Series Model

Load data.

```
load(fullfile(matlabroot,'toolbox','ident','iddemos','data','predprey2data'));
z = iddata(y,[],0.1);
set(z,'Tstart',0,'OutputUnit',{'Population (in thousands)',...
    'Population (in thousands)'},'TimeUnit','Years');
```

`z` is a two output time-series data set (no inputs) from a 1-predator 1-prey population. The population exhibits a decline in predator population due to crowding. The data set contains 201 data samples covering 20 years of evolution.

The changes in the predator (`y1`) and prey (`y2`) population can be represented as:

$$y_1(t) = p_1 * y_1(t-1) + p_2 * y_1(t-1) * y_2(t-1)$$

$$y_2(t) = p_3 * y_2(t-1) - p_4 * y_1(t-1) * y_2(t-1) - p_5 * y_2(t-1)^2$$

The nonlinearity in the predator and prey populations can be fit using a nonlinear ARX model with custom regressors.

Use part of the data as past data.

```
past_data = z(1:100);
```

Specify the standard regressors.

```
na = [1 0; 0 1];
nb = [];
nk = [];
```

Specify the custom regressors.

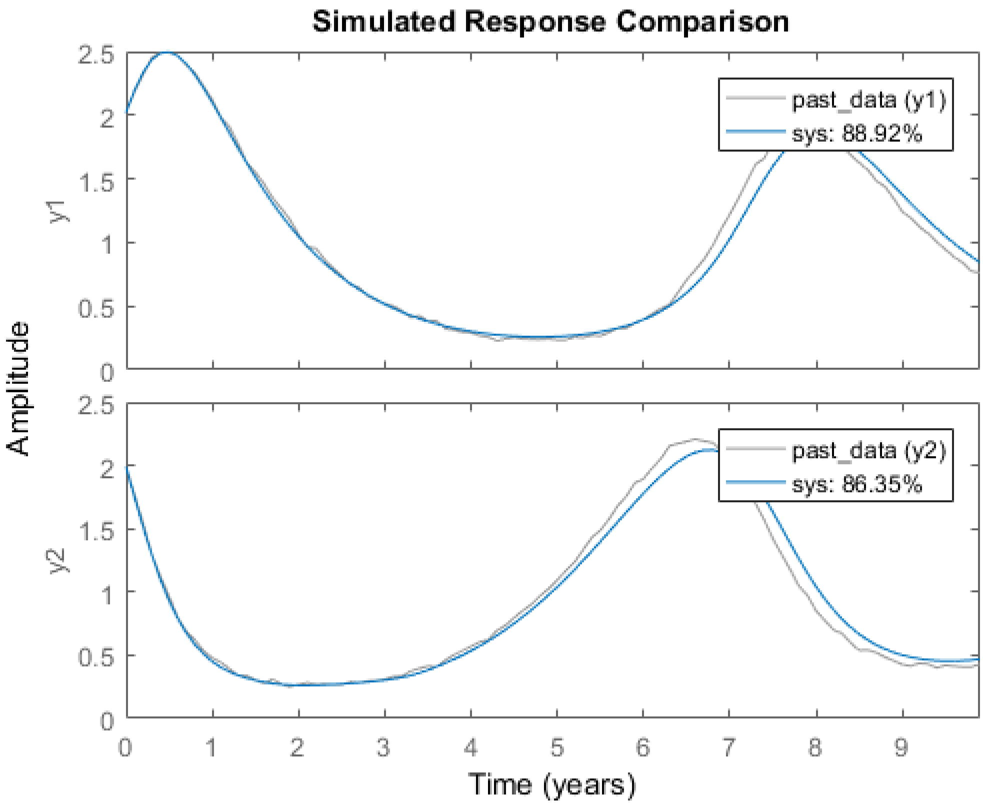
```
C = {{'y1(t-1)*y2(t-1)'};{'y1(t-1)*y2(t-1)','y2(t-1)^2'}};
```

Estimate a nonlinear ARX model using `past_data` as estimation data.

```
sys = nlarx(past_data,[na nb nk],'wavenet','CustomRegressors',C);
```

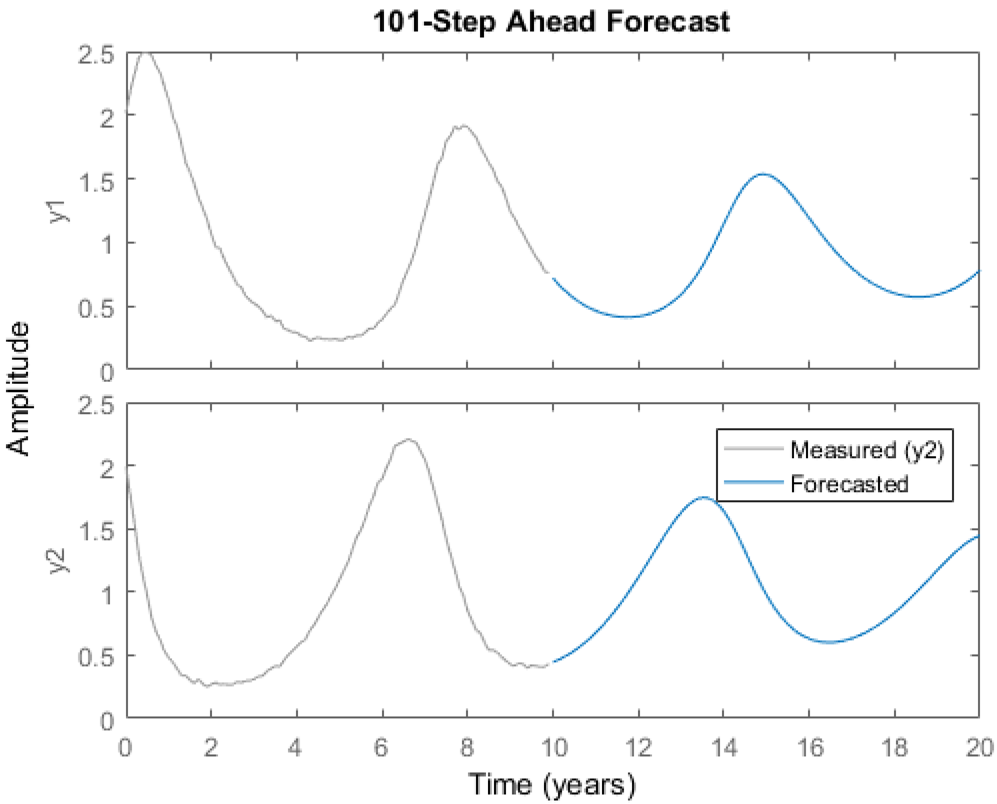
Compare the simulated output of `sys` with measured data to ensure it is a good fit.

```
compare(past_data,sys);
```



Plot the forecasted output of `sys`.

```
forecast(sys,past_data,101);
legend('Measured','Forecasted');
```



Reproduce Forecasting Results by Simulation

Obtain past data, future inputs, and identified linear model.

```
load iddata3 z3
past_data = z3(1:100);
future_inputs = z3.u(101:end);
sys = polyest(z3,[2 2 2 0 0 1]);
```

Forecast the system response into the future for a given time horizon and future inputs.

```
K = size(future_inputs,1);
[yf,x0,sysf] = forecast(sys,past_data,K,future_inputs);
```

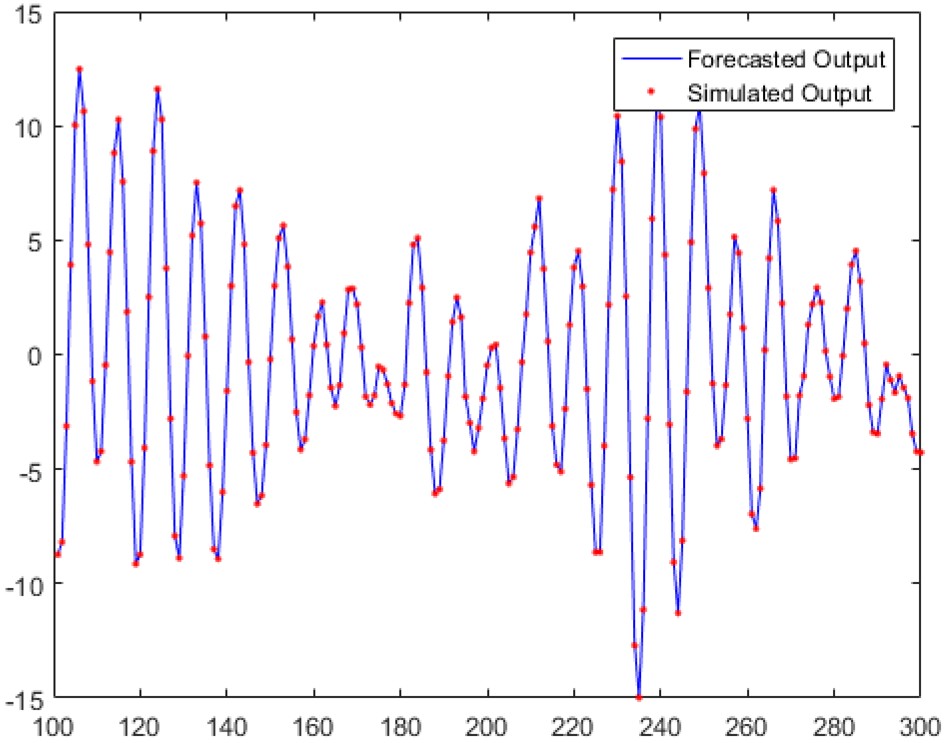
`yf` is the forecasted model response, `x0` is the estimated value for initial states, and `sysf` is the forecasting state-space model.

Simulate the forecasting state-space model with inputs, `future_inputs`, and initial conditions, `x0`.

```
opt = simOptions;
opt.InitialCondition = x0;
ys = sim(sysf,future_inputs(1:K),opt);
```

Plot the forecasted and simulated outputs.

```
t = yf.SamplingInstants;
plot(t,yf.OutputData,'b',t,ys,'.r');
legend('Forecasted Output','Simulated Output')
```



Simulation of forecasting model, `sysf`, with inputs, `future_inputs`, and initial conditions, `x0`, yields the forecasted output, `yf`.

- "Forecast the Output of a Dynamic System"
- "Perform Multivariate Time Series Forecasting"
- "Time Series Prediction and Forecasting for Prognosis"

Input Arguments

`sys` — Identified model

linear model | nonlinear model

Identified model whose output is to be forecasted, specified as one of the following:

- Linear model — `idpoly`, `idproc`, `idss`, `idtf`, or `idgrey`
- Nonlinear model — `idnlgrey`, `idnlhw`, or `idnlarx`

If a model is unavailable, estimate `sys` from `PastData` using commands such as `ar`, `arx`, `armax`, `nlarx`, and `ssest`.

`PastData` — Past input-output time-domain data

`iddata` object | matrix of doubles

Past input-output time-domain data, specified as one of the following:

- `iddata` object — Use observed input and output signals to create an `iddata` object. For time-series data (no inputs), specify as an `iddata` object with no inputs `iddata(output,[])`.
- Matrix of doubles — For discrete-time models only. Specify as an *N*-by-*Ny* matrix for time-series data. Here, *N* is the number of observations and *Ny* is the number of outputs.

  For models with*Nu* inputs, specify `PastData` as an *N*-by-(*Ny*+*Nu*) matrix.

`K` — Time horizon of forecasting

positive integer

Time horizon of forecasting, specified as a positive integer. The output, `yf`, is calculated `K` steps into the future, such that the prediction time horizon is `Ts*K`.

`FutureInputs` — Future input values

`[]` | matrix of doubles | `iddata` object | cell array of matrices

Future input values, specified as one of the following:

- `[]` — Future input values are assumed to be zero, or equal to input offset levels (if they are specified in `opts`). For time series models, specify as `[]`.
- `iddata` object — Specify as an `iddata` object with no outputs.
- *K*-by-*Nu* matrix of doubles — `K` is the forecast horizon, and *Nu* is the number of inputs.

  If you have data from multiple experiments, you can specify a cell array of matrices, one for each experiment in `PastData`.

`opts` — Forecast options

`forecastOptions` option set

Forecast options, specified as a `forecastOptions` option set.

`Linespec` — Line style, marker, and color

character vector

Line style, marker, and color, specified as a character vector. For example, `'b'` or `'b+:'`.

For more information about configuring `Linespec`, see “Specify Line Style, Color, and Markers” in the MATLAB documentation.

Output Arguments

`yf` — Forecasted response

`iddata` object

Forecasted response, returned as an `iddata` object. `yf` is the forecasted response at times after the last sample time in `PastData`. `yf` contains data for the time interval `T0+(N+1:N+K)*T1`, where `T0 = PastData.Tstart` and `T1 = PastData.Ts`. *N* is the number of samples in `PastData`.

`x0` — Estimated initial states

column vector | cell array

Estimated initial states at the start of forecasting, returned as a column vector of size equal to the number of states. Use `x0` with the forecasting model `sysf` to reproduce the result of forecasting by pure simulation.

If `PastData` is multiexperiment, `x0` is a cell array of size *Ne*, where *Ne* is the number of experiments.

When `sys` is not a state-space model (`idss`, `idgrey`, or `idnlgrey`), the definition of states depends on if `sys` is linear or nonlinear:

- Linear model (`idpoly`, `idproc`, `idtf`) – `sys` is converted to a discrete-time state-space model, and `x0` is returned as the states of the converted model at a time-point beyond the last data in `PastData`.

  If conversion of `sys` to `idss` is not possible, `x0` is returned empty. For example, if `sys` is a MIMO continuous-time model with irreducible internal delays.
- Nonlinear model (`idnlhw` or `idnlarx`) — For a definition of the states of `idnlarx` and `idnlhw` models, see “Definition of idnlarx States” on page 1-528, and “Definition of `idnlhw` States” on page 1-567.

`sysf` — Forecasting model

discrete-time `idss` | `idnlarx` | `idnlhw` | `idnlgrey` | cell array of models

Forecasting model, returned as one of the following:

- Discrete-time `idss` — If `sys` is a discrete-time `idss` model, `sysf` is the same as `sys`. If `sys` is a linear model that is not a state-space model (`idpoly`, `idproc`, `idtf`), or is a continuous-time state-space model (`idss`, `idgrey`), `sys` is converted to a discrete-time `idss` model. The converted model is returned in `sysf`.
- `idnlarx`, `idnlhw`, or `idnlgrey`— If `sys` is a nonlinear model, `sysf` is the same as `sys`.
- Cell array of models — If `PastData` is multiexperiment, `sysf` is an array of *Ne* models, where *Ne* is the number of experiments.

Simulation of `sysf` using `sim`, with inputs, `FutureInputs`, and initial conditions, `x0`, yields `yf` as the output. For time-series models, `FutureInputs` is empty.

`yf_sd` — Estimated standard deviations of forecasted response

matrix | cell array

Estimated standard deviations of forecasted response, returned as a *K*-by-*Ny* matrix, where `K` is the forecast horizon, and *Ny* is the number of outputs.

If `PastData` is multiexperiment, `yf_sd` is a cell array of size *Ne*, where *Ne* is the number of experiments.

`yf_sd` is empty if `sys` is a nonlinear ARX (`idnlarx`) or Hammerstein-Wiener model (`idnlhw`). `yf_sd` is also empty if `sys` does not contain parameter covariance information, that is if `getcov(sys)` is empty. For more information, see `getcov`.

`x` — Forecasted state trajectory

matrix | cell array

Forecasted state trajectory, returned as a *K*-by-*Nx* matrix, where `K`, the forecast horizon and *Nx* is the number of states. `x` are the states of the forecasting model.

If `PastData` is multiexperiment, `x` is a cell array of size *Ne*, where *Ne* is the number of experiments.

If `sys` is linear model other than a state-space model (not `idss` or `idgrey`), then it is converted to a discrete-time state-space model, and the states of the converted model are calculated. If conversion of `sys` to `idss` is not possible, `x` is returned empty. For example, if `sys` is a MIMO continuous-time model with irreducible internal delays.

`x` is empty if `sys` is a nonlinear ARX (`idnlarx`) or Hammerstein-Wiener model (`idnlhw`).

`x_sd` — Estimated standard deviations of forecasted states

matrix | cell array

Estimated standard deviations of forecasted states `x`, returned as a *K*-by-*Ns* matrix, where `K`, the forecast horizon and *Ns* is the number of states.

If `PastData` is multiexperiment, `x_sd` is a cell array of size *Ne*, where *Ne* is the number of experiments.

If `sys` is linear model other than a state-space model (not `idss` or `idgrey`), then it is converted to a discrete-time state-space model, and the states and standard deviations of the converted model are calculated. If conversion of `sys` to `idss` is not possible, `x_sd` is returned empty. For example, if `sys` is a MIMO continuous-time model with irreducible internal delays.

`x_sd` is empty if `sys` is a nonlinear ARX (`idnlarx`) or Hammerstein-Wiener model (`idnlhw`).

More About

Tips

- Right-clicking the plot opens the context menu, where you can access the following options:
  - **Systems** — Select systems to view forecasted ouput. By default, the forecasted output of all systems is plotted.
  - **Data Experiment** — For multi-experiment data only. Toggle between data from different experiments.
  - **Characteristics** — View the following data characteristics:
    - **Peak Value** — View peak value of the data.
    - **Mean Value** — View mean value of the data.
    - **Confidence Region** — View the estimated standard deviation of the forecasted output. To specify number of standard deviations to plot, double-click the plot and open the Property Editor dialog box. Specify the number of standard deviations in the **Options** tab, in **Confidence Region for Identified Models**. The default value is `1` standard deviation.

      The confidence region is not generated for nonlinear ARX and Hammerstein-Wiener models and models that do not contain parameter covariance information.
  - **Show Past Data** — Plot the past output data used for forecasting. By default, the past output data is plotted.
  - **I/O Grouping** — For datasets containing more than one input or output channel. Select grouping of input and output channels on the plot.
    - **None** — Plot input-output channels in their own separate axes.
    - **All** — Group all input channels together and all output channels together.
  - **I/O Selector** — For datasets containing more than one input or output channel. Select a subset of the input and output channels to plot. By default, all output channels are plotted.
  - **Grid** — Add grids to the plot.
  - **Normalize** — Normalize the y-scale of all data in the plot.
  - **Full View** — Return to full view. By default, the plot is scaled to full view.

  - **Properties** — Open the Property Editor dialog box to customize plot attributes.
- "Introduction to Forecasting of Dynamic System Response"

See Also

`ar` | `arx` | `compare` | `forecastOptions` | `iddata` | `predict` | `sim` | `ssest`

Introduced in R2012a

forecastOptions

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. `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` requires 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.

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

The offset value `InputOffset(i,j)` is subtracted from the *i*th input signal of the *j*th experiment in the `PastData` and `FutureInputs` arguments of `forecast`.

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. *Ny* is the number of outputs, and *Ne* is the number of experiments.

The offset value `OutputOffset(i,j)` is subtracted from the *i*th output signal of the *j*th experiment in the `PastData` argument of `forecast`. The offset value is added to the corresponding model output response after forecasting.

Default: `[]`

`'OutputWeight'`

Weight of output for initial condition estimation.

`OutputWeight` requires one of the following:

- `'[]'` — No weighting. This is the same as using `eye(Ny)`, where *Ny* is the number of outputs.
- `'noise'` — Inverse of the noise variance stored with the model.
- Matrix of doubles — A positive semidefinite matrix of dimension *Ny*-by-*Ny*, where *Ny* 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`

Introduced in R2012a

fpe

Akaike's Final Prediction Error for estimated model

Syntax

```
value = fpe(model)
value = fpe(model1,...,modeln)
```

Description

`value = fpe(model)` returns the Final Prediction Error (FPE) value for the estimated model.

`value = fpe(model1,...,modeln)` returns the FPE value for multiple estimated models.

Examples

Compute Final Prediction Error of Estimated Model

Estimate a transfer function model.

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

Compute the Final Prediction Error (FPE) value.

```
value = fpe(sys)
```

```
value =

    1.7252
```

Alternatively, use the `Report` property of the model to access the value.

```
sys.Report.Fit.FPE
```

```
ans =

    1.7252
```

Pick Model with Optimal Tradeoff Between Accuracy and Complexity Using FPE Criterion

Estimate multiple Output-Error (OE) models and use Akaike's Final Prediction Error (FPE) value to pick the one with optimal tradeoff between accuracy and complexity.

Load the estimation data.

```
load iddata2
```

Specify model orders varying in 1:4 range.

```
nf = 1:4;
nb = 1:4;
nk = 0:4;
```

Estimate OE models with all possible combinations of chosen order ranges.

```
NN = struc(nf,nb,nk);
models = cell(size(NN,1),1);
for ct = 1:size(NN,1)
   models{ct} = oe(z2, NN(ct,:));
end
```

Compute the small sample-size corrected AIC values for the models, and return the smallest value.

```
V = fpe(models{:});
[Vmin, I] = min(V);
```

Return the optimal model that has the smallest AICc value.

```
models{I}
```

```
ans =
Discrete-time OE model:  y(t) = [B(z)/F(z)]u(t) + e(t)
```

```
  B(z) = 1.067 z^-2

  F(z) = 1 - 1.824 z^-1 + 1.195 z^-2 - 0.2307 z^-3

Sample time: 0.1 seconds

Parameterization:
   Polynomial orders:   nb=1   nf=3   nk=2
   Number of free coefficients: 4
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using OE on time domain data "z2".
Fit to estimation data: 86.53%
FPE: 0.9809, MSE: 0.9615
```

Input Arguments

`model` — Identified model

`idtf` | `idgrey` | `idpoly` | `idproc` | `idss` | `idnlarx`, | `idnlhw` | `idnlgrey`

Identified model, specified as one of the following model objects:

- `idtf`
- `idgrey`
- `idpoly`
- `idproc`
- `idss`
- `idnlarx`, except nonlinear ARX model that includes a binary-tree or neural network nonlinearity estimator
- `idnlhw`
- `idnlgrey`

Output Arguments

`value` — Final Prediction Error (FPE) value

scalar | vector

Final Prediction Error (FPE) value, returned as a scalar or vector. For multiple models, `value` is a row vector where `value(k)` corresponds to the `k`th estimated model `modelk`.

More About

Akaike's Final Prediction Error (FPE)

Akaike's Final Prediction Error (FPE) criterion provides a measure of model quality by simulating the situation where the model is tested on a different data set. After computing several different models, you can compare them using this criterion. According to Akaike's theory, the most accurate model has the smallest FPE.

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:

$$FPE = \det\left(\frac{1}{N} \sum_{1}^{N} e\left(t,\theta_N\right)\left(e\left(t,\theta_N\right)\right)^T \right)\left(\frac{1 + d/N}{1 - d/N} \right)$$

where:

- N is the number of values in the estimation data set.
- $e(t)$ is a ny-by-1 vector of prediction errors.
- θ_N represents the estimated parameters.
- d is the number of estimated parameters.

If number of parameters exceeds the number of samples, FPE is not computed when model estimation is performed (`model.Report.FPE` is empty). The `fpe` command returns `NaN`.

Tips

- The software computes and stores the FPE value during model estimation. If you want to access this value, see the `Report.Fit.FPE` property of the model.
- “Loss Function and Model Quality Metrics”

References

[1] Ljung, L. *System Identification: Theory for the User*, Upper Saddle River, NJ, Prentice-Hal PTR, 1999. See sections 7.4 and 16.4.

See Also

`aic` | `goodnessOfFit`

Introduced before R2006a

frdata

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 `Ny` outputs and `Nu` inputs at `Nf` frequencies:

- `response` is an `Ny`-by-`Nu`-by-`Nf` multidimensional array where the `(i,j)` entry specifies the response from input `j` to output `i`.
- `freq` is a column vector of length `Nf` 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 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` | `spectrum` | `idfrd` | `freqresp`

Introduced before R2006a

freqresp

Frequency response over grid

Syntax

```
[H,wout] = freqresp(sys)
H = freqresp(sys,w)
H = freqresp(sys,w,units)
[H,wout,covH] = freqresp(idsys,...)
```

Description

`[H,wout] = freqresp(sys)` returns the frequency response of the dynamic system model `sys` at frequencies `wout`. The `freqresp` command automatically determines the frequencies based on the dynamics of `sys`.

`H = freqresp(sys,w)` returns the frequency response on the real frequency grid specified by the vector `w`.

`H = freqresp(sys,w,units)` explicitly specifies the frequency units of `w` with `units`.

`[H,wout,covH] = freqresp(idsys,...)` also returns the covariance `covH` of the frequency response of the identified model `idsys`.

Input Arguments

`sys`

Any dynamic system model or model array.

`w`

Vector of real frequencies at which to evaluate the frequency response. Specify frequencies in units of `rad/TimeUnit`, where `TimeUnit` is the time units specified in the `TimeUnit` property of `sys`.

`units`

Units of the frequencies in the input frequency vector `w`, specified as one of the following values:

- `'rad/TimeUnit'` — radians per the time unit specified in the `TimeUnit` property of `sys`
- `'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.

Examples

Compute Frequency Response of System

Create the following 2-input, 2-output system:

$$sys = \begin{bmatrix} 0 & \frac{1}{s+1} \\ \frac{s-1}{s+2} & 1 \end{bmatrix}$$

```
sys11 = 0;
sys22 = 1;
sys12 = tf(1,[1 1]);
sys21 = tf([1 -1],[1 2]);
sys = [sys11,sys12;sys21,sys22];
```

Compute the frequency response of the system.

```
[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`.

Compute Frequency Response on Specified Frequency Grid

Create the following 2-input, 2-output system:

$$sys = \begin{bmatrix} 0 & \frac{1}{s+1} \\ \frac{s-1}{s+2} & 1 \end{bmatrix}$$

```
sys11 = 0;
sys22 = 1;
sys12 = tf(1,[1 1]);
sys21 = tf([1 -1],[1 2]);
sys = [sys11,sys12;sys21,sys22];
```

Create a logarithmically-spaced grid of 200 frequency points between 10 and 100 radians per second.

```
w = logspace(1,2,200);
```

Compute the frequency response of the system on the specified frequency grid.

```
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);
```

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.

More About

Frequency Response

In continuous time, the *frequency response* at a frequency ω 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)`,..., `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 $T_s = 1$.

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

$$D + C(j\omega - A)^{-1} B, \quad \omega = \omega_1, \ldots, \omega_N$$

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.

See Also

`evalfr` | `bode` | `nyquist` | `nichols` | `sigma` | `interp` | `spectrum`

Introduced before R2006a

fselect

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

`idfrd` | `interp` | `fcat` | `fdel` | `frd`

Introduced before R2006a

get

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`. `'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)
          Numerator: {[0 1]}
     Denominator: {[1 2]}
```

```
          Variable: 'z'
           IODelay: 0
        InputDelay: 0
       OutputDelay: 0
                Ts: 0.1000
          TimeUnit: 'seconds'
         InputName: {'voltage'}
         InputUnit: {''}
        InputGroup: [1x1 struct]
        OutputName: {''}
        OutputUnit: {''}
       OutputGroup: [1x1 struct]
              Name: ''
             Notes: {}
          UserData: 'hello'
      SamplingGrid: [1x1 struct]
```

or query only about the numerator and sample time values by

```
get(h,'Numerator')

ans =
    [1x2 double]
```

and

```
get(h,'Ts')

ans =
    0.1000
```

Because the numerator data (`NUmerator` property) is always stored as a cell array, the first command evaluates to a cell array containing the row vector `[0 1]`.

More About

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
```

return the values of the sample time, *A* matrix, and `UserData` property of the (state-space) model `sys`.

See Also

`getpvec` | `frdata` | `set` | `ssdata` | `tfdata` | `idssdata` | `polydata` | `getcov`

Introduced before R2006a

getcov

Parameter covariance of linear identified parametric model

Syntax

```
cov_data = getcov(sys)
cov_data = getcov(sys,cov_type)
cov_data = getcov(sys,cov_type,'free')
```

Description

`cov_data = getcov(sys)` returns the raw covariance of the parameters of a linear identified parametric model.

- If `sys` is a single model, then `cov_data` is an *np*-by-*np* matrix. *np* is the number of parameters of `sys`.
- If `sys` is a model array, then `cov_data` is a cell array of size equal to the array size of `sys`.

  `cov_data(i,j,k,...)` contains the covariance data for `sys(:,:,i,j,k,...)`.

`cov_data = getcov(sys,cov_type)` returns the parameter covariance as either a matrix or a structure, depending on the covariance type that is specified.

`cov_data = getcov(sys,cov_type,'free')` returns the covariance data of only the free model parameters.

Examples

Raw Parameter Covariance for Identified Model

Obtain the identified model.

```
load iddata1 z1
sys = tfest(z1,2);
```

Get the raw parameter covariance for the model.

```
cov_data = getcov(sys)
```

```
cov_data =

    1.2131   -4.3949   -0.0309   -0.5531         0
   -4.3949  115.0838    1.8598   10.6660         0
   -0.0309    1.8598    0.0636    0.1672         0
   -0.5531   10.6660    0.1672    1.2433         0
         0         0         0         0         0
```

`cov_data` contains the covariance matrix for the parameter vector `[sys.Numerator,sys.Denominator(2:end),sys.IODelay]`.

`sys.Denominator(1)` is fixed to `1` and not treated as a parameter. The covariance matrix entries corresponding to the delay parameter (fifth row and column) are zero because the delay was not estimated.

Raw Parameter Covariance for Identified Model Array

Obtain the identified model array.

```
load iddata1 z1;
sys1 = tfest(z1,2);
sys2 = tfest(z1,3);
sysarr = stack(1,sys1,sys2);
```

`sysarr` is a 2-by-1 array of continuous-time, identified transfer functions.

Get the raw parameter covariance for the models in the array.

```
cov_data = getcov(sysarr)
```

```
cov_data =

  2×1 cell array

    [5×5 double]
    [7×7 double]
```

`cov_data` is a 2-by-1 cell array. `cov_data{1}` and `cov_data{2}` are the raw parameter covariance matrices for `sys1` and `sys2`.

Raw Covariance of Estimated Parameters of Identified Model

Load the estimation data.

```
load iddata1 z1
z1.y = cumsum(z1.y);
```

Estimate the model.

```
init_sys = idtf([100 1500],[1 10 10 0]);
init_sys.Structure.Numerator.Minimum = eps;
init_sys.Structure.Denominator.Minimum = eps;
init_sys.Structure.Denominator.Free(end) = false;
opt = tfestOptions('SearchMethod','lm');
sys = tfest(z1,init_sys,opt);
```

`sys` is an `idtf` model with six parameters, four of which are estimated.

Get the covariance matrix for the estimated parameters.

```
cov_type = 'value';
cov_data = getcov(sys,cov_type,'free')
```

```
cov_data =

   1.0e+05 *

    0.0269   -0.1237   -0.0001   -0.0017
   -0.1237    1.0221    0.0016    0.0133
   -0.0001    0.0016    0.0000    0.0000
   -0.0017    0.0133    0.0000    0.0002
```

`cov_data` is a `4x4` covariance matrix, with entries corresponding to the four estimated parameters.

Factored Parameter Covariance for Identified Model

Obtain the identified model.

```
load iddata1 z1
```

```
sys = tfest(z1,2);
```

Get the factored parameter covariance for the model.

```
cov_type = 'factors';
cov_data = getcov(sys,cov_type);
```

Factored Parameter Covariance for Identified Model Array

Obtain the identified model array.

```
load iddata1 z1
sys1 = tfest(z1,2);
sys2 = tfest(z1,3);
sysarr = stack(1,sys1,sys2);
```

`sysarr` is a 2-by-1 array of continuous-time, identified transfer functions.

Get the factored parameter covariance for the models in the array.

```
cov_type = 'factors';
cov_data = getcov(sysarr,cov_type)
```

```
cov_data = 

  2×1 struct array with fields:

    R
    T
    Free
```

`cov_data` is a 2-by-1 structure array. `cov_data(1)` and `cov_data(2)` are the factored covariance structures for `sys1` and `sys2`.

Factored Covariance of Estimated Parameters of Identified Model

Load the estimation data.

```
load iddata1 z1
z1.y = cumsum(z1.y);
```

Estimate the model.

```
init_sys = idtf([100 1500],[1 10 10 0]);
init_sys.Structure.Numerator.Minimum = eps;
init_sys.Structure.Denominator.Minimum = eps;
init_sys.Structure.Denominator.Free(end) = false;
opt = tfestOptions('SearchMethod','lm');
sys = tfest(z1,init_sys,opt);
```

`sys`, an `idtf` model, has six parameters, four of which are estimated.

Get the factored covariance for the estimated parameters.

```
cov_type = 'factors';
cov_data = getcov(sys,cov_type,'free');
```

Input Arguments

`sys` — Linear identified parametric model

`idtf`, `idss`, `idgrey`, `idpoly`, or `idproc` object | model array

Linear identified parametric model, specified as an `idtf`, `idss`, `idgrey`, `idpoly`, or `idproc` model or an array of such models.

`cov_type` — Covariance type

`'value'` (default) | `'factors'`

Covariance return type, specified as either `'value'` or `'factors'`.

- If `cov_type` is `'value'`, then `cov_data` is returned as a matrix (raw covariance).
- If `cov_type` is `'factors'`, then `cov_data` is returned as a structure containing the factors of the covariance matrix.

  Use this option for fetching the covariance data if the covariance matrix contains nonfinite values, is not positive definite, or is ill conditioned. You can calculate the response uncertainty using the covariance factors instead of the numerically disadvantageous covariance matrix.

  This option does not offer a numerical advantage in the following cases:

  - `sys` is estimated using certain instrument variable methods, such as `iv4`.
  - You have explicitly specified the parameter covariance of `sys` using the deprecated `CovarianceMatrix` model property.

Data Types: `char`

Output Arguments

`cov_data` — Parameter covariance of `sys`

matrix or cell array of matrices | structure or cell array of structures

Parameter covariance of `sys`, returned as a matrix, cell array of matrices, structure, or cell array of structures.

- If `sys` is a single model and `cov_type` is `'value'`, then `cov_data` is an *np*-by-*np* matrix. *np* is the number of parameters of `sys`.

  The value of the nonzero elements of this matrix is equal to `sys.Report.Parameters.FreeParCovariance` when `sys` is obtained via estimation. The row and column entries that correspond to fixed parameters are zero.
- If `sys` is a single model and `cov_type` is `'factors'`, then `cov_data` is a structure with fields:
  - `R` — Usually an upper triangular matrix.
  - `T` — Transformation matrix.
  - `Free` — Logical vector of length *np*, indicating if a model parameter is free (estimated) or not. *np* is the number of parameters of `sys`.

  To obtain the covariance matrix using the factored form, enter:

  ```
  Free = cov_factored.Free;
  T = cov_factored.T;
  R = cov_factored.R;
  np = nparams(sys);
  cov_matrix = zeros(np);
  cov_matrix(Free, Free) = T*inv(R'*R)*T';
  ```

  For numerical accuracy reasons, you can calculate `T*inv(R'*R)*T'` as `X*X'`, where `X = T/R`.
- If `sys` is a model array, then `cov_data` is a cell array of size equal to the array size of `sys`.

  `cov_data(i,j,k,...)` contains the covariance data for `sys(:,:,i,j,k,...)`.

More About

- “What Is Model Covariance?”
- “Types of Model Uncertainty Information”

See Also

`getpvec` | `nparams` | `rsample` | `setcov` | `sim` | `simsd`

Introduced in R2012a

getDelayInfo

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

Syntax

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

Description

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

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

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

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

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

The number of states of a nonlinear ARX model equals the sum of the maximum delays of each input and output variable. For more information about the definition of states for `idnlarx` models, see “Definition of idnlarx States” on page 1-528

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 `[ny, nu] = size(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,.., u1, u2,..]`.

Examples

Get Input and Output Delay Information for Nonlinear ARX Model

Create a two-output, three-input nonlinear ARX model.

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

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 model `M`. You can interpret the contents of matrix `Del` as follows:

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

- Similarly, in the dynamics for 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.

Find maximum delays for all the input and output variables in the order y_1, y_2, u_1, u_2, u_3.

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

```
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 fourth entry in `Del`), there are 5 terms corresponding to u_2 in the state vector. 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` | `getreg` | `idnlarx`

Introduced in R2008b

getexp

Specific experiments from multiple-experiment data set

Syntax

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

Description

`d1 = getexp(data,ExperimentNumber)` retrieves specific experiments from multiple-experiment data set. `data` is an `iddata` object that contains several experiments. `d1` is another `iddata` object containing the indicated experiment(s). `ExperimentNumber` is the experiment number as in `d1 = getexp(data,3)` or `d1 = getexp(data,[4 2])`.

`d1 = getexp(data,ExperimentName)` specifies the experiment name 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.

Introduced before R2006a

getinit

Values of `idnlgrey` model initial states

Syntax

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

Arguments

`model`

Name of the `idnlgrey` model object.

`Property`

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

Default: `'Value'`.

Description

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

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

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

See Also

`getpar` | `idnlgrey` | `setinit` | `setpar`

Introduced in R2007a

getoptions

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`

Introduced in R2012a

idParametric/getpar

Obtain attributes such as values and bounds of linear model parameters

Syntax

```
value = getpar(sys,'value')
free = getpar(sys,'free')
bounds = getpar(sys,'bounds')
label = getpar(sys,'label')
getpar(sys)
```

Description

`value = getpar(sys,'value')` returns the parameter values of the model `sys`. If `sys` is a model array, the returned value is a cell array of size equal to the model array.

`free = getpar(sys,'free')` returns the free or fixed status of the parameters.

`bounds = getpar(sys,'bounds')` returns the minimum and maximum bounds on the parameters.

`label = getpar(sys,'label')` returns the labels for the parameters.

`getpar(sys)` prints a table of parameter values, labels, free status and minimum and maximum bounds.

Examples

Get Parameter Values

Get the parameter values of an estimated ARMAX model.

Estimate an ARMAX model.

```
load iddata8
```

```
init_data = z8(1:100);
na = 1;
nb = [1 1 1];
nc = 1;
nk = [0 0 0];
sys = armax(init_data,[na nb nc nk]);
```

Get the parameter values.

```
val = getpar(sys,'value')
```

```
val =

   -0.7519
   -0.4341
    0.4442
    0.0119
    0.3431
```

To set parameter values, use `sys = setpar(sys,'value',value)`.

Get Free Parameters and Their Bounds

Get the free parameters and their bounds for a process model.

Construct a process model, and set its parameter values and free status.

```
m = idproc('P2DUZI');
m.Kp = 1;
m.Tw = 100;
m.Zeta = .3;
m.Tz = 10;
m.Td = 0.4;
m.Structure.Td.Free = 0;
```

Here, the value of `Td` is fixed.

Get the parameter values.

```
Val = getpar(m,'Value')
```

```
Val =
```

```
    1.0000
  100.0000
    0.3000
    0.4000
   10.0000
```

Get the free statuses of the parameters.

```
Free = getpar(m,'Free')
```

```
Free =

  5×1 logical array

   1
   1
   1
   0
   1
```

The output indicates that `Td` is a fixed parameter and the remaining parameters are free.

Get the default bounds on the parameters.

```
MinMax = getpar(m,'bounds')
```

```
MinMax =

  -Inf   Inf
     0   Inf
     0   Inf
     0   Inf
  -Inf   Inf
```

Extract the values of the free parameters.

```
FreeValues = Val(Free)
```

```
FreeValues =

    1.0000
  100.0000
    0.3000
   10.0000
```

Extract the bounds on the free parameters.

```
FreeValBounds = MinMax(Free,:)
```

```
FreeValBounds =

  -Inf   Inf
     0   Inf
     0   Inf
  -Inf   Inf
```

Get Parameter Labels

Get the parameter labels of an estimated ARMAX model.

Estimate an ARMAX model.

```
load iddata8;
init_data = z8(1:100);
na = 1;
nb = [1 1 1];
nc = 1;
nk = [0 0 0];
sys = armax(init_data,[na nb nc nk]);
```

Assign parameter labels.

```
sys.Structure.A.Info(2).Label = 'a2';
```

Get the parameter labels.

```
label = getpar(sys,'label')
```

```
label =

  5×1 cell array

    'a2'
    ''
    ''
    ''
    ''
```

Obtain a Table of Model Parameter Attributes

Obtain a table of all model parameter attributes of an ARMAX model.

Estimate an ARMAX model.

```
load iddata8;
init_data = z8(1:100);
na = 4;
nb = [3 2 3];
nc = 2;
nk = [0 0 0];
sys = armax(init_data,[na nb nc nk]);
```

Get all parameter attributes.

```
getpar(sys)
```

```
------------------------------------------------------
  #  Label     Value   Free      Min.         Max.
------------------------------------------------------
  1.          -1.4328    1       -Inf          Inf
  2.            0.497    1       -Inf          Inf
  3.          0.22904    1       -Inf          Inf
  4.         -0.09849    1       -Inf          Inf
  5.         -0.10246    1       -Inf          Inf
  6.           1.1671    1       -Inf          Inf
  7.          0.39579    1       -Inf          Inf
  8.          0.97219    1       -Inf          Inf
  9.         0.026995    1       -Inf          Inf
 10.         -0.17113    1       -Inf          Inf
 11.          0.16155    1       -Inf          Inf
 12.          0.48468    1       -Inf          Inf
 13.          -1.8871    1       -Inf          Inf
```

```
 14.         0.97391    1       -Inf          Inf
```

Input Arguments

`sys` — Identified linear model
`idss` | `idproc` | `idgrey` | `idtf` | `idpoly` | array of model objects

Identified linear model, specified as an `idss`, `idpoly`, `idgrey`, `idtf`, or `idfrd` model object or an array of model objects.

Output Arguments

`value` — Parameter values
vector of doubles

Parameter values, returned as a double vector of length `nparams(sys)`.

`free` — Free or fixed status of parameters
vector of logical values

Free or fixed status of parameters, returned as a logical vector of length `nparams(sys)`.

`bounds` — Minimum and maximum bounds on parameters
matrix of doubles

Minimum and maximum bounds on parameters, returned as a double matrix of size `nparams(sys)`-by-2. The first column contains the minimum bound, and the second column the maximum bound.

`label` — Parameter labels
cell array of character vectors

Parameter labels, returned as a cell array of character vectors of length `nparams(sys)`. For example, `{'a2','a3'}`, if `nparams(sys)` is two.

See Also

`getcov` | `getpvec` | `idssdata` | `polydata` | `setpar` | `tfdata`

Introduced in R2013b

getpar

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 `Np`-by-1 cell array of values, where `Np` is the number of parameters.

See Also

`getinit` | `idnlgrey` | `setinit` | `setpar` | `getpvec`

Introduced in R2007a

getpvec

Model parameters and associated uncertainty data

Syntax

```
pvec = getpvec(sys)
[pvec,pvec_sd] = getpvec(sys)
[ ___ ] = getpvec(sys,'free')
```

Description

`pvec = getpvec(sys)` returns a vector, `pvec`, containing the values of all the parameters of the identified model `sys`.

`[pvec,pvec_sd] = getpvec(sys)` also returns the 1 standard deviation value of the uncertainty associated with the parameters of `sys`. If the model covariance information for `sys` is not available, `pvec_sd` is `[]`.

`[ ___ ] = getpvec(sys,'free')` returns data for only the free parameters of `sys`, using any of the output arguments in previous syntaxes.

Input Arguments

`sys`

Identified model.

Output Arguments

`pvec`

Values of the parameters of `sys`.

If `sys` is an array of models, then `pvec` is a cell array with parameter value vectors corresponding to each model in `sys`.

`pvec_sd`

1 standard deviation value of the parameters of `sys`.

If the model covariance information for `sys` is not available, `pvec_sd` is `[]`.

If `sys` is an array of models, then `pvec_sd` is a cell array with standard deviation vectors corresponding to each model in `sys`.

Examples

Retrieve Parameter Values from Estimated Model

Load the estimation data.

```
load iddata1 z1;
```

Estimate a transfer function model.

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

Retrieve the parameter values from the estimated model.

```
pvec = getpvec(sys);
```

Retrieve Parameter Values and Standard Deviations from Estimated Model

Load the estimation data

```
load iddata2 z2;
```

Estimate a state-space model.

```
sys = ssest(z2,3);
```

Retrieve the model parameters, `pvec`, and associated standard deviations, `pvec_sd`, from the estimated model.

```
[pvec,pvec_sd] = getpvec(sys);
```

Retrieve Values of Free Parameters from Estimated Model

Load the estimation data.

```
load iddata2 z2;
```

Estimate a state-space model.

```
sys = ssest(z2,3);
```

Retrieve the values of the free parameters from the estimated model.

```
pvec = getpvec(sys,'free');
```

See Also

`getcov` | `idssdata` | `setpvec` | `tfdata` | `zpkdata`

Introduced in R2012a

getreg

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. `model` is an `idnlarx` object.

`Rs = getreg(model,subset)` returns regressor expressions for a specified subset of regressors.

`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` command. 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`

Subset of all regressors, specified as one of the following values:

- (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 character vectors. For a model with `ny` outputs, `Rs` is an `ny`-by-1 cell array of cell array of character vectors. 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 Predicted Model Output

Load sample data u and y.

```
load twotankdata;
Ts = 0.2;
```

Sample time is 0.2 min

Create data object and use first 1000 samples for estimation.

```
z = iddata(y,u,Ts);
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');
```

More About

- “Identifying Nonlinear ARX Models”

See Also

`addreg` | `customreg` | `evaluate` | `polyreg`

Introduced in R2007a

getTrend

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

Remove Offsets From Data

Remove specified offset from input and output signals.

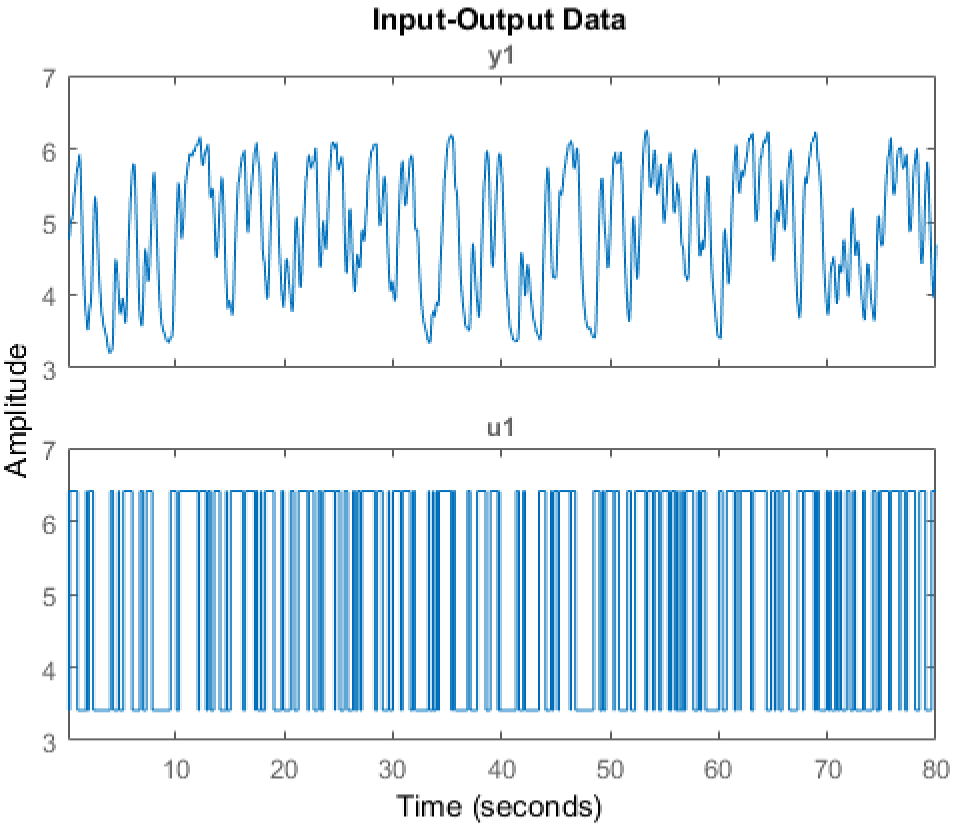
Load SISO data containing vectors `u2` and `y2`.

```
load dryer2
```

Create a data object with sample time of 0.08 seconds and plot it.

```
data = iddata(y2,u2,0.08);
```

```
plot(data)
```



The data has a nonzero mean value.

Store the data offset and trend information in a `TrendInfo` object.

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

Assign offset values to the `TrendInfo` object.

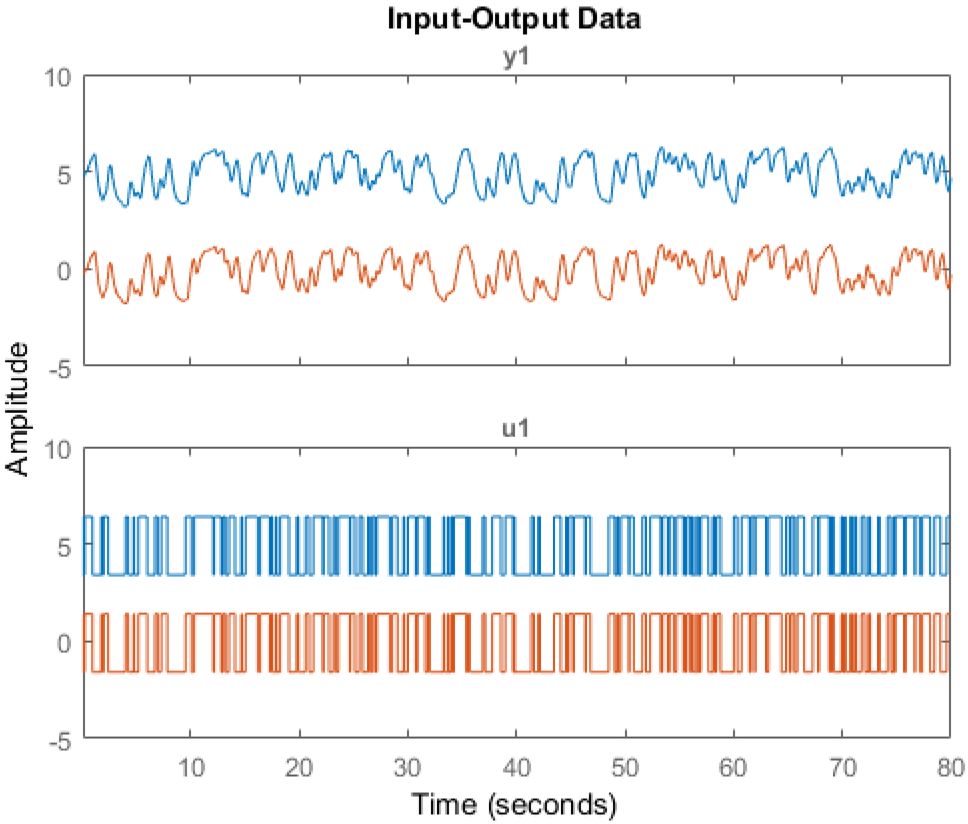
```
T.InputOffset = 5;
T.OutputOffset = 5;
```

Subtract offset from the data.

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

Plot the detrended data on the same plot.

```
hold on
plot(data_d)
```



View the mean value removed from the data.

```
get(T)
```

```
ans =
```

```
  struct with fields:

        DataName: 'data'
     InputOffset: 5
    OutputOffset: 5
      InputSlope: 0
     OutputSlope: 0
```

Compute and Store Means of Input and Output Signals

Compute input-output signal means, store them, and detrend the data.

Load SISO data containing vectors `u2` and `y2`.

```
load dryer2
```

Create a data object with sample time of 0.08 seconds.

```
data = iddata(y2,u2,0.08);
```

Compute the mean of the data.

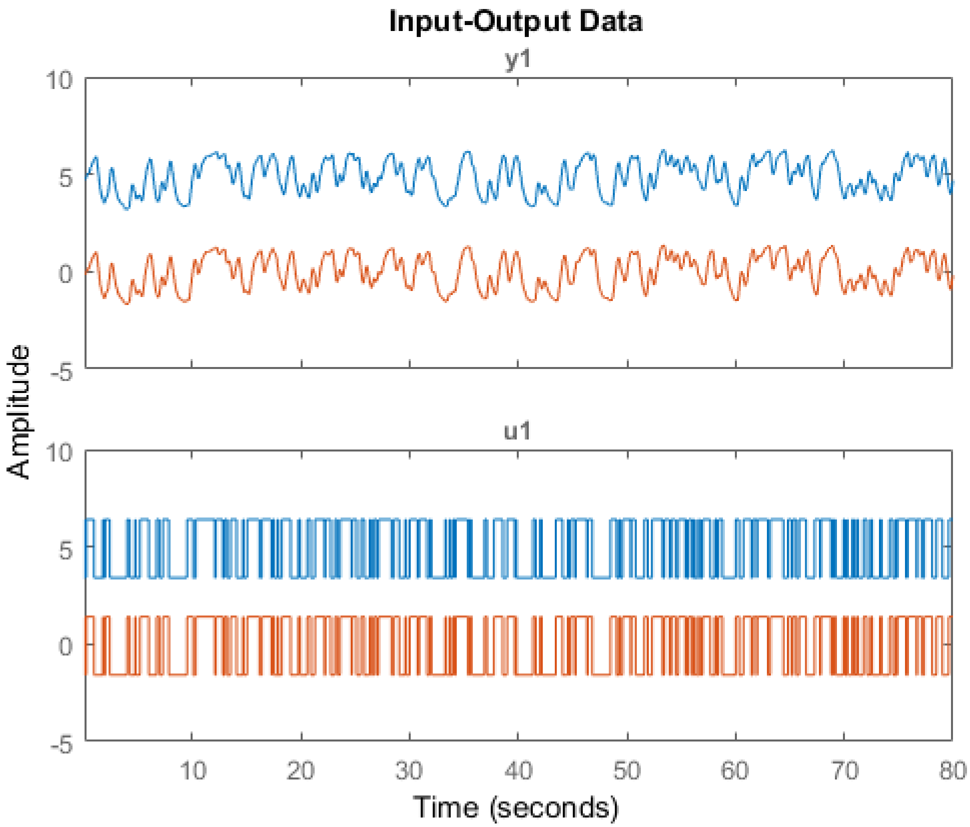
```
T = getTrend(data,0);
```

Remove the mean from the data.

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

Plot the original and detrended data on the same plot.

```
plot(data,data_d)
```



More About

- "Handling Offsets and Trends in Data"

See Also

`detrend` | `retrend` | `TrendInfo`

Introduced in R2009a

goodnessOfFit

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, `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 must 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` is specified as one of the following values:

- `'MSE'` — Mean square error:

  $$fit = \frac{\|x - xref\|^2}{Ns}$$

  where, N_s is the number of samples, and ‖ indicates the 2-norm of a vector. `fit` is a scalar value.
- `'NRMSE'` — Normalized root mean square error:

  $$fit(i) = 1 - \frac{\|xref(:,i) - x(:,i)\|}{\|xref(:,i) - mean(xref(:,i))\|}$$

  where, ‖ indicates the 2-norm of a vector. `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:

  $$fit(i) = 1 - \left\| \frac{xref(:,i) - x(:,i)}{xref(:,i) - mean(xref(:,i))} \right\|^2$$

  where, ‖ indicates the 2-norm of a vector. `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 returned as a:

- Scalar if `cost_func` is `MSE`.
- Row vector of length `N` if `cost_func` is `NRMSE` or `NMSE`. `N` is the number of channels.

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.

Examples

Calculate Goodness of Fit of Between Estimated and Measured Data

Obtain the measured output.

```
load iddata1 z1
yref = z1.y;
```

`z1` is an `iddata` object containing measured input/output data. `z1.y` is the measured output.

Obtain the estimated output.

```
sys = tfest(z1,2);
y_sim = sim(sys,z1(:,[],:));
```

`sys` is a second-order transfer function estimated using the measured input/output data. `y` is the output estimated using `sys` and the measured input.

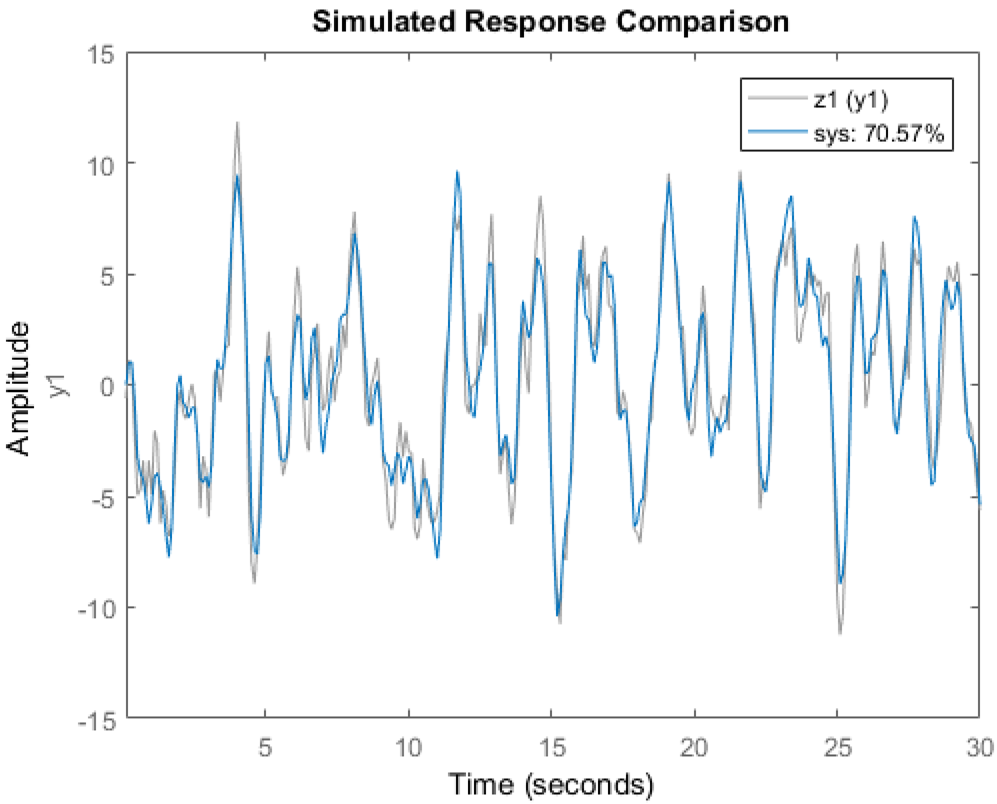
Calculate the goodness of the fit between the measured and estimated outputs.

```
cost_func = 'NRMSE';
y = y_sim.y;
fit = goodnessOfFit(y,yref,cost_func);
```

The goodness of fit is calculated using the normalized root mean square error as the cost function.

Alternatively, you can use `compare` to calculate the goodness of fit:

```
opt = compareOptions('InitialCondition','z');
compare(z1,sys,opt);
```



See Also

`aic` | `compare` | `fpe` | `pe` | `resid`

Introduced in R2012a

greyest

Linear grey-box model estimation

Syntax

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

Description

`sys = greyest(data,init_sys)` estimates a linear grey-box model, `sys`, using time or frequency domain data, `data`. The dimensions of the inputs and outputs of `data` and `init_sys`, an `idgrey` model, must match. `sys` is an identified `idgrey` model that has the same structure as `init_sys`.

`sys = greyest(data,init_sys,opt)` estimates a linear grey-box model using the option set, `opt`, to configure the estimation options.

Input Arguments

`data`

Estimation data.

The dimensions of the inputs and outputs of `data` and `init_sys` must match.

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 `Domain` property set to `'Frequency'`

`init_sys`

Identified linear grey-box model that configures the initial parameterization of `sys`.

`init_sys`, an `idgrey` model, must have the same input and output dimensions as `data`.

`opt`

Estimation options.

`opt` is an option set, created using `greyestOptions`, which specifies options including:

- Estimation objective
- Initialization choice
- Disturbance model handling
- Numerical search method to be used in estimation

Output Arguments

`sys`

Estimated grey-box model, returned as an `idgrey` model. This model is created using the specified initial system, and estimation options.

Information about the estimation results and options used is stored in the `Report` property of the model. `Report` has the following fields:

Report Field	Description
`Status`	Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.
`Method`	Estimation command used.
`InitialSt`	Handling of initial states during estimation, returned as one of the following: • `'model'` — The initial state is parameterized by the ODE file used by the `idgrey` model. • `'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.

Report Field	Description
	• Vector of doubles of length *Nx*, where *Nx* is the number of states. For multiexperiment data, a matrix with *Ne* columns, where *Ne* is the number of experiments. This field is especially useful to view how the initial states were handled when the `InitialState` option in the estimation option set is `'auto'`.
`Disturbar`	Handling of the disturbance component (*K*) during estimation, returned as one of the following values: • `'model'` — *K* values are parameterized by the ODE file used by the `idgrey` model. • `'fixed'` — The value of the `K` property of the `idgrey` model is fixed to its original value. • `'none'` — *K* is fixed to zero. • `'estimate'` — *K* is treated as an independent estimation parameter. This field is especially useful to view the how the disturbance component was handled when the `DisturbanceModel` option in the estimation option set is `'auto'`.

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Fit</code></td><td>Quantitative assessment of the estimation, returned as a structure. See “Loss Function and Model Quality Metrics” for more information on these quality metrics. The structure has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>FitPerc</code></td><td>Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as a percentage.</td></tr>
<tr><td><code>LossFcn</code></td><td>Value of the loss function when the estimation completes.</td></tr>
<tr><td><code>MSE</code></td><td>Mean squared error (MSE) measure of how well the response of the model fits the estimation data.</td></tr>
<tr><td><code>FPE</code></td><td>Final prediction error for the model.</td></tr>
<tr><td><code>AIC</code></td><td>Raw Akaike Information Criteria (AIC) measure of model quality.</td></tr>
<tr><td><code>AICc</code></td><td>Small sample-size corrected AIC.</td></tr>
<tr><td><code>nAIC</code></td><td>Normalized AIC.</td></tr>
<tr><td><code>BIC</code></td><td>Bayesian Information Criteria (BIC).</td></tr>
</table></td></tr>
<tr><td><code>Parameter</code></td><td>Estimated values of model parameters.</td></tr>
<tr><td><code>OptionsUs</code></td><td>Option set used for estimation. If no custom options were configured, this is a set of default options. See <code>greyestOptions</code> for more information.</td></tr>
<tr><td><code>RandState</code></td><td>State of the random number stream at the start of estimation. Empty, <code>[]</code>, if randomization was not used during estimation. For more information, see <code>rng</code> in the MATLAB documentation.</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation, returned as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
<tr><td><code>Type</code></td><td>Data type.</td></tr>
<tr><td><code>Length</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time.</td></tr>
<tr><td><code>InterSam</code></td><td>Input intersample behavior, returned as one of the following values:
• <code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
• <code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
• <code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.</td></tr>
<tr><td><code>InputOff</code></td><td>Offset removed from time-domain input data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
<tr><td><code>OutputOf</code></td><td>Offset removed from time-domain output data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Terminati</code></td><td>Termination conditions for the iterative search used for prediction error minimization. Structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>WhyStop</code></td><td>Reason for terminating the numerical search.</td></tr>
<tr><td><code>Iteratio</code></td><td>Number of search iterations performed by the estimation algorithm.</td></tr>
<tr><td><code>FirstOrd</code></td><td>∞ -norm of the gradient search vector when the search algorithm terminates.</td></tr>
<tr><td><code>FcnCount</code></td><td>Number of times the objective function was called.</td></tr>
<tr><td><code>UpdateNo</code></td><td>Norm of the gradient search vector in the last iteration. Omitted when the search method is <code>'lsqnonlin'</code>.</td></tr>
<tr><td><code>LastImpr</code></td><td>Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is <code>'lsqnonlin'</code>.</td></tr>
<tr><td><code>Algorith</code></td><td>Algorithm used by <code>'lsqnonlin'</code> search method. Omitted when other search methods are used.</td></tr>
</table>
For estimation methods that do not require numerical search optimization, the <code>Termination</code> field is omitted.</td></tr>
</table>

For more information on using `Report`, see “Estimation Report”.

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');
data.InputName = 'Voltage';
data.InputUnit = 'V';
data.OutputName = {'Angular position', 'Angular velocity'};
```

```
data.OutputUnit = {'rad', 'rad/s'};
data.Tstart = 0;
data.TimeUnit = 's';
```

`data` is an `iddata` object containing the measured data for the outputs, the angular position, the angular velocity. It also contains the input, the driving voltage.

Create a grey-box model representing the system dynamics.

For the DC motor, choose the angular position (rad) and the angular velocity (rad/s) as the outputs and the driving voltage (V) as the input. Set up a linear state-space structure of the following form:

$$\dot{x}(t) = \begin{bmatrix} 0 & 1 \\ 0 & -\frac{1}{\tau} \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ \frac{G}{\tau} \end{bmatrix} u(t)$$

$$y(t) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x(t).$$

τ 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('motorDynamics',tau,'cd',G,0);
```

The governing equations in state-space form are represented in the MATLAB® file `motorDynamics.m`. To view the contents of this file, enter `edit motorDynamics.m` at the MATLAB command prompt.

G is a known quantity that is provided to `motorDynamics.m` as an optional argument.

τ is a free estimation parameter.

`init_sys` is an `idgrey` model associated with `motor.m`.

Estimate τ.

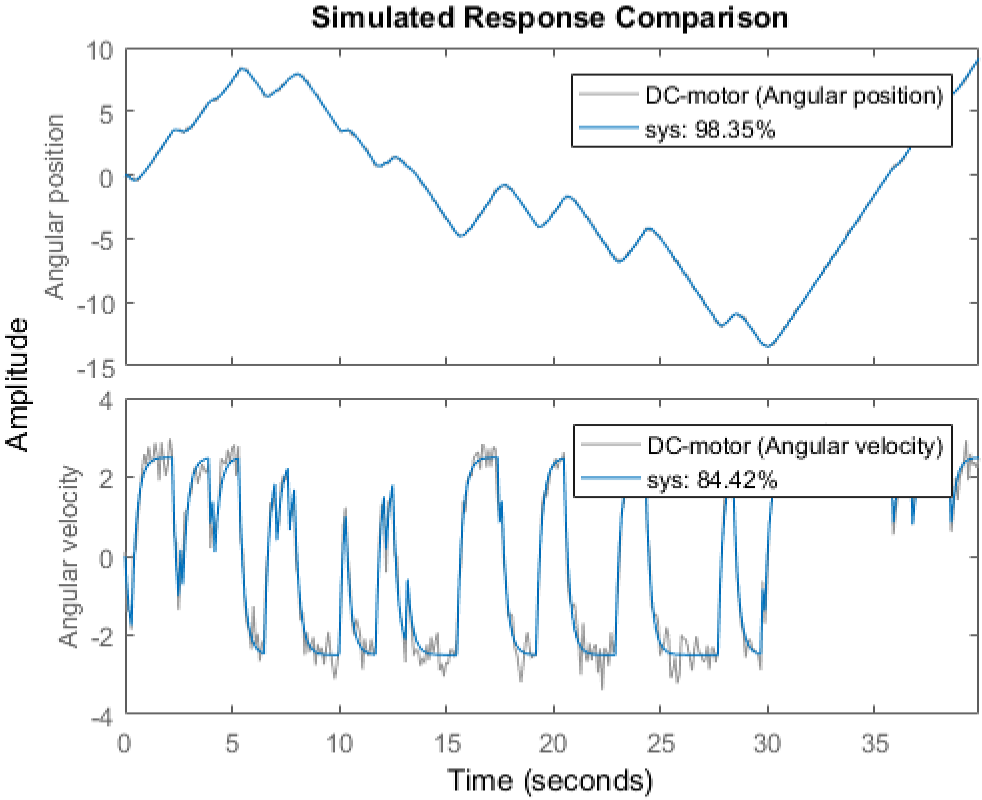
```
sys = greyest(data,init_sys);
```

`sys` is an `idgrey` model containing the estimated value of τ.

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.

Estimate Grey-Box Model Using Regularization

Estimate the parameters of a DC motor by incorporating prior information about the parameters when using regularization constants.

The model is parameterized by static gain `G` and time constant τ. From prior knowledge, it is known that `G` is about 4 and τ is about 1. Also, you have more confidence in the value of τ than `G` and would like to guide the estimation to remain close to the initial guess.

Load estimation data.

```
load regularizationExampleData.mat motorData
```

The data contains measurements of motor's angular position and velocity at given input voltages.

Create an `idgrey` model for DC motor dynamics. Use the function `DCMotorODE` that represents the structure of the grey-box model.

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

If you want to view the `DCMotorODE` function, type:

```
type DCMotorODE.m
```

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

%   Copyright 2013 The MathWorks, Inc.

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

```
end
```

Specify regularization options Lambda.

```
opt = greyestOptions;
opt.Regularization.Lambda = 100;
```

Specify regularization options R.

```
opt.Regularization.R = [1, 1000];
```

You specify more weighting on the second parameter because you have more confidence in the value of τ than `G`.

Specify the initial values of the parameters as regularization option θ*.

```
opt.Regularization.Nominal = 'model';
```

Estimate the regularized grey-box model.

```
sys = greyest(motorData, mi, opt);
```

- “Estimate Model Using Zero/Pole/Gain Parameters”
- “Regularized Estimates of Model Parameters”

See Also

`greyestOptions` | `iddata` | `idfrd` | `idgrey` | `idnlgrey` | `pem` | `ssest`

Introduced in R2012a

greyestOptions

Option set for `greyest`

Syntax

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

Description

`opt = greyestOptions` creates the default options set for `greyest`.

`opt = greyestOptions(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. `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'` — Handling of initial states

`'auto'` (default) | `'model'` | `'zero'` | `'estimate'` | `'backcast'`

Handling of initial states during estimation, specified as one of the following values:

- `'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 multiexperiment 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.

`'DisturbanceModel'` — Handling of disturbance component

`'auto'` (default) | `'model'` | `'fixed'` | `'none'` | `'estimate'`

Handling of the disturbance component (*K*) during estimation, specified as one of the following values:

- `'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 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.

`'Focus'` — Error to be minimized

`'prediction'` (default) | `'simulation'`

Error to be minimized in the loss function during estimation, specified as the comma-separated pair consisting of `'Focus'` and one of the following values:

- `'prediction'` — The one-step ahead prediction error between measured and predicted outputs is minimized during estimation. As a result, the estimation focuses on producing a good predictor model.
- `'simulation'` — The simulation error between measured and simulated outputs is minimized during estimation. As a result, the estimation focuses on making a good fit for simulation of model response with the current inputs.

The `Focus` option can be interpreted as a weighting filter in the loss function. For more information, see “Loss Function and Model Quality Metrics”.

`'WeightingFilter'` — Weighting prefilter

[ ] (default) | vector | matrix | cell array | linear system

Weighting prefilter applied to the loss function to be minimized during estimation. To understand the effect of `WeightingFilter` on the loss function, see “Loss Function and Model Quality Metrics”.

Specify `WeightingFilter` as one of the following values:

- `[]` — No weighting prefilter is used.
- Passbands — Specify a row vector or matrix containing frequency values that define desired passbands. You select a frequency band where the fit between estimated model and estimation data is optimized. For example, `[wl,wh]` where `wl` and `wh` represent lower and upper limits of a passband. For a matrix with several rows defining frequency passbands, `[w1l,w1h;w2l,w2h;w3l,w3h;...]`, the estimation algorithm uses the union of the frequency ranges to define the estimation passband.

  Passbands are expressed in `rad/TimeUnit` for time-domain data and in `FrequencyUnit` for frequency-domain data, where `TimeUnit` and `FrequencyUnit` are the time and frequency units of the estimation data.
- SISO filter — Specify a single-input-single-output (SISO) linear filter in one of the following ways:
  - A SISO LTI model
  - `{A,B,C,D}` format, which specifies the state-space matrices of a filter with the same sample time as estimation data.
  - `{numerator,denominator}` format, which specifies the numerator and denominator of the filter as a transfer function with same sample time as estimation data.

    This option calculates the weighting function as a product of the filter and the input spectrum to estimate the transfer function.
- Weighting vector — Applicable for frequency-domain data only. Specify a column vector of weights. This vector must have the same length as 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.

`'EnforceStability'` — Control whether to enforce stability of model

`false` (default) | `true`

Control whether to enforce stability of estimated model, specified as the comma-separated pair consisting of `'EnforceStability'` and either `true` or `false`.

Data Types: `logical`

`'EstCovar'` — Control whether to generate parameter covariance data

`true` (default) | `false`

Controls whether parameter covariance data is generated, specified as `true` or `false`.

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

`'Display'` — Specify whether to display the estimation progress

`'off'` (default) | `'on'`

Specify whether to display the estimation progress, specified as one of the following values:

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

`'InputOffset'` — Removal of offset from time-domain input data during estimation

`[]` (default) | vector of positive integers | matrix

Removal of offset from time-domain input data during estimation, specified as the comma-separated pair consisting of `'InputOffset'` and one of the following:

- A column vector of positive integers of length *Nu*, where *Nu* is the number of inputs.
- `[]` — Indicates no offset.
- *Nu*-by-*Ne* matrix — For multi-experiment data, specify `InputOffset` as an *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.

`'OutputOffset'` — Removal of offset from time-domain output data during estimation

`[]` (default) | vector | matrix

Removal of offset from time-domain output data during estimation, specified as the comma-separated pair consisting of `'OutputOffset'` and one of the following:

- A column vector of length *Ny*, where *Ny* is the number of outputs.
- `[]` — Indicates no offset.
- *Ny*-by-*Ne* matrix — For multi-experiment 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.

`'OutputWeight'` — Weighting of prediction errors in multi-output estimations

`[]` (default) | `'noise'` | positive semidefinite symmetric matrix

Weighting of prediction errors in multi-output estimations, specified as one of the following values:

- `'noise'` — Minimize $\det(E'^{*} E / N)$, where E represents the prediction error and `N` is the number of data samples. This choice is optimal in a statistical sense and leads to maximum likelihood estimates if 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 semidefinite symmetric matrix (`W`) — Minimize the trace of the weighted prediction error matrix `trace(E'*E*W/N)` where:
  - E is the matrix of prediction errors, with one column for each output, and 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-output models, or the reliability of corresponding data.
  - `N` is the number of data samples.
- `[]` — The software chooses between the `'noise'` or using the identity matrix for `W`.

This option is relevant for only multi-output models.

`'Regularization'` — Options for regularized estimation of model parameters

structure

Options for regularized estimation of model parameters. For more information on regularization, see “Regularized Estimates of Model Parameters”.

`Regularization` is a structure with the following fields:

- `Lambda` — Constant that determines the bias versus variance tradeoff.

  Specify a positive scalar to add the regularization term to the estimation cost.

  The default value of zero implies no regularization.

  Default: 0
- `R` — Weighting matrix.

  Specify a vector of nonnegative numbers or a square positive semi-definite matrix. The length must be equal to the number of free parameters of the model.

  For black-box models, using the default value is recommended. For structured and grey-box models, you can also specify a vector of `np` positive numbers such that each entry denotes the confidence in the value of the associated parameter.

  The default value of 1 implies a value of `eye(npfree)`, where `npfree` is the number of free parameters.

  Default: 1
- `Nominal` — The nominal value towards which the free parameters are pulled during estimation.

  The default value of zero implies that the parameter values are pulled towards zero. If you are refining a model, you can set the value to `'model'` to pull the parameters towards the parameter values of the initial model. The initial parameter values must be finite for this setting to work.

  Default: 0

`'SearchMethod'` — Search method used for iterative parameter estimation

`'auto'` (default) | `gn` | `gna` | `lm` | `lsqnonlin` | `grad`

Search method used for iterative parameter estimation, specified as one of 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` — The algorithm chooses one of the preceding options. The descent direction is calculated using `gn`, `gna`, `lm`, and `grad` successively at each iteration. The iterations continue until a sufficient reduction in error is achieved.

`'SearchOption'` — Option set for the search algorithm

search option set

Option set for the search algorithm with fields that depend on the value of `SearchMethod`.

SearchOption structure when SearchMethod is specified as 'gn', 'gna', 'lm', 'grad', or 'auto'

Field Name	Description
`Toleran`	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 stop. The estimate of the expected loss-function improvement at the next iteration is based on the Gauss-Newton vector computed for the current parameter value. **Default:** `0.01`
`MaxIter`	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 start-up procedure. Use `sys.Report.Termination.Iterations` to get the actual number of iterations during an estimation, where *sys* is an `idtf` model. **Default:** `20`
`Advance`	Advanced search settings. Specified as a structure with the following fields:

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td><code>GnPinvCon</code></td><td>Singular values of the Jacobian matrix that are smaller than <code>GnPinvConst*max(size(J)*norm(J)*eps)</code> are discarded when computing the search direction. Applicable when <code>SearchMethod</code> is <code>'gn'</code>.
<code>GnPinvConst</code> must be a positive, real value.
Default: <code>10000</code></td></tr>
<tr><td><code>InitGnaTo</code></td><td>Initial value of <i>gamma</i>. Applicable when <code>SearchMethod</code> is <code>'gna'</code>.
Default: <code>0.0001</code></td></tr>
<tr><td><code>LMStartVa</code></td><td>Starting value of search-direction length <i>d</i> in the Levenberg-Marquardt method. Applicable when <code>SearchMethod</code> is <code>'lm'</code>.
Default: <code>0.001</code></td></tr>
<tr><td><code>LMStep</code></td><td>Size of the Levenberg-Marquardt step. The next value of the search-direction length <i>d</i> in the Levenberg-Marquardt method is <code>LMStep</code> times the previous one. Applicable when <code>SearchMethod</code> is <code>'lm'</code>.
Default: <code>2</code></td></tr>
<tr><td><code>MaxBisect</code></td><td>Maximum number of bisections used by the line search along the search direction.
Default: <code>25</code></td></tr>
<tr><td><code>MaxFunEva</code></td><td>Iterations stop if the number of calls to the model file exceeds this value.
<code>MaxFunEvals</code> must be a positive, integer value.
Default: <code>Inf</code></td></tr>
<tr><td><code>MinParCha</code></td><td>Smallest parameter update allowed per iteration.
<code>MinParChange</code> must be a positive, real value.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>Default: <code>0</code></td></tr>
<tr><td><code>RelImprov</code></td><td>Iterations stop if the relative improvement of the criterion function is less than <code>RelImprovement</code>.
<code>RelImprovement</code> must be a positive, integer value.
Default: <code>0</code></td></tr>
<tr><td><code>StepReduc</code></td><td>Suggested parameter update is reduced by the factor <code>StepReduction</code> after each try. This reduction continues until either <code>MaxBisections</code> tries are completed or a lower value of the criterion function is obtained.
<code>StepReduction</code> must be a positive, real value that is greater than 1.
Default: <code>2</code></td></tr>
</table>
</td></tr>
</table>

SearchOption structure when SearchMethod is specified as 'lsqnonlin'

Field Name	Description
`TolFun`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values. The value of `TolFun` is the same as that of `opt.SearchOption.Advanced.TolFun`. **Default:** `1e-5`
`TolX`	Termination tolerance on the estimated parameter values. The value of `TolX` is the same as that of `opt.SearchOption.Advanced.TolX`. **Default:** `1e-6`

Field Name	Description
`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 `opt.SearchOption.Advanced.MaxIter`. **Default:** `20`
`Advance`	Options set for `lsqnonlin`. For more information, see the Optimization Options table in “Optimization Options”. Use `optimset('lsqnonlin')` to create an options set for `lsqnonlin`, and then modify it to specify its various options.

`'Advanced'` — Additional advanced options

structure

Additional advanced options, specified as 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 loss function. The standard deviation is estimated robustly as the median of the absolute deviations from the median of the prediction errors, divided by `0.7`. For more information on robust norm choices, see section 15.2 of [2].

  `ErrorThreshold = 0` disables robustification and leads to a purely quadratic loss function. When estimating with frequency-domain data, the software sets `ErrorThreshold` to zero. For time-domain data that contains outliers, try setting `ErrorThreshold` to `1.6`.

  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 state.

  The initial state is estimated when

  $$\frac{\left\| y_{p,z} - y_{meas} \right\|}{\left\| y_{p,e} - y_{meas} \right\|} > \text{AutoInitThreshold}$$

  - y_{meas} is the measured output.
  - $y_{p,z}$ is the predicted output of a model estimated using zero initial states.
  - $y_{p,e}$ is the predicted output of a model estimated using estimated initial states.

  Applicable when `InitialState` is `'auto'`.

  Default: `1.05`

Output Arguments

`opt` — Options set for `greyest`

`greyestOptions` option set

Option set for `greyest`, returned as an `greyestOptions` option set.

Examples

Create Default Options Set for Linear Grey Box Estimation

```
opt = greyestOptions;
```

Specify Options for Linear Grey Box Estimation

Create an options set for `greyest` using the `'backcast'` algorithm to initialize the state. Specify `Display` as `'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';
```

More About

- “Loss Function and Model Quality Metrics”

References

[1] 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.

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

See Also

`greyest` | `idgrey` | `idnlgrey` | `pem` | `ssest`

Introduced in R2012a

hasdelay

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, I/O delays, or internal 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`

Introduced in R2012a

iddata

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 sample time 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 sample time 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 sample time 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 sample time of the experimental data. `'P1',V1,...,'PN',VN` are property-value pairs, as described in “Properties” on page 1-448.

`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 1-448.

Here, `Ts` is the sample time, or the time interval, between successive data samples:

- For uniformly sampled data, `Ts` is a scalar value equal to the sample time of your experiment.
- 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 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 `Ts` is the sample time.

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 1-448.

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, `ny` is the number of output channels, `nu` is the number of input channels, and `Ne` is the number of experiments.

Tip Property names are not case sensitive. You do not need to type the entire property name. However, the portion you enter must by enough to uniquely identify the property.

Property Name	Description	Value
`Domain`	Specifies whether the data is in the time domain or frequency domain.	- `'Frequency'` — Frequency-domain data. - `'Time'` (Default) — Time-domain data.
`ExperimentName`	Name of each data set contained in the `iddata` object.	For `Ne` experiments, a `1`-by-`Ne` cell array of character vectors. Each cell contains the name of the corresponding experiment. Default names are `{'Exp1','Exp2',...}`.
`Frequency`	(Frequency-domain data only) Frequency values for defining the Fourier Transforms of the signals.	For a single experiment, this is an `N`-by-`1` vector. For `Ne` experiments, a `1`-by-`Ne` cell array and each cell contains the frequencies of the corresponding experiment.
`InputData`	Name of MATLAB variable that stores the input signal to a system.	For `nu` input channels and `N` data samples, this is an `N`-by-`nu` matrix.
`InputName`	Specifies the names of individual input channels.	Cell array of length `nu`-by-`1` contains the name of each input channel. Default names are `{'u1';'u2';...}`.
`InputUnit`	Specifies the units of each input channel.	Cell array of character vectors of length `nu`-by-`1`. Each cell contains the units of the corresponding input channel, for example `{'rad','rad/s'}`.

Property Name	Description	Value
`InterSample`	Specifies the behavior of the input signals between samples for transformations between discrete-time and continuous-time.	For a single experiment: • `zoh`— (Default) Zero-order hold maintains a piecewise-constant input signal between samples. • `foh`— First-order hold maintains a piecewise-linear input signal between samples. • `bl`— Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency. For `Ne` experiments, `InterSample` is an `nu`-by-`Ne` cell array. Each cell contains one of these values corresponding to each experiment.
`Name`	Name of the data set.	Character vector that contains the name of the data set. For example, `'dryer data'`. The default value is `''`.
`Notes`	Comments about the data set.	Character vector or cell array of character vectors that contain optional details about the data set. For example, `'data from experiment 2'`.
`OutputData`	Name of MATLAB variable that stores the output signal from a system.	For `ny` output channels and `N` samples, this is an `N`-by-`ny` matrix.

Property Name	Description	Value
`OutputName`	For a multiple-output system, specifies the names of individual output channels.	Cell array of length `ny`-by-`1` contains the name of each output channel. Default names are `{'y1';'y2';...}`.
`OutputUnit`	Specifies the units of each output channel.	A cell array of character vectors of length `ny`-by-`1`. Each cell contains the units of the corresponding output channel, for example `{'rad','rad/s'}`.
`Period`	Period of the input signal.	(Default) For a nonperiodic signal, set to `inf`. For a multiple-input signal, this is an `nu`-by-`1` vector and the `k`th entry contains the period of the `k`th input. For `Ne` experiments, this is a `1`-by-`Ne` cell array and each cell contains a scalar or vector of periods for the corresponding experiment.
`SamplingInstants`	(Time-domain data only) The time values in the time vector calculated from the properties `Tstart` and `Ts`.	For a single experiment, this is an `N`-by-`1` vector. For `Ne` experiments, this is a `1`-by-`Ne` cell array and each cell contains the sampling instants of the corresponding experiment.
`TimeUnit`	(Time-domain data only) Time unit.	Specify `TimeUnit` as one of the following values: `'nanoseconds'`,`'microseconds'`,`'millis` (default), `'minutes'`,`'hours'`, `'days'`, `'weeks'`, `'months'` or `'years'`.

Property Name	**Description**	**Value**
`Ts`	Time interval between successive data samples in seconds. Must be specified for both time- and frequency-domain data. For frequency-domain, it is used to compute Fourier transforms of the signals as discrete-time Fourier transforms (DTFT) with the indicated sample time. **Note:** Your data must be uniformly sampled.	Default value is `1`. For continuous-time data in the frequency domain, set to `0`; the inputs and outputs are interpreted as continuous-time Fourier transforms of the signals. Note that `Ts` is essential also for frequency-domain data, for proper interpretation of how the Fourier transforms were computed: They are interpreted as discrete-time Fourier transforms (DTFT) with the indicated sample time. For multiple-experiment data, `Ts` is a `1`-by-`Ne` cell array and each cell contains the sample time 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.

Property Name	Description	Value
`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.	Any MATLAB data type. The default value is `[]`.

Examples

Create an iddata Object for Time-Domain Data

Create an `iddata` object using single-input/single-output (SISO) data. The input and output each contain 1000 samples with the sample time of 0.08 second.

```
load dryer2
data = iddata(y2,u2,0.08)
```

```
data =

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

Outputs      Unit (if specified)
   y1
```

```
Inputs       Unit (if specified)
   u1
```

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

View and Modify Properties of iddata Object

To view and modify a property of an `iddata` object, use dot notation.

Load input `u2` and output `y2` of the data.

```
load dryer2
```

Create an `iddata` object.

```
data = iddata(y2,u2,0.08);
```

You can use `get(data)` to view all properties of the `iddata` object. You can specify properties when you create an `iddata` object using the constructor syntax. For example, `data = iddata(y,u,Ts,'Property1',Value1,...,'PropertyN',ValueN)`.

Use dot notation to change property values for an existing `iddata` object.

```
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.

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.

Examine iddata Object that Contains Frequency-Domain Data

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

Load input `u2` and output `y2` of the data.

```
load dryer2;
```

Create an `iddata` object.

```
data = iddata(y2,u2,0.08);
```

Transform teh data to frequency domain using the Fourier transform.

```
data = fft(data);
```

Get the frequency vector of the data.

```
data.Frequency;
```

Get the frequency units of the data.

```
data.Units;
```

See Also

`advice` | `detrend` | `fcat` | `getexp` | `idfilt` | `idfrd` | `plot` | `resample` | `size`

Introduced before R2006a

iddataPlotOptions

Option set for `iddata/plot`

Syntax

```
opt = iddataPlotOptions('time')
opt = iddataPlotOptions('frequency')
opt = iddataPlotOptions(___,'identpref')
```

Description

`opt = iddataPlotOptions('time')` creates the default option set for plotting time-domain data. Use dot notation to customize the option set, if needed.

`opt = iddataPlotOptions('frequency')` creates a default option set for plotting frequency-domain data. Use dot notation to customize the option set, if needed.

`opt = iddataPlotOptions(___,'identpref')` initializes the plot options with the System Identification Toolbox preferences. This syntax can include any of the input argument combinations in the previous syntaxes. Use this syntax to change a few plot options but otherwise use your toolbox preferences.

Examples

Create Option Set for Plotting Time-Domain Data

Create an options set with default options for time-domain data.

```
opt = iddataPlotOptions('time');
```

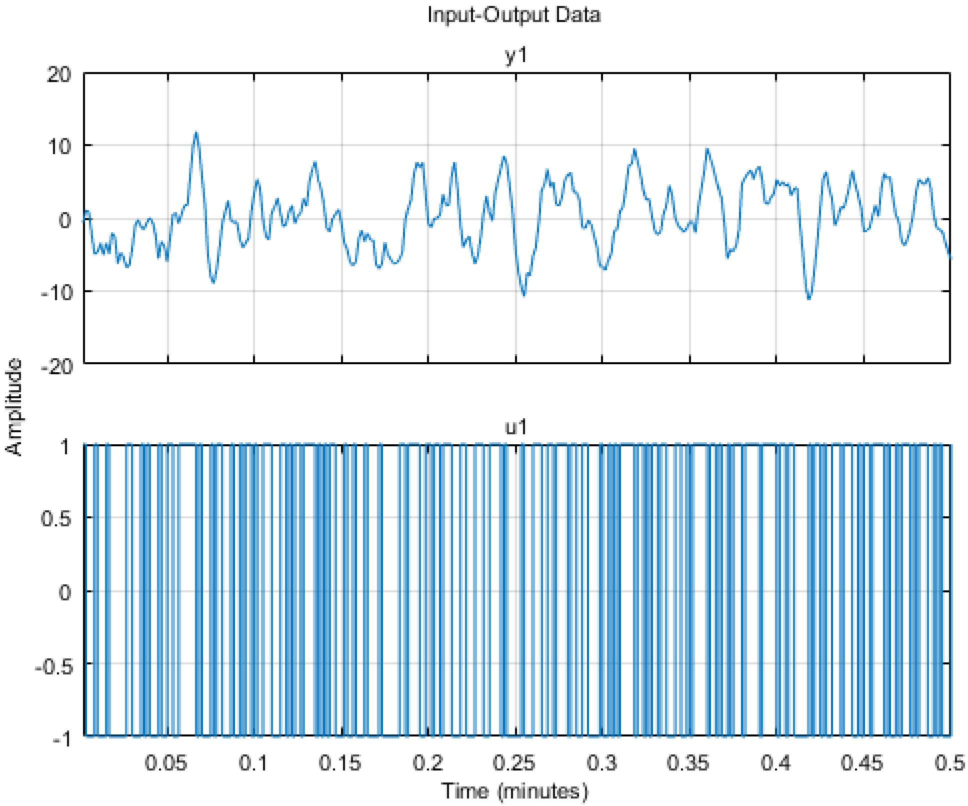
Specify plot properties, such as time units and grid. View the plot in minutes

```
opt.TimeUnits = 'minutes';
% Turn grid on
```

```
opt.Grid = 'on';
```

Create a plot using the specified options.

```
load iddata1 z1
h = plot(z1, opt);
```



Change Orientation of Input-Output Data Axes

Generate data with two inputs and one output.

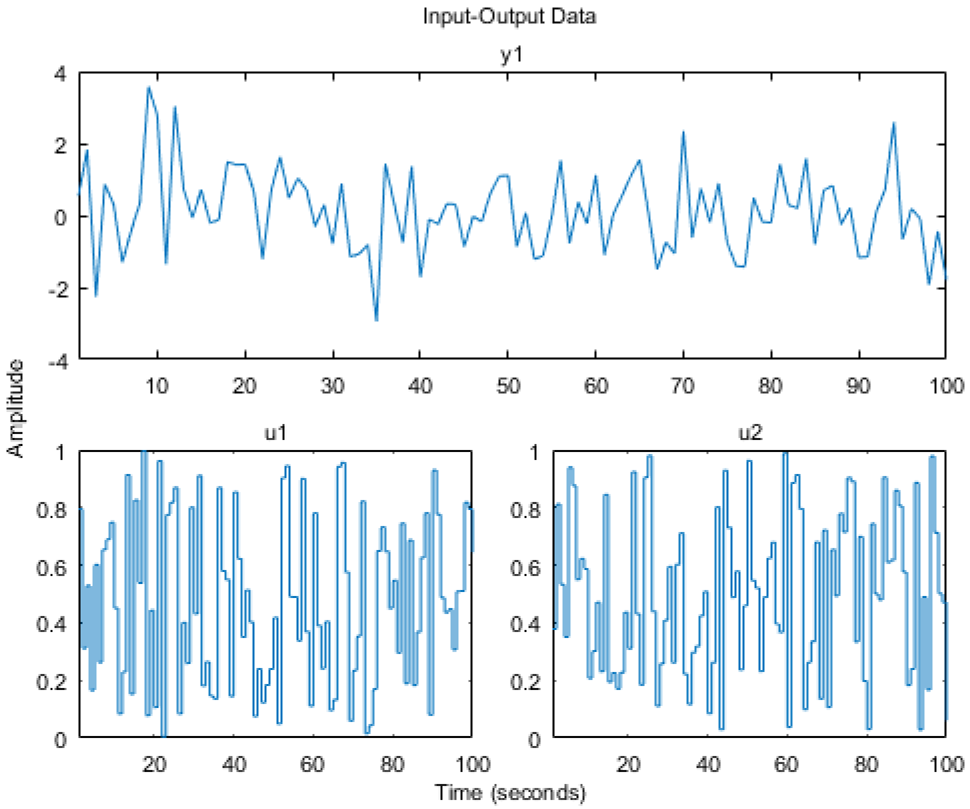
```
z = iddata(randn(100,1),rand(100,2));
```

Configure a time plot.

```
opt = iddataPlotOptions('time');
```

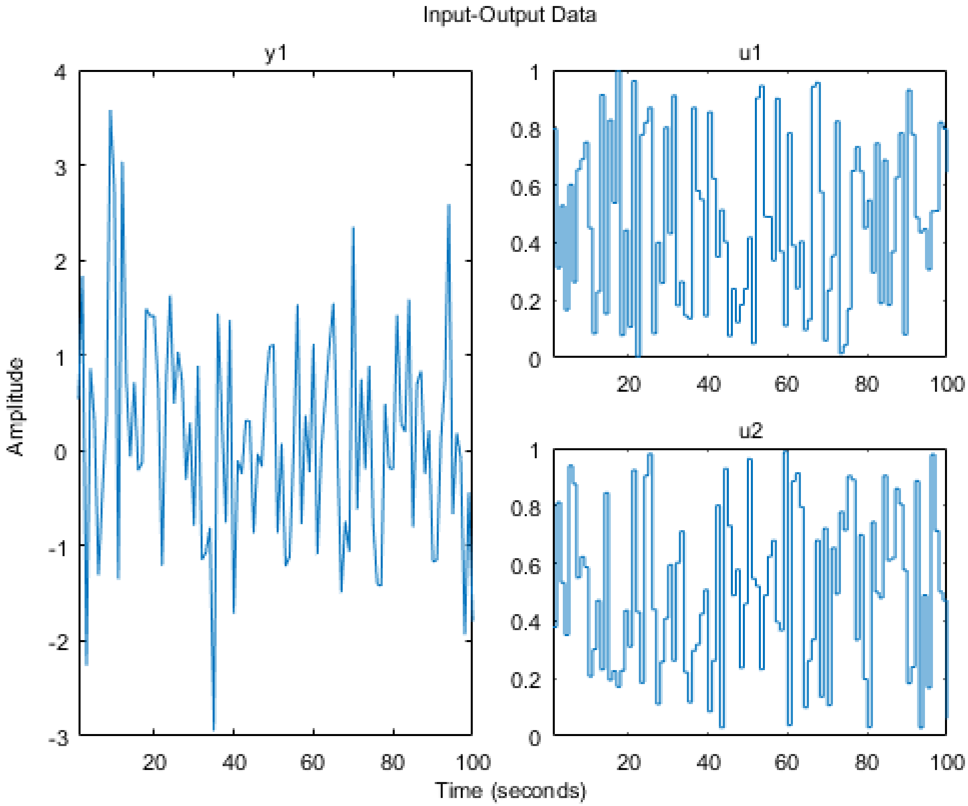
Plot the data.

```
h = plot(z,opt);
```



Change the orientation of the plots such that all inputs are plotted in one column, and all outputs are in a second column.

```
opt.Orientation = 'two-column';
h = plot(z,opt);
```



Alternatively, use `setoptions`.

```
setoptions(h,'Orientation','two-column')
```

You can also change the orientation by right-clicking the plot and choosing `Orientation` in the context menu.

Create Option Set for Plotting Frequency-Domain Data

Create an option set with default options for frequency-domain data.

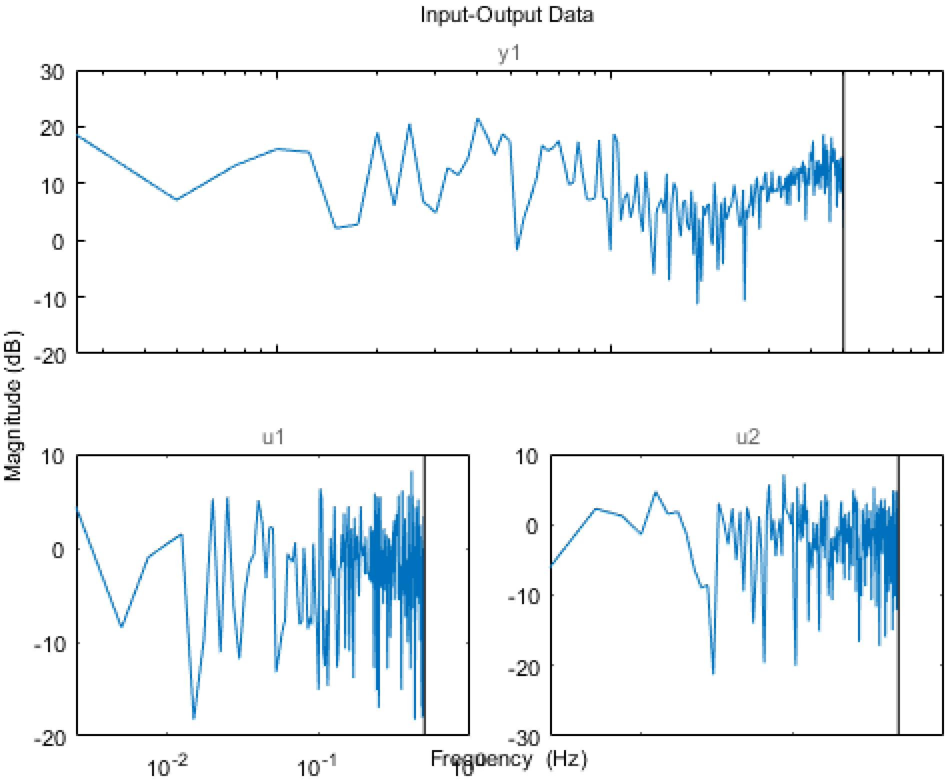
```
opt = iddataPlotOptions('frequency');
```

Specify plot properties, such as phase visibility and frequency units.

```
opt.PhaseVisible = 'off';
opt.FreqUnits = 'Hz';
```

Create a plot with the specified options.

```
load iddata7 z7
zf = fft(z7);
h = plot(zf,opt);
```



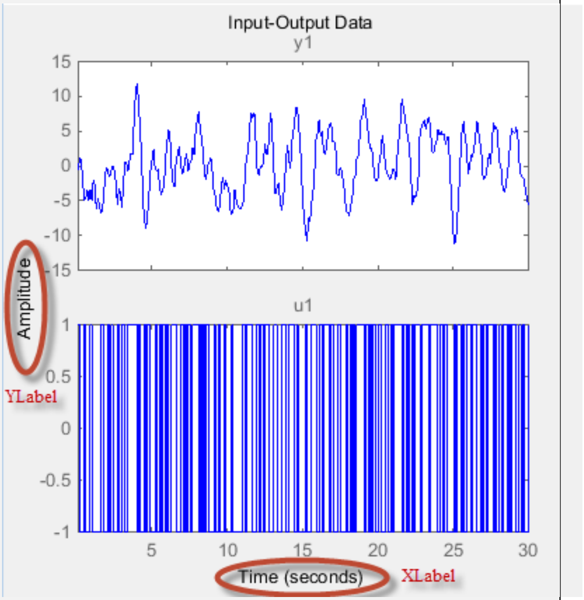
Initialize a Plot Using Toolbox Preferences

```
opt = iddataPlotOptions('time','identpref');
```

Output Arguments

`opt` — Option set for `iddata/plot`
`iddataPlotOptions` option set

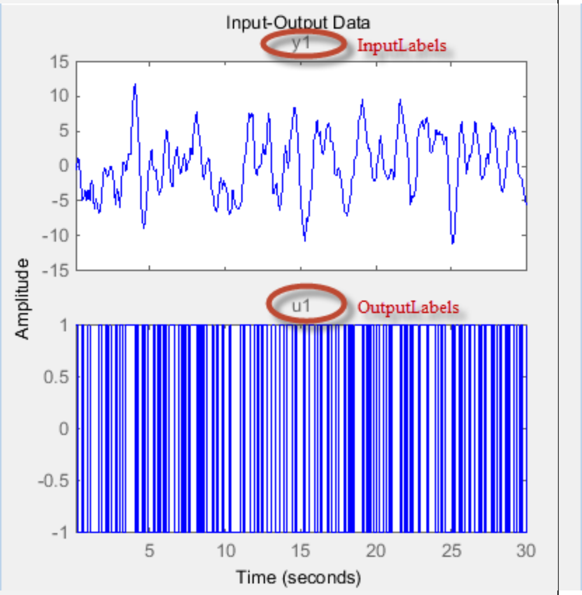
Option set containing the specified options for `iddata/plot`. The structure has the following fields:

Field	Description
`Title`, `XLabel`, `YLabel`	Text and style for axes labels and plot title, specified as a structure array with the following fields:  • `String` — Title and axes label text, specified as a character vector. **Default `Title`**: `'Input-Output Data'` **Default `XLabel`**: `'Time'` **Default `YLabel`**: `'Amplitude'` • `FontSize` — Font size, specified as scalar value greater than `0`. **Default**: `8` • `FontWeight` — Thickness of text, specified as one of the following values: `'Normal'` | `'Bold'`

Field	Description
	Default: `'Normal'` - `Font Angle` — Text character angle, specified as one of the following values: `'Normal'` | `'Italic'` **Default**: `'Normal'` - `Color` — Color of text, specified as vector of RGB values between `0` to `1`. **Default**: `[0,0,0]` - `Interpreter` — Interpretation of text characters, specified as one of the following values: `'tex'` | `'latex'`| `'none'` **Default**: `'tex'`
`TickLabel`	Tick label style, specified as a structure array with the following fields: - `FontSize` — Font size, specified as scalar value greater than `0`. **Default**: `8` - `FontWeight` — Thickness of text, specified as one of the following values: `'Normal'` | `'Bold'` **Default**: `'Normal'` - `Font Angle` — Text character angle, specified as one of the following values: `'Normal'` | `'Italic'` **Default**: `'Normal'` - `Color` — Color of text, specified as vector of RGB values between `0` to `1` | character vector of color name | `'none'`. For example, for yellow color, specify as one of the following: `[1 1 0]`, `'yellow'`, or `'y'`. **Default**: `[0,0,0]`

Field	Description
`Grid`	Show or hide the grid, specified as one of the following values: `'off'` | `'on'` **Default**: `'off'`
`GridColor`	Color of the grid lines, specified as one of the following values: vector of RGB values in the range `[0,1]` | character vector of color name | `'none'`. For example, for yellow color, specify as one of the following: `[1 1 0]`, `'yellow'`, or `'y'`. **Default**: `[0.15,0.15,0.15]`
`XlimMode`, `YlimMode`	Axes limit modes, specified as one of the following values: • `'auto'` — The axis limits are based on the data plotted • `'manual'` — The values explicitly set with `Xlim`, `Ylim` **Default**: `'auto'`
`Xlim`, `Ylim`	Axes limits, specified as maximum and minimum values. **Default**: `[0 1]`.
`IOGrouping`	Grouping of input-output pairs, specified as one of the following values: `'none'` | `'inputs'` | `'outputs'`|`'all'` **Default**: `'none'`

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>InputLabels</code>, <code>OutputLabels</code></td><td>
Input and output label styles on individual plot axes, specified as a structure array with the following fields:



- `FontSize` — Font size, specified as data type `scalar`.
Default: `8`
- `FontWeight` — Thickness of text, specified as one of the following values: `'Normal'` | `'Bold'`
Default: `'Normal'`
- `Font Angle` — Text character angle, specified as one of the following values: `'Normal'` | `'Italic'`
Default: `'Normal'`
- `Color` — Color of text, specified as a vector of RGB values between `0` to `1` | character vector of color name |
</td></tr>
</table>

Field	Description
	`'none'`. For example, for yellow color, specify as one of the following: `[1 1 0]`, `'yellow'`, or `'y'`. **Default**: `[0.4,0.4,0.4]` • `Interpreter` — Interpretation of text characters, specified as one of the following values: `'tex'` | `'latex'`| `'none'` **Default**: `'tex'`
`InputVisible`, `OutputVisible`	Visibility of input and output channels, specified as one of the following values: `'off'` | `'on'` **Default**: `'on'`
`Orientation`	Orientation of the input and output data plots, specified as one of the following values: • `'two-row'` — Plot all outputs in one row and all inputs in a second row • `'two-column'` — Plot all outputs in one column and all inputs in a second column • `'single-row'` — Plot all inputs and outputs in one row • `'single-column'`— Plot all inputs and outputs in one column **Default**: `'two-row'`.

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td colspan="2">For time-domain data plots only:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>TimeUnits</code></td><td>Time units, specified as one of the following values:
• <code>'nanoseconds'</code>
• <code>'microseconds'</code>
• <code>'milliseconds'</code>
• <code>'seconds'</code>
• <code>'minutes'</code>
• <code>'hours'</code>
• <code>'days'</code>
• <code>'weeks'</code>
• <code>'months'</code>
• <code>'years'</code>
You can also specify <code>'auto'</code> which uses time units specified in the <code>TimeUnit</code> property of the data. For multiple systems with different time units, the units of the first system is used.</td></tr>
<tr><td><code>Normalize</code></td><td>Normalize responses, specified as one of the following values: <code>'on'</code> |<code>'off'</code>
Default: <code>'off'</code></td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td colspan="2">For frequency-domain data plots only:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>FreqUnits</code></td><td>Frequency units, specified as one of the following values:
• <code>'Hz'</code>
• <code>'rad/second'</code>
• <code>'rpm'</code>
• <code>'kHz'</code>
• <code>'MHz'</code>
• <code>'GHz'</code>
• <code>'rad/nanosecond'</code>
• <code>'rad/microsecond'</code>
• <code>'rad/millisecond'</code>
• <code>'rad/minute'</code>
• <code>'rad/hour'</code>
• <code>'rad/day'</code>
• <code>'rad/week'</code>
• <code>'rad/month'</code>
• <code>'rad/year'</code>
• <code>'cycles/nanosecond'</code>
• <code>'cycles/microsecond'</code>
• <code>'cycles/millisecond'</code>
• <code>'cycles/hour'</code>
• <code>'cycles/day'</code>
• <code>'cycles/week'</code>
• <code>'cycles/month'</code>
• <code>'cycles/year'</code>
Default: <code>'rad/s'</code></td></tr>
</table>
</td></tr>
</table>

Field	Description
Field	**Description**
	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.
`FreqScale`	Frequency scale, specified as one of the following values: `'linear'` | `'log'` **Default**: `'log'`
`MagUnits`	Magnitude units, specified as one of the following values: `'dB'` | `'abs'` **Default**: `'dB'`
`MagScale`	Magnitude scale, specified as one of the following values: `'linear'` | `'log'` **Default**: `'linear'`
`MagVisible`	Magnitude plot visibility, specified as one of the following values: `'on'` | `'off'` **Default**: `'on'`
`MagLowerLimMode`	Enables a lower magnitude limit, specified as one of the following values: `'auto'` | `'manual'` **Default**: `'auto'`
`MagLowerLim`	Lower magnitude limit, , specified as data type `double`. It is typically decided by the range of the amplitudes the plotted data takes.
`PhaseUnits`	Phase units, specified as one of the following values: `'deg'` | `'rad'` **Default**: `'deg'`
`PhaseVisible`	Phase plot visibility, specified as one of the following values: `'on'` | `'off'` **Default**: `'on'`

Field	Description
Field	**Description**
`PhaseWrapping`	Enable phase wrapping, specified as one of the following values: `'on'` | `'off'` **Default**: `'off'`
`PhaseWrappingBranch`	Phase value at which the plot wraps accumulated phase when `PhaseWrapping` is set to `'on'`. **Default**: –180 (phase wraps into the interval [–180º,180º))
`PhaseMatching`	Enable phase matching, specified as one of the following values: `'on'` | `'off'` **Default**: `'off'`
`PhaseMatchingFreq`	Frequency for matching phase, specified as data type `double`.
`PhaseMatchingValue`	The value to which phase responses are matched closely, specified as a real number representing the desired phase value `PhaseMatchingFreq`.

See Also

`iddata/plot` | `identpref`

Introduced in R2014a

identpref

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.

More About

- "Toolbox Preferences Editor"

Introduced in R2012a

idfilt

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

`Zf = idfilt(Z,filter)` filters data using user-defined passbands, general filters, or Butterworth filters. `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 identified linear model or LTI model object. Alternatively the filter can be defined as a cell array `{A,B,C,D}` of SISO state-space matrices or as a cell array `{num,den}` of numerator/denominator filter coefficients.
- As a vector or matrix that defines one or several passbands.

  ```
  filter=[[wp1l,wp1h];[ wp2l,wp2h]; ....;[wpnl,wpnh]]
  ```

  The matrix is `n`-by-2, where each row defines a passband. 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 time-domain data — The passbands are in units of `rad/TimeUnit`, where `TimeUnit` is the time units of the estimation data.
  - For frequency-domain data — The passbands are in the frequency units (`FrequencyUnit` property) of the estimation data.

  For example, to define a stopband between `ws1` and `ws2`, use

```
filter = [0 ws1; ws2,Nyqf]
```

where `Nyqf` is the Nyquist frequency.

- For frequency-domain data, only the frequency response of the filter can be specified.

  ```
  filter = Wf
  ```

  Here `Wf` is a vector of possibly complex values that define the filter's frequency response, so that the inputs and outputs at frequency `Z.Frequency(kf)` are multiplied by `Wf(kf)`. `Wf` is a column vector of length = number of frequencies in `Z`. If the data object has several experiments, `Wf` is a cell array of length = # of experiments in `Z`.

`Zf = idfilt(Z,filter,causality)` specifies causality. For time-domain data, the filtering is carried out in the time domain as causal filtering as default. This corresponds to a last argument `causality = 'causal'`. With `causality = 'noncausal'`, a noncausal, zero-phase filter is used for the filtering (corresponding to `filtfilt` in the Signal Processing Toolbox product).

For frequency-domain data, the signals are multiplied by the frequency response of the filter. With the filters defined as passband, this gives ideal, zero-phase filtering ("brickwall filters"). Frequencies that have been assigned zero weight by the filter (outside the passband, or via the frequency response) are removed from the `iddata` object `Zf`.

`Zf = idfilt(Z,filter,'FilterOrder',NF)` specifies the filter order. The time domain filters in the pass-band case are calculated as cascaded Butterworth pass-band and stop-band filters. The orders of these filters are 5 by default, which can be changed to an arbitrary integer `NF`.

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 `'WeightingFilter'` option of the estimation command than just to prefilter the data. The values for `'WeightingFilter'` are the same as the argument `filter` in `idfilt`.

More About

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`

Introduced before R2006a

idfrd

Frequency-response data or model

Syntax

```
h = idfrd(Response,Freq,Ts)
h =
idfrd(Response,Freq,Ts,'CovarianceData',Covariance,'SpectrumData',Spec,'NoiseC
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 sample time. For a continuous-time system, set `Ts=0`.

`h =`
`idfrd(Response,Freq,Ts,'CovarianceData',Covariance,'SpectrumData',Spec,'NoiseC`
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

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

`idfrd` object stores the transfer function estimate $G\left(e^{i\omega}\right)$, as well as the spectrum of the additive noise (Φ_v) at the output.

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

where λ is the estimated variance of $e(t)$, and T is the sample time.

For a continuous-time system, the noise spectrum is given by:

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

Creating idfrd from Given Responses

`Response` is a 3-D array of dimension `ny`-by-`nu`-by-`Nf`, with `ny` being the number of outputs, `nu` the number of inputs, and `Nf` the number of frequencies (that is, the length of `Freqs`). `Response(ky,ku,kf)` is thus the complex-valued frequency response from input `ku` to output `ky` at frequency ω`=Freqs(kf)`. When defining the response of a SISO system, `Response` can be given as a vector.

`Freqs` is a column vector of length `Nf` containing the frequencies of the response.

`Ts` is the sample time. `Ts = 0` means a continuous-time model.

Intersample behavior: For discrete-time frequency response data (`Ts>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 *Nu*-by-1 cell array, where *Nu* is the number of inputs. The `InterSample` property is irrelevant for continuous-time data.

`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 `ny`-by-`ny`-by-`Nf`, 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 = []`.

Converting to idfrd

An `idfrd` object 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`.

Estimated covariance:

- If you obtain `mod` by identification, the software computes the estimated covariance for the `idfrd` object from the uncertainty information in `mod`. The software uses the Gauss approximation formula for this calculation for all model types, except grey-box models. For grey-box models (`idgrey`), the software applies numerical differentiation. The step sizes for the numerical derivatives are determined by `nuderst`.
- If you create `mod` by using commands such as `idss`, `idtf`, `idproc`, `idgrey`, or `idpoly`, then the software sets `CovarianceData` to `[]`.

Delay treatment: If `mod` contains delays, then the software assigns the delays of the `idfrd` object, `h`, as follows:

- `h.InputDelay = mod.InputDelay`

- `h.IODelay = mod.IODelay+repmat(mod.OutputDelay,[1,nu])`

  The expression `repmat(mod.OutputDelay,[1,nu])` returns a matrix containing the output delay for each input/output pair.

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 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 sample time, which is used when measuring or computing the frequency response. If you are working with a continuous-time system, set `Ts` to `0`.

`response(ky,ku,kf)`, where `ky`, `ku`, and `kf` reference the `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

`idfrd` object properties include:

`ResponseData`

Frequency response data.

The `'ResponseData'` property stores the frequency response data as a 3-D array of complex numbers. For SISO systems, `'ResponseData'` is a vector of frequency response values at the frequency points specified in the `'Frequency'` property. For MIMO systems with `Nu` inputs and `Ny` outputs, `'ResponseData'` is an array of size `[Ny Nu Nw]`, where `Nw` is the number of frequency points.

`Frequency`

Frequency points of the frequency response data. Specify `Frequency` values in the units specified by the `FrequencyUnit` property.

`FrequencyUnit`

Frequency units of the model.

Units of the frequency vector in the `Frequency` property, specified as one of the following values:

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

The units `'rad/TimeUnit'` and `'cycles/TimeUnit'` are relative to the time units specified in the `TimeUnit` property.

Changing this property changes the overall system behavior. Use `chgFreqUnit` to convert between frequency units without modifying system behavior.

Default: `'rad/TimeUnit'`

`SpectrumData`

Power spectra and cross spectra of the system output disturbances (noise). Specify `SpectrumData` as a 3-D array of complex numbers.

Specify `SpectrumData` as a 3-D array with dimension `ny`-by-`ny`-by-`Nf`.
Here, `ny` is the number of outputs and `Nf` is the number of frequency points. `SpectrumData(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, specify `SpectrumData` as a vector.

`CovarianceData`

Response data covariance matrices.

Specify `CovarianceData` as a 5-D array with dimension `ny`-by-`nu`-by-`Nf`-by-2-by-2. Here, `ny`, `nu`, and `Nf` are the number of outputs, inputs and frequency points, respectively. `CovarianceData(ky,ku,kf,:,:)` is the 2-by-2 covariance matrix of the response data `ResponseData(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,:,:))
```

`NoiseCovariance`

Power spectra variance.

Specify `NoiseCovariance` as a 3-D array with dimension `ny`-by-`ny`-by-`Nf`.
Here, `ny` is the number of outputs and `Nf` is the number of frequency points. `NoiseCovariance(ky1,ky1,kf)` is the variance of the corresponding power spectrum. To eliminate the influence of the noise component from the model, specify `NoiseVariance` as `0`. Zero variance makes the predicted output the same as the simulated output.

`Report`

Summary report that contains information about the estimation options and results when the frequency-response model is obtained using estimation commands, such as `spa`, `spafdr`, and `etfe`. Use `Report` to query a model for how it was estimated, including its:

- Estimation method
- Estimation options
- Search termination conditions
- Estimation data fit and other quality metrics

The contents of `Report` are irrelevant if the model was created by construction.

```
f = logspace(-1,1,100);
[mag,phase] = bode(idtf([1 .2],[1 2 1 1]),f);
response = mag.*exp(1j*phase*pi/180);
m = idfrd(response,f,0.08);
m.Report.Method
```

```
ans =

     ''
```

If you obtain the frequency-response model using estimation commands, the fields of `Report` contain information on the estimation data, options, and results.

```
load iddata3;
m = spa(z3);
m.Report.Method
```

```
ans =

SPA
```

`Report` is a read-only property.

For more information on this property and how to use it, see the Output Arguments section of the corresponding estimation command reference page and “Estimation Report”.

`InterSample`

Input intersample behavior.

Specifies the behavior of the input signals between samples for transformations between discrete-time and continuous-time. This property is meaningful for discrete-time `idfrd` models only.

Set `InterSample` to one of the following:

- `'zoh'` — The input signal used for construction/estimation of the frequency response data was subject to a zero-order-hold filter.
- `'foh'` — The input signal was subject to a first-order-hold filter.
- `'bl'` — The input signal has no power above the Nyquist frequency (`pi/sys.Ts` 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 `sys.Ts` to zero.

For multi-input data, specify `InterSample` as an *Nu*-by-1 cell array, where *Nu* is the number of inputs.

`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 in integer multiples of the sample time, `Ts`.

For a MIMO system with `Ny` outputs and `Nu` inputs, set `IODelay` to a `Ny`-by-`Nu` array. Each entry of this array is a numerical value that represents 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 delay for each input channel, specified as a scalar value or numeric vector. 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 sample time `Ts`. For example, `InputDelay = 3` means a delay of three sample times.

For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector. Each entry of this vector is a numerical value that represents 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

`OutputDelay`

Output delays.

For identified systems, like `idfrd`, `OutputDelay` is fixed to zero.

`Ts`

Sample time. For continuous-time models, `Ts = 0`. For discrete-time models, `Ts` is a positive scalar representing the sample time expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample time, set `Ts = -1`.

Changing this property does not discretize or resample the model.

Default: `1`

`TimeUnit`

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as one of the following values:

- `'nanoseconds'`
- `'microseconds'`
- `'milliseconds'`
- `'seconds'`
- `'minutes'`
- `'hours'`
- `'days'`
- `'weeks'`
- `'months'`

- `'years'`

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

Default: `'seconds'`

`InputName`

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, `'controls'`.
- Cell array of character vectors — For multi-input models.

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 input names automatically expand to `{'controls(1)';'controls(2)'}`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `InputName` to `data.InputName`.

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: `''` for all input channels

`InputUnit`

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, `'seconds'`.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

Default: `''` 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. In this structure, field names are the group names, and field values are the input channels belonging 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, specified as one of the following:

- Character vector — For single-output models. For example, `'measurements'`.
- Cell array of character vectors — For multi-output models.

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 output names automatically expand to `{'measurements(1)';'measurements(2)'}`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `OutputName` to `data.OutputName`.

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: `''` for all output channels

`OutputUnit`

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, `'seconds'`.
- Cell array of character vectors — For multi-output models.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

Default: `''` for all output 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. In this structure, field names are the group names, and field values are the output channels belonging 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, specified as a character vector. For example, `'system_1'`.

Default: `''`

`Notes`

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, `'System is MIMO'`.

Default: `{}`

`UserData`

Any type of data you want to associate with system, specified as any MATLAB data type.

Default: `[]`

`SamplingGrid`

Sampling grid for model arrays, specified as a data structure.

For arrays of identified linear (IDLTI) models that are derived by sampling one or more independent variables, this property tracks the variable values associated with each model. This information appears when you display or plot the model array. Use this information to trace results back to the independent variables.

Set the field names of the data structure to the names of the sampling variables. Set the field values to the sampled variable values associated with each model in the array. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, if you collect data at various operating points of a system, you can identify a model for each operating point separately and then stack the results together into a single system array. You can tag the individual models in the array with information regarding the operating point:

```
nominal_engine_rpm = [1000 5000 10000];
sys.SamplingGrid = struct('rpm', nominal_engine_rpm)
```

where `sys` is an array containing three identified models obtained at rpms 1000, 5000 and 10000, respectively.

For model arrays generated by linearizing a Simulink® model at multiple parameter values or operating points, the software populates `SamplingGrid` automatically with the variable values that correspond to each entry in the array. For example, the Simulink Control Design™ commands `linearize` and `slLinearizer` populate `SamplingGrid` in this way.

Default: `[]`

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

View and Modify Properties of idfrd Object

To view and modify a property of an `idfrd` object, use dot notation.

The following example shows how to create an `idfrd` object that contains 100 frequency-response values with a sample time of 0.08s and get its properties.

Create an `idfrd` object.

```
f = logspace(-1,1,100);
[mag, phase] = bode(idtf([1 .2],[1 2 1 1]),f);
response = mag.*exp(1j*phase*pi/180);
fr_data = idfrd(response,f,0.08);
```

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

You can use `get(fr_data)` to view all properties of the `idfrd` object. You can specify properties when you create an `idfrd` object using the constructor syntax. For example, `fr_data = idfrd(y,u,Ts,'Property1',Value1,...,'PropertyN',ValueN)` .

Use dot notation to change property values for an existing `idfrd` object. For example, change the name of the `idfrd` object.

```
fr_data.Name = 'DC_Converter';
```

If you import `fr_data` into the System Identification app, this data is named `DC_Converter` in the app, and not the variable name `fr_data` .

See Also

`bode` | `etfe` | `freqresp` | `nyquist` | `spa` | `spafdr` | `tfest`

Introduced before R2006a

idgrey

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:

$$\dot{x}(t) = Ax(t) + Bu(t) + Ke(t)$$
$$y(t) = Cx(t) + Du(t) + e(t)$$

In discrete time, the state-space model takes the form:

$$x[k+1] = Ax[k] + Bu[k] + Ke[k]$$
$$y[k] = Cx[k] + Du[k] + e[k]$$

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. You should be able to express these dynamics in the form of ordinary differential or difference equations. You specify complex relationships and constraints among the parameters that cannot be done through structured state-space models (`idss`).

You can create an `idgrey` model using the `idgrey` command. To do so, write a MATLAB function that returns the A, B, C, and D matrices for given values of the estimable parameters and sample 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 sample 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 Estimable 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 = 'motorDynamics';
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 `motorDynamics.m` to return the A, B, C, D, and K matrices. `motorDynamics.m` also returns the initial conditions, $X0$. The motor constant, τ, is defined in `motorDynamics.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 τ.

Configure Estimable Parameter of Grey-Box Model

Specify the known parameters of a grey-box model as fixed for estimation. Also specify a minimum bound for an estimable parameter.

Create an ODE file that relates the pendulum model coefficients to its state-space representation. Save this function as `LinearPendulum.m` such that it is in the MATLAB® search path.

```
function [A,B,C,D] = LinearPendulum(m,g,l,b,Ts)
A = [0 1; -g/l, -b/m/l^2];
B = zeros(2,0);
C = [1 0];
D = zeros(1,0);
end
```

In this function:

- `m` is the pendulum mass.
- `g` is the gravitational acceleration.
- `l` is the pendulum length.
- `b` is the viscous friction coefficient.
- `Ts` is the model sample time.

Create a linear grey-box model associated with the ODE function.

```
odefun = 'LinearPendulum';

m = 1;
g = 9.81;
l = 1;
b = 0.2;
parameters = {'mass',m;'gravity',g;'length',l;'friction',b};

fcn_type = 'c';

sys = idgrey(odefun,parameters,fcn_type);
```

`sys` has four parameters.

Specify the known parameters, `m`, `g`, and `l`, as fixed for estimation.

```
sys.Structure.Parameters(1).Free = false;
sys.Structure.Parameters(2).Free = false;
sys.Structure.Parameters(3).Free = false;
```

`m`, `g`, and `l` are the first three parameters of `sys`.

Specify a zero lower bound for `b`, the fourth parameter of `sys`.

```
sys.Structure.Parameters(4).Minimum = 0;
```

Similarly, to specify an upper bound for an estimable parameter, use the `Maximum` field of the parameter.

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.

Use `Name,Value` pair arguments to specify additional model properties on model creation.

```
odefun = 'motorDynamics';
parameters = 1;
fcn_type = 'cd';
optional_args = 0.25;
Ts = 0;
sys = idgrey(odefun,parameters,fcn_type,optional_args,Ts,'InputName','Voltage',...
            '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';
```

Create Array of Grey-Box Models

Use the `stack` command to create an array of linear grey-box models.

```
odefun1 = @motorDynamics;
parameters1 = [1 2];
fcn_type = 'cd';
optional_args1 = 1;
sys1 = idgrey(odefun1,parameters1,fcn_type,optional_args1);

odefun2 = 'motorDynamics';
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`.

- “Estimate Coefficients of ODEs to Fit Given Solution”
- “Estimate Model Using Zero/Pole/Gain Parameters”
- “Estimate Discrete-Time Grey-Box Model with Parameterized Disturbance”

Input Arguments

`odefun`

MATLAB function that relates the model parameters to its state-space representation.

`odefun` specifies 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,...)
```

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. Instead, these values 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` requires one of the following values:

- `'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`.

  Otherwise, 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 sample time.

If `Ts` is unspecified, it is assumed to be:

- `-1` — If `fcn_type` is `'d'` or `'cd'`.

  `Ts = -1` indicates a discrete-time model with unknown sample time.
- `0` — If `fcn_type` is `'c'`.

  `Ts = 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 Nx-by-Nx matrix, as returned by the ODE function associated with the `idgrey` model. Nx is the number of states.
- `B` — Input-to-state matrix B, an Nx-by-Nu matrix, as returned by the ODE function associated with the `idgrey` model. Nu is the number of inputs and Nx is the number of states.

- `C` — State-to-output matrix *C*, an *Ny*-by-*Nx* matrix, as returned by the ODE function associated with the `idgrey` model. *Nx* is the number of states and *Ny* is the number of outputs.
- `D` — Feedthrough matrix *D*, an *Ny*-by-*Nu* matrix, as returned by the ODE function associated with the `idgrey` model. *Ny* is the number of outputs and *Nu* is the number of inputs.

The values `A,B,C,D` are returned by the ODE function associated with the `idgrey` model. Thus, you can only read these matrices; you cannot set their values.

K

Value of state disturbance matrix, *K*

`K` is *Nx*-by-*Ny* matrix, where *Nx* is the number of states and *Ny* 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 five 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 *Nx*-by-*Ny*. *Nx* is the number of states and *Ny* is the number of outputs. The value is treated as a fixed value of the *K* matrix during estimation. To make the value estimable, use 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 *Nx*-by-*Ny* matrix. *Nx* is the number of states and *Ny* is the number of outputs. The specified value is treated as a fixed value of the *K* matrix during estimation. To make the value estimable, use the `DisturbanceModel` estimation option.

To create an estimation option set for `idgrey` models, use `greyestOptions`.

StateName

State names, specified as one of the following:

- Character vector — For first-order models, for example, `'velocity'`.
- Cell array of character vectors — For models with two or more states
- `''` — For unnamed states.

Default: `''` for all states

`StateUnit`

State units, specified as one of the following:

- Character vector — For first-order models, for example, `'velocity'`.
- Cell array of character vectors — For models with two or more states
- `''` — For unnamed states.

Use `StateUnit` to keep track of the units each state is expressed in. `StateUnit` has no effect on system behavior.

Default: `''` 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.FunctionType` — 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 value specifying whether the parameter is estimable. If you want to fix the value of a parameter during estimation, set `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.

    Specify parameter units and labels as character vectors. For example, `'Time'`.
- `Structure.ExtraArguments` — 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 Ny-by-Ny matrix, where Ny is the number of outputs in the system.

Report

Summary report that contains information about the estimation options and results when the grey-box model is obtained using the `greyest` estimation command. Use `Report` to query a model for how it was estimated, including its:

- Estimation method
- Estimation options
- Search termination conditions
- Estimation data fit and other quality metrics

The contents of `Report` are irrelevant if the model was created by construction.

```
odefun = 'motorDynamics';
m = idgrey(odefun,1,'cd',0.25,0);
m.Report.OptionsUsed
```

```
ans =

     []
```

If you obtain the grey-box model using estimation commands, the fields of `Report` contain information on the estimation data, options, and results.

```
load(fullfile(matlabroot,'toolbox','ident','iddemos','data','dcmotordata'));
data = iddata(y,u,0.1,'Name','DC-motor');
odefun = 'motorDynamics';
init_sys = idgrey('motorDynamics',1,'cd',0.25,0);
m = greyest(data,init_sys);
m.Report.OptionsUsed
```

```
InitialState: 'auto'
    DisturbanceModel: 'auto'
               Focus: 'prediction'
            EstCovar: 1
             Display: 'off'
         InputOffset: []
        OutputOffset: []
      Regularization: [1x1 struct]
        OutputWeight: []
        SearchMethod: 'auto'
        SearchOption: [1x1 idoptions.search.identsolver]
            Advanced: [1x1 struct]
```

`Report` is a read-only property.

For more information on this property and how to use it, see the Output Arguments section of the corresponding estimation command reference page and “Estimation Report”.

InputDelay

Input delay for each input channel, specified as a scalar value or numeric vector. 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 sample time `Ts`. For example, `InputDelay = 3` means a delay of three sample times.

For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector. Each entry of this vector is a numerical value that represents 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

`OutputDelay`

Output delays.

For identified systems, like `idgrey`, `OutputDelay` is fixed to zero.

`Ts`

Sample time.

For continuous-time models, `Ts = 0`. For discrete-time models, `Ts` is a positive scalar representing the sample time expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample time, set `Ts = -1`.

Changing this property does not discretize or resample the model.

For `idgrey` models, there is no unique default value for `Ts`. `Ts` depends on the value of `fcn_type`.

`TimeUnit`

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as one of the following values:

- `'nanoseconds'`
- `'microseconds'`
- `'milliseconds'`
- `'seconds'`
- `'minutes'`
- `'hours'`
- `'days'`
- `'weeks'`
- `'months'`
- `'years'`

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

Default: `'seconds'`

`InputName`

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, `'controls'`.
- Cell array of character vectors — For multi-input models.

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 input names automatically expand to `{'controls(1)';'controls(2)'}`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `InputName` to `data.InputName`.

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: `''` for all input channels

`InputUnit`

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, `'seconds'`.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

Default: `''` 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. In this structure, field names are the group names, and field values are the input channels belonging 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, specified as one of the following:

- Character vector — For single-output models. For example, `'measurements'`.
- Cell array of character vectors — For multi-output models.

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 output names automatically expand to `{'measurements(1)';'measurements(2)'}`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `OutputName` to `data.OutputName`.

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: `''` for all output channels

`OutputUnit`

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, `'seconds'`.
- Cell array of character vectors — For multi-output models.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

Default: `''` for all output 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. In this structure, field names are the group names, and field values are the output channels belonging 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, specified as a character vector. For example, `'system_1'`.

Default: `''`

`Notes`

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, `'System is MIMO'`.

Default: `{}`

`UserData`

Any type of data you want to associate with system, specified as any MATLAB data type.

Default: [ ]

`SamplingGrid`

Sampling grid for model arrays, specified as a data structure.

For arrays of identified linear (IDLTI) models that are derived by sampling one or more independent variables, this property tracks the variable values associated with each model. This information appears when you display or plot the model array. Use this information to trace results back to the independent variables.

Set the field names of the data structure to the names of the sampling variables. Set the field values to the sampled variable values associated with each model in the array. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, if you collect data at various operating points of a system, you can identify a model for each operating point separately and then stack the results together into a single system array. You can tag the individual models in the array with information regarding the operating point:

```
nominal_engine_rpm = [1000 5000 10000];
sys.SamplingGrid = struct('rpm', nominal_engine_rpm)
```

where `sys` is an array containing three identified models obtained at rpms 1000, 5000 and 10000, respectively.

For model arrays generated by linearizing a Simulink model at multiple parameter values or operating points, the software populates `SamplingGrid` automatically with the variable values that correspond to each entry in the array. For example, the Simulink Control Design commands `linearize` and `slLinearizer` populate `SamplingGrid` in this way.

Default: [ ]

More About

- "Specifying the Linear Grey-Box Model Structure"

See Also

`getpvec` | `greyest` | `greyestOptions` | `idnlgrey` | `idss` | `pem` | `setpvec` | `ssest`

Introduced before R2006a

idinput

Generate input signals

Syntax

```
u = idinput(N)
u = idinput(N,type,band,levels)
[u,freqs] = idinput(N,'sine',band,levels,sinedata)
```

Description

`u = idinput(N)` generates input signals which are typically used for identification. `N` determines the number of generated input data. `u` is returned as a matrix or column vector:

- 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`.

It is recommended that you create an `iddata` object from `u`, indicating sample time, input names, periodicity, and so on:

```
u = iddata([],u);
```

`u = idinput(N,type,band,levels)` specifies the type of input signal to be generated. This argument takes one of the following values:

- `'rgs'` — Gives a random, Gaussian signal.
- `'rbs'` — Gives a random, binary signal. This is the default.
- `'prbs'` — Gives a pseudorandom, binary signal.
- `'sine'` — Gives a signal that is a sum of sinusoids. The sinusoids are chosen from the frequency grid `freq = 2*pi*[1:Grid_Skip:fix(P/2)]/P` intersected with

`pi*[band(1) band(2)]`. For multi-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 `[u,freqs] = idinput(....)`, where row `ku` of `freqs` contains the frequencies of input number `ku`.

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.

`[u,freqs] = idinput(N,'sine',band,levels,sinedata)` specifies sine wave as the generated signal where `sinedata= [No_of_Sinusoids, No_of_Trials, Grid_Skip]`, meaning that `No_of_Sinusoids` are equally spread over the indicated `band`, trying `No_of_Trials` different, random, relative phases, until the lowest amplitude signal is found. Default value of `sindedata` is [10,10,1].

Some PRBS Aspects

If more than one period is demanded (that is, `M > 1`), the length of the data sequence and the period of the PRBS signal are adjusted so that an integer number of maximum length PRBS periods is always obtained. If `M = 1`, the period of the PRBS signal is chosen to that it is longer than `P = N`. In the multiple-input case, the signals are maximally shifted. This means `P/nu` is an upper bound for the model orders that can be estimated with such a signal.

Some Sine Aspects

In the `'sine'` case, the sinusoids are chosen from the frequency grid

```
freq = 2*pi*[1:Grid_Skip:fix(P/2)]/P
```

intersected with `pi*[band(1) band(2)]`. For `Grid_Skip`, see below. For multiple-input signals, the different inputs use different frequencies from this grid. An integer number of full periods is always delivered. The selected frequencies are obtained as the second output argument, `freqs`, where row `ku` of `freqs` contains the frequencies of input number `ku`. The resulting signal is affected by a fifth input argument, `sinedata`

```
sinedata = [No_of_Sinusoids, No_of_Trials, Grid_Skip]
```

meaning that `No_of_Sinusoids` is equally spread over the indicated `band`. `No_of_Trials` (different, random, relative phases) are tried until the lowest amplitude signal is found.

```
Default: sinedata = [10,10,1];
```

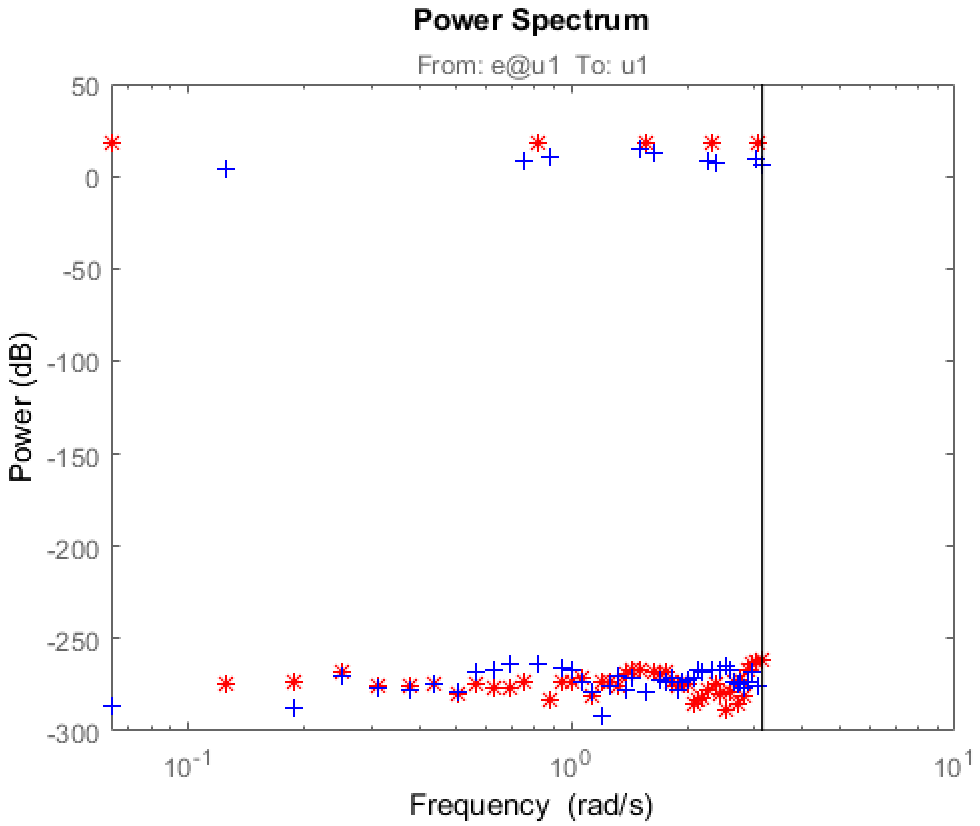
`Grid_Skip` can be useful for controlling odd and even frequency multiples, for example, to detect nonlinearities of various kinds.

Examples

Compare Spectrum of Sinusoid Signal with Square

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),'+')
```



More About

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.

Introduced before R2006a

idnlarx

Nonlinear ARX model

Syntax

```
sys = idnlarx(Orders)
sys = idnlarx(Orders,Nonlinearity)
sys = idnlarx(Orders,Nonlinearity,Name,Value)
sys = idnlarx(LinModel)
sys = idnlarx(LinModel,Nonlinearity)
sys = idnlarx(LinModel,Nonlinearity,Name,Value)
```

Description

`sys = idnlarx(Orders)` creates a nonlinear ARX model with the specified orders using a default wavelet network nonlinearity estimator.

`sys = idnlarx(Orders,Nonlinearity)` specifies a nonlinearity estimator for the model.

`sys = idnlarx(Orders,Nonlinearity,Name,Value)` specifies additional attributes of the `idnlarx` model structure using one or more `Name,Value` pair arguments.

`sys = idnlarx(LinModel)` uses a linear ARX model `LinModel` to specify the model orders and the initial values of the linear coefficients of the model.

`sys = idnlarx(LinModel,Nonlinearity)` specifies a nonlinearity estimator for the model.

`sys = idnlarx(LinModel,Nonlinearity,Name,Value)` specifies additional attributes of the `idnlarx` model structure using one or more `Name,Value` pair arguments.

Object Description

`idnlarx` represents a nonlinear ARX model, which 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".

Use the `nlarx` command to both construct an `idnlarx` object and estimate the model parameters.

You can also use the `idnlarx` constructor to create the nonlinear ARX model and then estimate the model parameters using `nlarx` or `pem`.

For `idnlarx` object properties, see "Properties" on page 1-518.

Examples

Create Nonlinear ARX Model with Default Wavelet Network Nonlinearity

```
m = idnlarx([2 2 1]);
```

Create and Configure Nonlinear ARX Model

Create a nonlinear ARX model with specific orders.

```
M = idnlarx([3 2 1]);
```

Configure the model with the following property settings:

- Sigmoid network nonlinearity with default settings
- Use inputs only as nonlinear regressors

```
M.Nonlinearity = 'sigmoidnet';
```

```
M.NonlinearRegressors = 'input';
```

Create Nonlinear ARX Model with Sigmoid Network Nonlinearity

```
m = idnlarx([2 3 1],sigmoidnet('NumberOfUnits',15));
```

Create Nonlinear ARX Model Without Nonlinear Function in Nonlinearity Estimator

```
m = idnlarx([2 2 1],[]);
```

Create Nonlinear ARX Model with Custom Regressors

Specify two custom regressors.

```
C = {'y1(t-1)^2','y1(t-2)*u1(t-3)'};
```

Create a nonlinear ARX model with custom regressors and no standard regressors.

```
sys = idnlarx([0 0 0],'wavenet','CustomRegressors',C);
```

Create Nonlinear ARX Model Using 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);
```

- “”
- “Estimate Nonlinear ARX Models Using Linear ARX Models”
- “Identifying Nonlinear ARX Models”
- “Using Linear Model for Nonlinear ARX Estimation”

Input Arguments

`Orders` — Model orders and delays

1-by-3 vector of positive integers | 1-by-3 vector of matrices

Model orders and delays for defining the regressor configuration, specified as a 1-by-3 vector, `[na nb nk]`.

For a model with n_y output channels and n_u input channels:

- `na` is an n_y-by-n_y matrix, where `na(i,j)` specifies the number of regressors from the *j*th output used to predict the *i*th output.
- `nb` is an n_y-by-n_u matrix, where `nb(i,j)` specifies the number of regressors from the *j*th input used to predict the *i*th output.
- `nk` is an n_y-by-n_u matrix, where `nk(i,j)` specifies the lag in the *j*th input used to predict the *i*th output.

```
na = [1 2; 2 3]
nb = [1 2 3; 2 3 1];
nk = [2 0 3; 1 0 5];
```

The estimation data for this system has three inputs (`u1`, `u2`, `u3`) and two outputs (`y1`, `y2`). Consider the regressors used to predict output, `y2(t)`:

- Since `na(2,:)` is `[2 3]`, the contributing regressors from the outputs are:
  - `y1(t-1)` and `y1(t-2)`
  - `y2(t-1)`, `y2(t-2)`, and `y2(t-3)`
- Since `nb(2,:)` is `[2 3 1]` and `nk(2,:)` is `[1 0 5]`, the contributing regressors from the inputs are:
  - `u1(t-1)` and `u1(t-2)`
  - `u2(t)`, `u2(t-1)`, and `u2(t-2)`
  - `u3(t-5)`

Note: The minimum lag for regressors based on output variables is always `1`, while the minimum lag for regressors based on input variables is dictated by `nk`. Use `getreg` to view the complete set of regressors used by the nonlinear ARX model.

`Nonlinearity` — Nonlinearity estimator

`'wavenet'` (default) | `'sigmoidnet'` | `'treepartition'` | `'linear'` | nonlinearity estimator object | array of nonlinearity estimator objects

Nonlinearity estimator, specified as one of the following:

`'wavenet'` or `wavenet` object	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 — Requires Neural Network Toolbox™.
`customnet` object	Custom network — Similar to `sigmoidnet`, but with a user-defined replacement for the sigmoid function.

For more information, see “Nonlinearity Estimators for Nonlinear ARX Models”.

Specifying a character vector, for example `'sigmoidnet'`, creates a nonlinearity estimator object with default settings. Alternatively, you can specify nonlinearity estimator settings in two ways:

- Use the associated nonlinearity estimator function with Name-Value pair arguments.

  ```
  NL = sigmoidnet('NumberOfUnits',10);
  ```
- Create and modify a default nonlinearity estimator object.

  ```
  NL = sigmoidnet;
  NL.NumberOfUnits = 10;
  ```

For `ny` output channels, you can specify nonlinear estimators individually for each channel by setting `Nonlinearity` to an `ny`-by-1 array of nonlinearity estimator objects. To specify the same nonlinearity for all outputs, specify `Nonlinearity` as a character vector or a single nonlinearity estimator object.

Example: `’sigmoidnet’` specifies a sigmoid network nonlinearity with a default configuration.

Example: `treepartition('NumberOfUnits',5)` specifies a binary-tree nonlinearity with 5 terms in the binary tree expansion.

Example: `[wavenet('NumberOfUnits',10);sigmoidnet]` specifies different nonlinearity estimators for two output channels.

`LinModel` — Discrete time input-output polynomial model of ARX structure

`idpoly` model

Discrete time input-output polynomial model of ARX structure, specified as an `idpoly` model. Create this object using the `idpoly` constructor or estimate it using the `arx` command.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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 `idnlarx` models during model creation. For example, `m = idnlarx([2 3 1],'treepartition','InputName','Pressure','Ts',0.1)` creates an `idnlarx` model with the input name `Pressure`, and a sample time of `0.1` seconds

Properties

`na, nb, nk`

Model orders and delays for defining the regressor configuration, specified as nonnegative integers.

For a model with n_y output channels and n_u input channels:

- `na` is an n_y-by-n_y matrix, where `na(i,j)` specifies the number of regressors from the *j*th output used to predict the *i*th output.
- `nb` is an n_y-by-n_u matrix, where `nb(i,j)` specifies the number of regressors from the *j*th input used to predict the *i*th output.
- `nk` is an n_y-by-n_u matrix, where `nk(i,j)` specifies the lag in the *j*th input used to predict the *i*th output.

```
na = [1 2; 2 3]
nb = [1 2 3; 2 3 1];
nk = [2 0 3; 1 0 5];
```

The estimation data for this system has three inputs (`u1`, `u2`, `u3`) and two outputs (`y1`, `y2`). Consider the regressors used to predict output, `y2(t)`:

- Since `na(2,:)` is `[2 3]`, the contributing regressors from the outputs are:

  - `y1(t-1)` and `y1(t-2)`
  - `y2(t-1)`, `y2(t-2)`, and `y2(t-3)`
- Since `nb(2,:)` is `[2 3 1]` and `nk(2,:)` is `[1 0 5]`, the contributing regressors from the inputs are:
  - `u1(t-1)` and `u1(t-2)`
  - `u2(t)`, `u2(t-1)`, and `u2(t-2)`
  - `u3(t-5)`

Note: The minimum lag for regressors based on output variables is always 1, while the minimum lag for regressors based on input variables is dictated by `nk`. Use `getreg` to view the complete set of regressors used by the nonlinear ARX model.

`CustomRegressors`

Regressors constructed from combinations of inputs and outputs, specified as one of the following:

- Cell array of character vectors. For example:
  - `{’y1(t-3)^3’,'y2(t-1)*u1(t-3)','sin(u3(t-2))'}`

  Each character vector must represent a valid formula for a regressor contributing towards the prediction of the model output. The formula must be written using the input and output names and the time-variable name as variables.
- Array of custom regressor objects, created using `customreg` or `polyreg`.

For a model with n_y outputs, specify an n_y-by-1 cell array of `customreg` objects arrays or an n_y-by-1 cell array of cell array of character vectors.

These regressors are in addition to the standard regressors based on `na`, `nb`, and `nk`.

Default: `{}`

`NonlinearRegressors`

Subset of regressors that enter as inputs to the nonlinear block of the model, specified as one of the following:

- `'all'` — All regressors
- `'output'` — Regressors containing output variables
- `'input'` — Regressors containing input variables
- `'standard'` — Standard regressors
- `'custom'` — Custom regressors
- `'search'` — The estimation algorithm performs a search for the best regressor combination. This option must be applied to all output models simultaneously.
- `[]` — No regressors
- Vector of regressor indices. To determine the number and order of regressors, use `getreg`.

For a model with multiple outputs, specify a cell array of n_y elements, where n_y is the number of output channels. For each output, specify one of the preceding options. Alternatively, to apply the same regressor subset to all model outputs, specify `[]` or any of the character vector options alone, for example `'standard'`.

Default: `'all'`

`Nonlinearity`

Nonlinearity estimator, specified as one of the following:

`'wavenet'` or `wavenet` object	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 — Requires Neural Network Toolbox.
`customnet` object	Custom network — Similar to `sigmoidnet`, but with a user-defined replacement for the sigmoid function.

For more information, see “Nonlinearity Estimators for Nonlinear ARX Models”.

Specifying a character vector, for example `'sigmoidnet'`, creates a nonlinearity estimator object with default settings. Alternatively, you can specify nonlinearity estimator settings in two ways:

- Use the associated nonlinearity estimator function with Name-Value pair arguments:

  ```
  NL = sigmoidnet('NumberOfUnits',10);
  ```
- Create and modify a default nonlinearity estimator object:

  ```
  NL = sigmoidnet;
  NL.NumberOfUnits = 10;
  ```

For `ny` output channels, you can specify nonlinear estimators individually for each channel by setting `Nonlinearity` to an array of n_y nonlinearity estimator objects, where n_y is the number of outputs. To specify the same nonlinearity for all outputs, specify `Nonlinearity` as a character vector or a single nonlinearity estimator object.

Default: `'wavenet'`

Report

Summary report that contains information about the estimation options and results when the model is estimated using the `nlarx` command. Use `Report` to query a model for how it was estimated, including its:

- Estimation method
- Estimation options
- Search termination conditions
- Estimation data fit

The contents of `Report` are irrelevant if the model was constructed.

```
m = idnlarx([2 2 1]);
m.Report.OptionsUsed
```

```
ans =

     []
```

If you use `nlarx` to estimate the model, the fields of `Report` contain information on the estimation data, options, and results.

```
load iddata1;
m = nlarx(z1, [2 2 1]);
m.Report.OptionsUsed
```

```
Option set for the nlarx command:
```

```
        IterWavenet: 'auto'
              Focus: 'prediction'
            Display: 'off'
     Regularization: [1x1 struct]
       SearchMethod: 'auto'
       SearchOption: [1x1 idoptions.search.identsolver]
       OutputWeight: 'noise'
           Advanced: [1x1 struct]
```

`Report` is a read-only property.

For more information on this property and how to use it, see "Output Arguments" on page 1-829 in the `nlarx` reference page and "Estimation Report".

`TimeVariable`

Independent variable for the inputs, outputs, and—when available—internal states, specified as a character vector.

Default: `'t'` (time)

`NoiseVariance`

Noise variance (covariance matrix) of the model innovations *e*.
Assignable value is an `ny`-by-`ny` matrix.
Typically set automatically by the estimation algorithm.

`Ts`

Sample time. `Ts` is a positive scalar representing the sampling period. This value is expressed in the unit specified by the `TimeUnit` property of the model.

Default: `1`

`TimeUnit`

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as one of the following values:

- `'nanoseconds'`
- `'microseconds'`

- `'milliseconds'`
- `'seconds'`
- `'minutes'`
- `'hours'`
- `'days'`
- `'weeks'`
- `'months'`
- `'years'`

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

Default: `'seconds'`

InputName

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, `'controls'`.
- Cell array of character vectors — For multi-input models.

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 input names automatically expand to `{'controls(1)';'controls(2)'}`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `InputName` to `data.InputName`.

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: `''` for all input channels

`InputUnit`

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, `'seconds'`.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

Default: `''` 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. In this structure, field names are the group names, and field values are the input channels belonging 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, specified as one of the following:

- Character vector — For single-output models. For example, `'measurements'`.
- Cell array of character vectors — For multi-output models.

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 output names automatically expand to `{'measurements(1)';'measurements(2)'}`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `OutputName` to `data.OutputName`.

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: `''` for all output channels

`OutputUnit`

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, `'seconds'`.
- Cell array of character vectors — For multi-output models.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

Default: `''` for all output 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. In this structure, field names are the group names, and field values are the output channels belonging 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, specified as a character vector. For example, `'system_1'`.

Default: `''`

`Notes`

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, `'System is MIMO'`.

Default: `{}`

`UserData`

Any type of data you want to associate with system, specified as any MATLAB data type.

Default: `[]`

Output Arguments

`sys` — Nonlinear ARX model

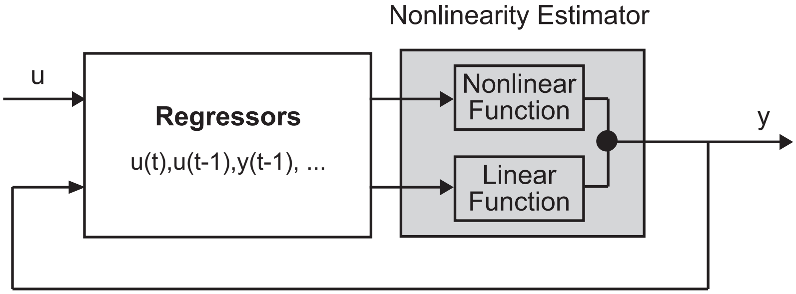
`idnlarx` object

Nonlinear ARX model, returned as an `idnlarx` object. This model is created using the specified model orders, nonlinearity estimator, and properties.

More About

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 multilayer neural networks. You can also exclude either the linear or the nonlinear function block from the nonlinearity estimator.

The nonlinearity estimator block can include linear and nonlinear blocks in parallel. For example:

$$F(x) = L^T(x-r) + d + g\left(Q(x-r)\right)$$

x is a vector of the regressors. $L^T(x)+d$ is the output of the linear function block and is affine when d≠0. d is a scalar offset. $g\left(Q(x-r)\right)$ represents the output of the nonlinear function block. r is the mean of the regressors x. Q is a projection matrix that makes the calculations well conditioned. The exact form of $F(x)$ depends on your choice of the nonlinearity estimator.

Estimating a nonlinear ARX model computes the model parameter values, such as L, r, d, Q, and other parameters specifying g. Resulting models are `idnlarx` objects that store all model data, including model regressors and parameters of the nonlinearity estimator. See the `idnlarx` reference page for more information.

Definition of idnlarx States

The states of an `idnlarx` object are an ordered list of delayed input and output variables that define the structure of the model. The toolbox uses this definition of states for creating the initial state vector that is used during simulation and prediction with `sim`, `predict`, and `compare`. This definition is also used for linearization of nonlinear ARX models using `linearize`.

This toolbox provides several options to facilitate how you specify the initial states. For example, you can use `findstates` and `data2state` to 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 depend on 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: $p(t\text{-}1)$, $p(t\text{-}2)$, ..., $p(t\text{-}D)$.

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 `y1` is 10 samples and the maximum delay in the input `u1` is two samples. Thus, this model has a total of 12 states:

```
X(t) = [y1(t-1),y2(t-2),…,y1(t-10),u1(t-1),u1(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]);
```

This model has 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 `y1` is two samples. This delay occurs in the 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) = [y1(t-1), y1(t-2), u1(t-1), u1(t-2), u2(t-1),
            u2(t-2), u2(t-3), u2(t-4), u2(t-5)]
```

Variables `y2` and `u3` 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 (`y1`, `y2`, `u1`, `u2`, `u3`). The total number of model states is `sum(maxDel) = 9`.

The set of states for an `idnlarx` model is not required to be minimal.

See Also

`addreg` | `customnet` | `customreg` | `getreg` | `idnlarx/findop` | `linear` | `linearize` | `nlarx` | `pem` | `polyreg` | `sigmoidnet` | `wavenet`

Introduced in R2007a

idnlgrey

Nonlinear grey-box model

Syntax

```
sys = idnlgrey(FileName,Order,Parameters)
sys = idnlgrey(FileName,Order,Parameters,InitialStates)
sys = idnlgrey(FileName,Order,Parameters,InitialStates,Ts)
sys = idnlgrey(FileName,Order,Parameters,InitialStates,Ts,
Name,Value)
```

Description

`sys = idnlgrey(FileName,Order,Parameters)` creates a nonlinear grey-box model using the specified model structure in `FileName`, number of outputs, inputs, and states in `Order`, and the model parameters.

`sys = idnlgrey(FileName,Order,Parameters,InitialStates)` specifies the initial states of the model.

`sys = idnlgrey(FileName,Order,Parameters,InitialStates,Ts)` specifies the sample time of a discrete-time model.

`sys = idnlgrey(FileName,Order,Parameters,InitialStates,Ts, Name,Value)` specifies additional attributes of the `idnlgrey` model structure using one or more `Name,Value` pair arguments.

Object Description

`idnlgrey` represents a nonlinear grey-box model. For information about the nonlinear grey-box models, see “Estimate Nonlinear Grey-Box Models”.

Use the `idnlgrey` constructor to create the nonlinear grey-box model and then estimate the model parameters using `nlgreyest`.

For `idnlgrey` object properties, see “Properties” on page 1-538.

Examples

Create a Nonlinear Grey-Box Model

Load data.

```
load(fullfile(matlabroot,'toolbox','ident','iddemos','data','dcmotordata'));
z = iddata(y,u,0.1,'Name','DC-motor');
```

The data is from a linear DC motor with one input (voltage), and two outputs (angular position and angular velocity). The structure of the model is specified by `dcmotor_m.m` file.

Create a nonlinear grey-box model.

```
file_name = 'dcmotor_m';
Order = [2 1 2];
Parameters = [1;0.28];
InitialStates = [0;0];

sys = idnlgrey(file_name,Order,Parameters,InitialStates,0, ...
    'Name','DC-motor');
```

Selectively Estimate Parameters of Nonlinear Grey-Box Model

Load data.

```
load(fullfile(matlabroot,'toolbox','ident','iddemos','data','twotankdata'));
z = iddata(y,u,0.2,'Name','Two tanks');
```

The data contains 3000 input-output data samples of a two tank system. The input is the voltage applied to a pump, and the output is the liquid level of the lower tank.

Specify file describing the model structure for a two-tank system. The file specifies the state derivatives and model outputs as a function of time, states, inputs, and model parameters.

```
FileName = 'twotanks_c';
```

Specify model orders [ny nu nx].

```
Order = [1 1 2];
```

Specify initial parameters (Np = 6).

```
Parameters = {0.5;0.0035;0.019; ...
    9.81;0.25;0.016};
```

Specify initial initial states.

```
InitialStates = [0;0.1];
```

Specify as continuous system.

```
Ts = 0;
```

Create `idnlgrey` model object.

```
nlgr = idnlgrey(FileName,Order,Parameters,InitialStates,Ts, ...
    'Name','Two tanks');
```

Set some parameters as constant.

```
nlgr.Parameters(1).Fixed = true;
nlgr.Parameters(4).Fixed = true;
nlgr.Parameters(5).Fixed = true;
```

Estimate the model parameters.

```
nlgr = nlgreyest(z,nlgr);
```

- “Represent Nonlinear Dynamics Using MATLAB File for Grey-Box Estimation”
- “Creating IDNLGREY Model Files”

Input Arguments

`FileName` — Name of the function or MEX-file that stores the model structure

character vector | function handle

Name of the function or MEX-file storing the model structure, specified as a character vector (without the file extension) or a function handle for computing the states and the outputs. If `FileName` is a character vector, for example `'twotanks_c'`, 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”.

`Order` — Number of outputs, inputs, and states of the model

vector | structure

Number of outputs, inputs, and states of the model, specified as one of the following:

- Vector `[Ny Nu Nx]`, specifying the number of model outputs `Ny`, inputs `Nu`, and states `Nx`.
- Structure with fields `'Ny'`, `'Nu'`, and `'Nx'`.

For time series, `Nu` is set to `0`, and for static model structures, `Nx` is set to `0`.

`Parameters` — Parameters of the model

structure | vector | cell array

Parameters of the model, specified as one of the following:

- `Np`-by-1 structure array, where `Np` is the number of parameters. The structure contains the following fields:

Field	Description	Default
`Name`	Name of the parameter, specified as a character vector. For example, `'pressure'`.	`'pi'`, where `i` is an integer in `[1,Np]`
`Unit`	Unit of the parameter, specified as a character vector.	`''`
`Value`	Initial value of the parameter, specified as: • Finite real scalar • Finite real column vector • Two-dimensional real matrix	
`Minimum`	Minimum value of the parameter, specified as a real scalar, column vector, or matrix of the same size as `Value`. `Minimum` >= `Value` for all components.	`-Inf(size(Value))`

Field	Description	Default
`Maximum`	Maximum value of the parameter, specified as a real scalar, column vector, or matrix of the same size as `Value`. `Value` <= `Maximum` for all components.	`Inf(size(Value))`
`Fixed`	Specifies whether parameter is fixed to their initial values, specified as a boolean scalar, column vector, or matrix of the same size as `Value`.	`false(size(Value))` — Implies, estimate all parameters

Use dot notation to access the subfields of the `i`th parameter. For example, for `idnlgrey` model `M`, the `i`th parameter is accessed through `M.Parameters(i)` and its subfield `Fixed` by `M.Parameters(i).Fixed`.

- `Np`-by-1 vector of real finite initial values, `InParameters`.

  The data is converted into a structure with default values for the fields `Name`, `Unit`, `Minimum`, `Maximum`, and `Fixed`.

  `Value` is assigned the value `InParameters(i)`, where `i` is an integer in `[1,Np]`
- `Np`-by-1 cell array containing finite real scalars, finite real vectors, or finite real two-dimensional matrices of initial values.

  Default values are used for the fields `Name`, `Unit`, `Minimum`, `Maximum`, and `Fixed`.

`InitialStates` — Initial states of the model

structure | `[]` | cell array | `{}`

Initial states of the model parameters specified as one of the following:

- `Nx`-by-1 structure array, where `Nx` is the number of states. The structure contains the following fields:

Field	Description	Default
`Name`	Name of the states, specified as a character vector.	`'xi'`, where `i` is an integer in `[1,Nx]`
`Unit`	Unit of the states, specified as a character vector.	`''`
`Value`	Initial value of the initial states, specified as: • A finite real scalar • A finite real 1-by-`Ne` vector, where `Ne` is the number of experiments in the data set to be used for estimation	
`Minimum`	Minimum value of the initial states, specified as a real scalar or 1-by-`Ne` vector of the same size as `Value`. `Minimum` >= `Value` for all components.	`-Inf(size(Value))`
`Maximum`	Maximum value of the parameters, specified as a real scalar or 1-by-`Ne` vector of the same size as `Value`. `Value` <= `Maximum` for all components.	`Inf(size(Value))`

Field	Description	Default
`Fixed`	Specifies whether initial states are fixed to their initial values, specified as boolean scalar or 1-by-`Ne` vector of the same size as `Value`	`true(size(Value))` — Implies, do not estimate the initial states.

Use dot notation to access the subfields of the `i`th initial state. For example, for `idnlgrey` model `M`, the `i`th initial state is accessed through `M.InitialStates(i)` and its subfield `Fixed` by `M.InitialStates(i).Fixed`.

- `[]`.

  A structure is created with default values for the fields `Name`, `Unit`, `Minimum`, `Maximum`, and `Fixed`.

  `Value` is assigned the value `0`.
- A real finite `Nx`-by-`Ne` matrix (`InitStates`).

  `Value` of the `i`th structure array element is `InitStates(i,Ne)`, a row vector with `Ne` elements. `Minimum`, `Maximum`, and `Fixed` will be `-Inf`, `Inf` and `true` row vectors of the same size as `InitStates(i,Ne)`.
- Cell array with finite real vectors of size 1-by-`Ne` or `{}` (same as `[]`).

`Ts` — Sample time

`0` (default) | scalar

Sample time, specified as a positive scalar representing the sampling period. The value is expressed in the unit specified by the `TimeUnit` property of the model. For a continuous time model `Ts` is equal to 0 (default).

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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 `idnlgrey` models during model creation.

Properties

`idnlgrey` object properties include:

`FileName`

Name of the function or MEX-file storing the model structure, specified as a character vector (without extension) or a function handle for computing the states and the outputs. If `FileName` is a character vector, for example `'twotanks_c'`, 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”.

`Order`

Number of outputs, inputs, and states of the model, specified as one of the following:

- Vector `[Ny Nu Nx]`, specifying the number of model outputs `Ny`, inputs `Nu`, and states `Nx`.
- Structure with fields`'Ny'`, `'Nu'`, and `'Nx'`.

For time series, `Nu` is set to `0`, and for static model structures, `Nx` is set to `0`.

`Parameters`

Parameters of the model, specified as one of the following:

- `Np`-by-1 structure array, where `Np` is the number of parameters. The structure contains the following fields:

Field	Description	Default
`Name`	Name of the parameter, specified as a character vector. For example, `'pressure'`.	`'pi'`, where `i` is an integer in `[1,Np]`
`Unit`	Unit of the parameter, specified as a character vector.	`''`
`Value`	Initial value of the parameter, specified as: • Finite real scalar	

Field	Description	Default
	• Finite real column vector • Two-dimensional real matrix	
`Minimum`	Minimum value of the parameter, specified as a real scalar, column vector, or matrix of the same size as `Value`. `Minimum` >= `Value` for all components.	`-Inf(size(Value))`
`Maximum`	Maximum value of the parameter, specified as a real scalar, column vector, or matrix of the same size as `Value`. `Value` <= `Maximum` for all components.	`Inf(size(Value))`
`Fixed`	Specifies whether parameter is fixed to their initial values, specified as a boolean scalar, column vector, or matrix of the same size as `Value`.	`false(size(Value))` — Implies, estimate all parameters

- `Np`-by-1 vector of real finite initial values, `InParameters`.

  The data is converted into a structure with default values for the fields `Name`, `Unit`, `Minimum`, `Maximum`, and `Fixed`.

  `Value` is assigned the value `InParameters(i)`, where `i` is an integer in `[1,Np]`
- `Np`-by-1 cell array containing finite real scalars, finite real vectors, or finite real two-dimensional matrices of initial values.

  A structure is created with default values for the fields `Name`, `Unit`, `Minimum`, `Maximum`, and `Fixed`.

Use dot notation to access the subfields of the `i`th parameter. For example, for `idnlgrey` model `M`, the `i`th parameter is accessed through `M.Parameters(i)` and its subfield `Fixed` by `M.Parameters(i).Fixed`.

`InitialStates`

Initial states of the model parameters specified as one of the following:

- `Nx`-by-1 structure array, where `Nx` is the number of states. The structure contains the following fields:

Field	Description	Default
`Name`	Name of the states, specified as a character vector.	`'xi'`, where `i` is an integer in `[1,Nx]`
`Unit`	Unit of the states, specified as a character vector.	`''`
`Value`	Initial value of the initial states, specified as: • A finite real scalar • A finite real 1-by-`Ne` vector, where `Ne` is the number of experiments in the data set to be used for estimation	
`Minimum`	Minimum value of the initial states, specified as a real scalar or 1-by-`Ne` vector of the same size as `Value`. `Minimum` >= `Value` for all components.	`-Inf(size(Value))`
`Maximum`	Maximum value of the parameters, specified as a real scalar or 1-by-`Ne` vector of the same size as `Value`.	`Inf(size(Value))`

Field	Description	Default
	`Value` <= `Maximum` for all components.	
`Fixed`	Specifies whether initial states are fixed to their initial values, specified as boolean scalar or 1-by-`Ne` vector of the same size as `Value`	`true(size(Value))` — Implies, do not estimate the initial states.

- `[]`.

  A structure is created with default values for the fields `Name`, `Unit`, `Minimum`, `Maximum`, and `Fixed`.

  `Value` is assigned the value `0`.
- A real finite `Nx`-by-`Ne` matrix (`InitStates`).

  `Value` of the `i`th structure array element is `InitStates(i,Ne)`, a row vector with `Ne` elements. `Minimum`, `Maximum`, and `Fixed` will be `-Inf`, `Inf` and `true` row vectors of the same size as `InitStates(i,Ne)`.
- Cell array with finite real vectors of size 1-by-`Ne` or `{}` (same as `[]`).

  A structure is created with default values for the fields `Name`, `Unit`, `Minimum`, `Maximum`, and `Fixed`.

Use dot notation to access the subfields of the `i`th initial state. For example, for `idnlgrey` model `M`, the `i`th initial state is accessed through `M.InitialStates(i)` and its subfield `Fixed` by `M.InitialStates(i).Fixed`.

FileArgument

Contains auxiliary variables passed to the ODE file (function or MEX-file) specified in `FileName`, specified as a cell array. These variables are used as extra inputs for specifying the state and/or output equations.
Default: `{}`.

`SimulationOptions`

A structure that specifies the simulation method and related options, containing the following fields:

Field	Description	Default
`AbsTol`	Absolute error tolerance. This scalar applies to all components of the state vector. Applicable to: Variable step solvers. Assignable value: A positive real value.	`1e-6`
`FixedStep`	Step size used by the solver. Applicable to: Fixed-step time-continuous solvers. Assignable values: • `'Auto'` — Automatically chooses the initial step. • A real value such that `0<FixedStep<=1`.	`'Auto'` — Automatically chooses the initial step.
`InitialStep`	Specifies the initial step at which the ODE solver starts. Applicable to: Variable-step, time-continuous solvers. Assignable values: • `'Auto'` — Automatically chooses the initial step. • A positive real value such that `MinStep<=InitialStep<=MaxStep`.	`'Auto'` — Automatically chooses the initial step.
`MaxOrder`	Specifies the order of the Numerical Differentiation Formulas (NDF). Applicable to: ode15s. Assignable values: 1, 2, 3, 4 or 5.	`5`
`MaxStep`	Specifies the largest time step of the ODE solver. Applicable to: Variable-step, time-continuous solvers. Assignable values:	`'Auto'` — Automatically chooses the time step.

Field	Description	Default
	- `'Auto'` — Automatically chooses the time step. - A positive real value > `MinStep`.	
`MinStep`	Specifies the smallest time step of the ODE solver. Applicable to: Variable-step, time-continuous solvers. Assignable values: - `'Auto'` — Automatically chooses the time step. - A positive real value < `MaxStep`.	`'Auto'` — Automatically chooses the time step.
`RelTol`	Relative error tolerance that applies to all components of the state vector. The estimated error in each integration step satisfies `|e(i)| <= max(RelTol*abs(x(i)), AbsTol(i))`. Applicable to: Variable-step, time-continuous solvers. Assignable value: A positive real value.	`1e-3` (0.1% accuracy).

Field	Description	Default

`Solver`	ODE (Ordinary Differential/Difference Equation) solver for solving state space equations.
	• Variable-step solvers for time-continuous `idnlgrey` models:
	  • `'ode45'` — Runge-Kutta (4,5) solver for nonstiff problems.
	  • `'ode23'` — Runge-Kutta (2,3) solver for nonstiff problems.
	  • `'ode113'` — Adams-Bashforth-Moulton solver for nonstiff problems.
	  • `'ode15s'` — Numerical Differential Formula solver for stiff problems.
	  • `'ode23s'` — Modified Rosenbrock solver for stiff problems.
	  • `'ode23t'` — Trapezoidal solver for moderately stiff problems.
	  • `'ode23tb'` — Implicit Runge-Kutta solver for stiff problems.
	• Fixed-step solvers for time-continuous `idnlgrey` models:
	  • `'ode5'` — Dormand-Prince solver.
	  • `'ode4'` — Fourth-order Runge-Kutta solver.
	  • `'ode3'` — Bogacki-Shampine solver.
	  • `'ode2'` — Heun or improved Euler solver.
	  • `'ode1'` — Euler solver.
	• Fixed-step solvers for time-discrete `idnlgrey` models: `'FixedStepDiscrete'`
		`'Auto'`
		— Automatically chooses one of the solvers.

Field	Description	Default
	• General: `'Auto'` — Automatically chooses one of the previous solvers.	

`Report`

Summary report that contains information about the estimation options and results when the model is estimated using the `nlgreyest` command. Use `Report` to query a model for how it was estimated, including:

- Estimation method
- Estimation options
- Search termination conditions
- Estimation data fit

The contents of `Report` are irrelevant if the model was created by construction.

```
nlgr = idnlgrey('dcmotor_m',[2,1,2],[1;0.28],[0;0],0,'Name','DC-motor');
nlgr.Report.OptionsUsed
```

```
ans =

     []
```

If you use `nlgreyest` to estimate the model, the fields of `Report` contain information on the estimation data, options, and results.

```
load(fullfile(matlabroot,'toolbox','ident','iddemos','data','dcmotordata'));
z = iddata(y,u,0.1,'Name','DC-motor');
nlgr = idnlgrey('dcmotor_m',[2,1,2],[1;0.28],[0;0],0,'Name','DC-motor');
nlgr = nlgreyest(z,nlgr);
nlgr.Report.OptionsUsed
```

```
Option set for the nlgreyest command:

    GradientOptions: [1x1 struct]
           EstCovar: 1
            Display: 'off'
     Regularization: [1x1 struct]
       SearchMethod: 'auto'
       SearchOption: [1x1 idoptions.search.lsqnonlin]
       OutputWeight: []
           Advanced: [1x1 struct]
```

`Report` is a read-only property.

For more information on this property and how to use it, see “Output Arguments” on page 1-845 in the `nlgreyest` reference page and “Estimation Report”.

`TimeVariable`

Independent variable for the inputs, outputs, and—when available—internal states, specified as a character vector.

Default: `'t'`

`NoiseVariance`

Noise variance (covariance matrix) of the model innovations *e*.
Assignable value is an `ny`-by-`ny` matrix.
Typically set automatically by the estimation algorithm.

`Ts`

Sample time. `Ts` is a positive scalar representing the sampling period. This value is expressed in the unit specified by the `TimeUnit` property of the model. For a continuous time model, `Ts` is equal to 0 (default).

Changing this property does not discretize or resample the model.

Default: `0`

`TimeUnit`

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as one of the following values:

- `'nanoseconds'`
- `'microseconds'`
- `'milliseconds'`
- `'seconds'`
- `'minutes'`
- `'hours'`
- `'days'`
- `'weeks'`

- `'months'`
- `'years'`

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

Default: `'seconds'`

`InputName`

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, `'controls'`.
- Cell array of character vectors — For multi-input models.

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 input names automatically expand to `{'controls(1)';'controls(2)'}`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `InputName` to `data.InputName`.

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: `''` for all input channels

`InputUnit`

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, `'seconds'`.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

Default: `''` 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. In this structure, field names are the group names, and field values are the input channels belonging 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, specified as one of the following:

- Character vector — For single-output models. For example, `'measurements'`.
- Cell array of character vectors — For multi-output models.

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 output names automatically expand to `{'measurements(1)';'measurements(2)'}`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `OutputName` to `data.OutputName`.

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: `''` for all output channels

`OutputUnit`

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, `'seconds'`.
- Cell array of character vectors — For multi-output models.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

Default: `''` for all output 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. In this structure, field names are the group names, and field values are the output channels belonging 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, specified as a character vector. For example, `'system_1'`.

Default: `''`

`Notes`

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, `'System is MIMO'`.

Default: `{}`

`UserData`

Any type of data you want to associate with system, specified as any MATLAB data type.

Default: `[]`

Output Arguments

`sys` — Nonlinear grey-box model
`idnlgrey` object

Nonlinear grey-box model, returned as an `idnlgrey` object.

More About

Definition of idnlgrey States

The states of an `idnlgrey` model are defined explicitly in the function or MEX-file storing the model structure. States are required for simulation and prediction of nonlinear grey-box models. Use `findstates` to search for state values for simulation and prediction with `sim`, `predict`, and `compare`.

Note: The initial values of the states are configured by the `InitialStates` property of the `idnlgrey` model.

- “Estimate Nonlinear Grey-Box Models”

See Also

`get` | `getinit` | `getpar` | `nlgreyest` | `pem` | `set` | `setinit` | `setpar`

Introduced in R2007a

idnlhw

Hammerstein-Wiener model

Syntax

```
sys = idnlhw(Orders)
sys = idnlhw(Orders,InputNL,OutputNL)
sys = idnlhw(Orders,InputNL,OutputNL,Name,Value)
sys = idnlhw(LinModel)
sys = idnlhw(LinModel,InputNL,OutputNL)
sys = idnlhw(LinModel,InputNL,OutputNL,Name,Value)
```

Description

`sys = idnlhw(Orders)` creates a Hammerstein-Wiener model with the specified orders, and using piecewise linear functions as input and output nonlinearity estimators.

`sys = idnlhw(Orders,InputNL,OutputNL)` uses `InputNL` and `OutputNL` as the input and output nonlinearity estimators, respectively.

`sys = idnlhw(Orders,InputNL,OutputNL,Name,Value)` specifies additional attributes of the `idnlhw` model structure using one or more `Name,Value` pair arguments.

`sys = idnlhw(LinModel)` uses a linear model `LinModel` to specify the model orders and default piecewise linear functions for the input and output nonlinearity estimators.

`sys = idnlhw(LinModel,InputNL,OutputNL)` specifies input and output nonlinearity estimators for the model.

`sys = idnlhw(LinModel,InputNL,OutputNL,Name,Value)` specifies additional attributes of the `idnlhw` model structure using one or more `Name,Value` pair arguments.

Object Description

`idnlhw` represents a Hammerstein-Wiener model. The Hammerstein-Wiener structure represents a linear model with input-output nonlinearities.

Use the `nlhw` command to both construct an `idnlhw` object and estimate the model parameters.

You can also use the `idnlhw` constructor to create the Hammerstein-Wiener model and then estimate the model parameters using `nlhw`.

For `idnlhw` object properties, see “Properties” on page 1-557.

Examples

Create a Hammerstein-Wiener Model Structure with Default Nonlinearities

Create a Hammersein-Wiener model with `nb` and `nf` = 2 and `nk` = 1.

```
 m = idnlhw([2 2 1]);
```

`m` has piecewise linear input and output nonlinearity.

Create Hammerstein-Wiener Model with Specific Input-Output Nonlinearities

```
m = idnlhw([2 2 1],'sigmoidnet','deadzone');
```

The above is equivalent to:

```
m = idnlhw([2 2 1],'sig','dead');
```

The specified nonlinearities have a default configuration.

Create Hammerstein-Wiener Model and Configure the Nonlinearities

```
m = idnlhw([2 2 1],sigmoidnet('num',5),deadzone([-1,2]),'InputName','Volts','OutputName
```

Create a Wiener Model and Estimate Model Parameters

Create a Wiener model (no input nonlinearity).

```
m = idnlhw([2 2 1],[],'saturation');
```

Estimate the model.

```
load iddata1;
m = nlhw(z1,m);
```

Create Hammerstein-Wiener Model Using 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',[],'InputName','Control');
```

- “”
- “Estimate Hammerstein-Wiener Models Using Linear OE Models”

Input Arguments

`Orders` — Order and delays of the linear subsystem transfer function
`[nb nf nk]` vector of positive integers | `[nb nf nk]` vector of matrices

Order and delays of the linear subsystem transfer function, specified as a `[nb nf nk]` vector.

Dimensions of `Orders`:

- For a SISO transfer function, `Orders` is a vector of positive integers.

  `nb` is the number of zeros plus 1, `nf` is the number of poles, and `nk` is the input delay.
- For a MIMO transfer function with n_u inputs and n_y outputs, `Orders` is a vector of matrices.

  `nb`, `nf`, and `nk` are n_y-by-n_u 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` — Input static nonlinearity
`'pwlinear'` (default) | `'sigmoidnet'` | `'wavenet'` | `'saturation'` | `'deadzone'` | `'poly1d'` | `'unitgain'` | nonlinearity estimator object | array of nonlinearity estimators

Input static nonlinearity estimator, specified as one of the following.

`'pwlinear'` or `pwlinear` 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 `[]` or `unitgain` object	Unit gain
`customnet` object	Custom network — Similar to `sigmoidnet`, but with a user-defined replacement for the sigmoid function.

Specifying a character vector, for example `'sigmoidnet'`, creates a nonlinearity estimator object with default settings. Use object representation instead to configure the properties of a nonlinearity estimator.

```
InputNL = wavenet;
InputNL.NumberOfUnits = 10;
```

Alternatively, use the associated input nonlinearity estimator function with Name-Value pair arguments.

```
InputNL = wavenet('NumberOfUnits',10);
```

For n_u input channels, you can specify nonlinear estimators individually for each input channel by setting `InputNL` to an n_u-by-1 array of nonlinearity estimators.

```
InputNL = [sigmoidnet('NumberofUnits',5); deadzone([-1,2])]
```

To specify the same nonlinearity for all inputs, specify a single input nonlinearity estimator.

`OutputNL` — Output static nonlinearity

`'pwlinear'` (default) | `'sigmoidnet'` | `'wavenet'` | `'saturation'` | `'deadzone'` | `'poly1d'` | `'unitgain'` | nonlinearity estimator object | array of nonlinearity estimators

Output static nonlinearity estimator, specified as one of the following:

`'pwlinear'` or `pwlinear` 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 `[]` or `unitgain` object	Unit gain
`customnet` object	Custom network — Similar to `sigmoidnet`, but with a user-defined replacement for the sigmoid function.

Specifying a character vector creates a nonlinearity estimator object with default settings. Use object representation instead to configure the properties of a nonlinearity estimator.

```
OutputNL = sigmoidnet;
OutputNL.NumberOfUnits = 10;
```

Alternatively, use the associated input nonlinearity estimator function with Name-Value pair arguments.

```
OutputNL = sigmoidnet('NumberOfUnits',10);
```

For n_y output channels, you can specify nonlinear estimators individually for each output channel by setting `OutputNL` to an n_y-by-1 array of nonlinearity estimators. To specify the same nonlinearity for all outputs, specify a single output nonlinearity estimator.

`LinModel` — Discrete time linear model

`idpoly` | `idss` with `K = 0` | `idtf`

Discrete-time linear model used to specify the linear subsystem, specified as one of the following:

- Input-output polynomial model of Output-Error (OE) structure (`idpoly`)
- State-space model with no disturbance component (`idss` with `K = 0`)
- Transfer function model (`idtf`)

Typically, you estimate the model using `oe`, `n4sid`, or `tfest`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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 `idnlhw` models during model creation. For example, `m = idnlhw([2 3 1],'pwlinear','wavenet','InputName','Volts','Ts',0.1)` creates an `idnlhw` model object with input nonlinearity estimator `'pwlinear'`, output nonlinearity estimator `'wavenet'`, input name `Volts`, and a sample time of `0.1` seconds.

Properties

`idnlhw` object properties include:

`nb, nf, nk`

Model orders and delays of the linear subsystem transfer function, where `nb` is the number of zeros plus 1, `nf` is the number of poles, and `nk` is the input delay.

For a MIMO transfer function with n_u inputs and n_y outputs, `nb`, `nf`, and `nk` are n_y-by-n_u 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.

`B`

B polynomial of the linear block in the model structure, specified 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 `B{i,j}` is a row vector representing the numerator polynomial for the `j`th input to `i`th output transfer function. The element contains `nk` leading zeros, where `nk` is the number of input delays.

`F`

F polynomial of the linear block in the model structure, specified 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 `F{i,j}` is a row vector representing the denominator polynomial for the `j`th input to `i`th output transfer function.

`InputNonlinearity`

Input nonlinearity estimator, specified as one of the following:

`'pwlinear'` or `pwlinear` 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 `[ ]` or `unitgain` object	Unit gain
`customnet` object	Custom network

Specifying a character vector creates a nonlinearity estimator object with default settings. Use object representation instead to configure the properties of a nonlinearity estimator.

```
InputNonlinearity = wavenet;
InputNonlinearity.NumberOfUnits = 10;
```

Alternatively, use the associated input nonlinearity estimator function with Name-Value pair arguments.

```
InputNonlinearity = wavenet('NumberOfUnits',10);
```

For n_u input channels, you can specify nonlinear estimators individually for each input channel by setting `InputNL` to an n_u-by-1 array of nonlinearity estimators. To specify the same nonlinearity for all inputs, specify a single input nonlinearity estimator.

Default: `'pwlinear'`

`OutputNonlinearity`

Output nonlinearity estimator, specified as one of the following:

`'pwlinear'` or `pwlinear` 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 `[]` or `unitgain` object	Unit gain
`customnet` object	Custom network

Specifying a character vector creates a nonlinearity estimator object with default settings. Use object representation instead to configure the properties of a nonlinearity estimator.

```
OutputNonlinearity = sigmoidnet;
OutputNonlinearity.NumberOfUnits = 10;
```

Alternatively, use the associated input nonlinearity estimator function with Name-Value pair arguments.

```
OutputNonlinearity = sigmoidnet('NumberOfUnits',10);
```

For n_y output channels, you can specify nonlinear estimators individually for each output channel by setting `OutputNL` to an n_y-by-1 array of nonlinearity estimators. To specify the same nonlinearity for all outputs, specify a single output nonlinearity estimator.

Default: `'pwlinear'`

`LinearModel`

The linear model in the linear block of the model structure, represented as an `idpoly` object. This property is read only.

`Report`

Summary report that contains information about the estimation options and results when the model is estimated using the `nlhw` command. Use `Report` to query a model for how it was estimated, including:

- Estimation method
- Estimation options
- Search termination conditions
- Estimation data fit

The contents of `Report` are irrelevant if the model was created by construction.

```
m = idnlhw([2 2 1]);
m.Report.OptionsUsed
```

```
ans =

     []
```

If you use `nlhw` to estimate the model, the fields of `Report` contain information on the estimation data, options, and results.

```
load iddata1;
m = nlhw(z1,[2 2 1],[],'pwlinear');
m.Report.OptionsUsed
```

```
Option set for the nlhw command:

    InitialCondition: 'zero'
             Display: 'off'
      Regularization: [1x1 struct]
        SearchMethod: 'auto'
        SearchOption: [1x1 idoptions.search.identsolver]
        OutputWeight: 'noise'
            Advanced: [1x1 struct]
```

`Report` is a read-only property.

For more information on this property and how to use it, see “Output Arguments” on page 1-868 in the `nlhw` reference page and “Estimation Report”.

`TimeVariable`

Independent variable for the inputs, outputs, and—when available—internal states, specified as a character vector.

Default: `'t'`

NoiseVariance

Noise variance (covariance matrix) of the model innovations *e*.
Assignable value is an `ny`-by-`ny` matrix.
Typically set automatically by the estimation algorithm.

Ts

Sample time. `Ts` is a positive scalar representing the sampling period. This value is expressed in the unit specified by the `TimeUnit` property of the model.

Changing this property does not discretize or resample the model.

Default: `1`

TimeUnit

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as one of the following values:

- `'nanoseconds'`
- `'microseconds'`
- `'milliseconds'`
- `'seconds'`
- `'minutes'`
- `'hours'`
- `'days'`
- `'weeks'`
- `'months'`
- `'years'`

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

Default: `'seconds'`

InputName

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, `'controls'`.
- Cell array of character vectors — For multi-input models.

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 input names automatically expand to `{'controls(1)';'controls(2)'}`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `InputName` to `data.InputName`.

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: `''` for all input channels

InputUnit

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, `'seconds'`.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

Default: `''` 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. In this structure, field names are the group names, and field values are the input channels belonging 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, specified as one of the following:

- Character vector — For single-output models. For example, `'measurements'`.
- Cell array of character vectors — For multi-output models.

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 output names automatically expand to
`{'measurements(1)';'measurements(2)'}`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `OutputName` to `data.OutputName`.

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: `''` for all output channels

OutputUnit

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, `'seconds'`.
- Cell array of character vectors — For multi-output models.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

Default: `''` for all output 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. In this structure, field names are the group names, and field values are the output channels belonging 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, specified as a character vector. For example, `'system_1'`.

Default: `''`

`Notes`

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, `'System is MIMO'`.

Default: `{}`

`UserData`

Any type of data you want to associate with system, specified as any MATLAB data type.

Default: [ ]

Output Arguments

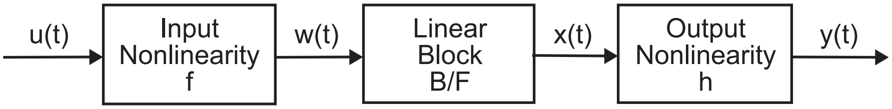
`sys` — Hammerstein-Wiener model
`idnlhw` object

Hammerstein-Wiener model, returned as an `idnlhw` object. This model is created using the specified model orders and delays, input and output nonlinearity estimators, and properties.

More About

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 ny outputs and nu inputs, the linear block is a transfer function matrix containing entries:

  $$\frac{B_{j,i}(q)}{F_{j,i}(q)}$$

  where j = `1,2,...,ny` and i = `1,2,...,nu`.

- $y(t) = h(x(t))$ is a nonlinear function that maps the output of the linear block to the system output.

w(t) and *x(t)* are internal variables that define the input and output of the linear block, respectively.

Because f acts on the input port of the linear block, this function is called the *input nonlinearity*. Similarly, because h acts on the output port of the linear block, this function is called the *output nonlinearity*. If system contains several inputs and outputs, you must define the functions f and h for each input and output signal.

You do not have to include both the input and the output nonlinearity in the model structure. When a model contains only the input nonlinearity f, it is called a *Hammerstein* model. Similarly, when the model contains only the output nonlinearity h), it is called a *Wiener* model.

The nonlinearities f and h are scalar functions, one nonlinear function for each input and output channel.

The Hammerstein-Wiener model computes the output y in three stages:

1. Computes $w(t) = f(u(t))$ from the input data.

   $w(t)$ is an input to the linear transfer function B/F.

   The input nonlinearity is a static (*memoryless*) function, where the value of the output a given time t depends only on the input value at time t.

   You can configure the input nonlinearity as a sigmoid network, wavelet network, saturation, dead zone, piecewise linear function, one-dimensional polynomial, or a custom network. You can also remove the input nonlinearity.
2. Computes the output of the linear block using $w(t)$ and initial conditions: $x(t) = (B/F)w(t)$.

   You can configure the linear block by specifying the numerator B and denominator F orders.
3. Compute the model output by transforming the output of the linear block $x(t)$ using the nonlinear function h: $y(t) = h(x(t))$.

   Similar to the input nonlinearity, the output nonlinearity is a static function. Configure the output nonlinearity in the same way as the input nonlinearity. You can also remove the output nonlinearity, such that $y(t) = x(t)$.

Resulting models are `idnlhw` objects that store all model data, including model parameters and nonlinearity estimator.

Definition of `idnlhw` States

The states of a Hammerstein-Wiener model correspond to the states of the linear block in the model structure. The linear block contains all the dynamic elements of the model. If the linear block 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.

States are required for simulation, prediction, and linearization of Hammerstein-Wiener models. To specify the initial states:

- Use `findstates` to search for state values for simulation and prediction with `sim`, `predict`, and `compare`.
- Use `findop` when linearizing the model with `linearize`.
- Alternatively, specify the states manually.

- “Identifying Hammerstein-Wiener Models”
- “Using Linear Model for Hammerstein-Wiener Estimation”

See Also

`customnet` | `findop` | `linear` | `linearize` | `nlhw` | `pem` | `poly1d` | `saturation` | `saturation` | `sigmoidnet` | `wavenet`

Introduced in R2007a

idpar

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 `paramname`.

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`

`Name` property of `p`, specified as a character vector. For example, you can assign `'x0'` as the name of a parameter created for initial state estimation.

The `Name` property is not used in state estimation or input level estimation. You can optionally assign a name for convenience.

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 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.

More About

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.

See Also

`findstates` | `findstatesOptions` | `forecast` | `pem` | `predict` | `ssest`

Introduced in R2012a

idpoly

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 sample 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 sample time. (Omit `NoiseVariance` and `Ts` to use their default values.)

If `D = []`, then `sys` represents the ARMA model given by:

$$A\left(q^{-1}\right)y(t)=C\left(q^{-1}\right)e(t).$$

`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 (nonidentified) `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 commands such as `polyest`, `arx`, `armax`, `oe`, `bj`, `iv4`, or `ivar`. These 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:

$$\begin{aligned}
&y_1(t) + 0.5y_1(t-1) + 0.9y_2(t-1) + 0.1y_2(t-2) = \\
&\qquad u(t) + 5u(t-1) + 2u(t-2) + e_1(t) + 0.01e_1(t-1) \\
&y_2(t) + 0.05y_2(t-1) + 0.3y_2(t-2) = \\
&\qquad 10u(t-2) + e_2(t) + 0.1e_2(t-1) + 0.02e_2(t-2).
\end{aligned}$$

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) + C(z)e_1(t)
    A(z) = 1 + 0.5 z^-1

    A_2(z) = 0.9 z^-1 + 0.1 z^-2

    B(z) = 1 + 5 z^-1 + 2 z^-2

    C(z) = 1 + 0.01 z^-1

  Model for output number 2: A(z)y_2(t) = B(z)u(t) + C(z)e_2(t)
    A(z) = 1 + 0.05 z^-1 + 0.3 z^-2

    B(z) = 10 z^-2

    C(z) = 1 + 0.1 z^-1 + 0.02 z^-2

Sample time: unspecified

Parameterization:
   Polynomial orders:   na=[1 2;0 2]   nb=[3;1]   nc=[1;2]
   nk=[0;2]
   Number of free coefficients: 12
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

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`.

- "Estimate Polynomial Models in the App"
- "Estimate Polynomial Models at the Command Line"
- "Polynomial Sizes and Orders of Multi-Output Polynomial Models"

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,j}` 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,j}` and `F{i,j}`contain coefficients of relations between output y_i and input u_j
`C,D`	N_y-by-1 array of row vectors	`C{i}` and `D{i}`contain coefficients of relations between output y_i and noise e_i

The leading coefficients of the diagonal entries of `A` (`A{i,i},i=1:Ny`) 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 wanted 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`

Sample time. For continuous-time models, `Ts = 0`. For discrete-time models, `Ts` is a positive scalar representing the sample time expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample time, set `Ts = -1`.

Default: –1 (discrete-time model with unspecified sample 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 to be converted into an `idpoly` object.

When `sys0` is an identified model, its estimated parameter covariance is lost during conversion. If you want to translate the estimated parameter covariance during the conversion, use `translatecov`.

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. `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,j}` 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,j}` and `F{i,j}`contain coefficients of relations between output y_i and input u_j
`C,D`	N_y-by-1 array of row vectors	`C{i}` and `D{i}`contain coefficients of relations between output y_i and noise e_i

The leading coefficients of the diagonal entries of `A` (`A{i,i}, i=1:Ny`) 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`

Polynomial model display variable, specified as one of 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.

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, transport delays are expressed as integers denoting delay of a multiple of the sample time `Ts`.

For a MIMO system with `Ny` outputs and `Nu` inputs, set `IODelay` is a `Ny`-by-`Nu` 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 ith 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.B.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.B.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.

  Specify parameter units and labels as character vectors. For example, `'Time'`.

For a MIMO model with `Ny` outputs and `Nu` inputs, the dimensions of the `Structure` elements are as follows:

- `sys.Structure.A` — `Ny`-by-`Ny`

- `sys.Structure.B` — `Ny`-by-`Nu`
- `sys.Structure.C` — `Ny`-by-1
- `sys.Structure.D` — `Ny`-by-1
- `sys.Structure.F` — `Ny`-by-`Nu`

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`

Summary report that contains information about the estimation options and results when the polynomial model is obtained using estimation commands, such as `polyest`, `armax`, `oe`, and `bj`. Use `Report` to query a model for how it was estimated, including its:

- Estimation method
- Estimation options
- Search termination conditions
- Estimation data fit and other quality metrics

The contents of `Report` are irrelevant if the model was created by construction.

```
m = idpoly({[1 0.5]},{[1 5]},{[1 0.01]});
m.Report.OptionsUsed
```

```
ans =

     []
```

If you obtain the polynomial model using estimation commands, the fields of `Report` contain information on the estimation data, options, and results.

```
load iddata2 z2;
m = polyest(z2,[2 2 3 3 2 1]);
m.Report.OptionsUsed
```

```
Option set for the polyest command:

    InitialCondition: 'auto'
               Focus: 'prediction'
            EstCovar: 1
             Display: 'off'
         InputOffset: []
        OutputOffset: []
      Regularization: [1x1 struct]
        SearchMethod: 'auto'
        SearchOption: [1x1 idoptions.search.identsolver]
            Advanced: [1x1 struct]
```

`Report` is a read-only property.

For more information on this property and how to use it, see the Output Arguments section of the corresponding estimation command reference page and "Estimation Report".

InputDelay

Input delay for each input channel, specified as a scalar value or numeric vector. 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 sample time `Ts`. For example, `InputDelay = 3` means a delay of three sample times.

For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector. Each entry of this vector is a numerical value that represents 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

OutputDelay

Output delays.

For identified systems, such as `idpoly`, `OutputDelay` is fixed to zero.

`Ts`

Sample time. For continuous-time models, `Ts = 0`. For discrete-time models, `Ts` is a positive scalar representing the sample time expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample 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 sample time of a discrete-time system.

Default: –1 (discrete-time model with unspecified sample time)

`TimeUnit`

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as one of the following values:

- `'nanoseconds'`
- `'microseconds'`
- `'milliseconds'`
- `'seconds'`
- `'minutes'`
- `'hours'`
- `'days'`
- `'weeks'`
- `'months'`
- `'years'`

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

Default: `'seconds'`

`InputName`

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, `'controls'`.
- Cell array of character vectors — For multi-input models.

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 input names automatically expand to `{'controls(1)';'controls(2)'}`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `InputName` to `data.InputName`.

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: `''` for all input channels

InputUnit

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, `'seconds'`.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

Default: `''` 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. In this structure, field names are the group names, and field values are the input channels belonging 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, specified as one of the following:

- Character vector — For single-output models. For example, `'measurements'`.
- Cell array of character vectors — For multi-output models.

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 output names automatically expand to `{'measurements(1)';'measurements(2)'}`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `OutputName` to `data.OutputName`.

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: `''` for all output channels

OutputUnit

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, `'seconds'`.
- Cell array of character vectors — For multi-output models.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

Default: `''` for all output 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. In this structure, field names are the group names, and field values are the output channels belonging 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, specified as a character vector. For example, `'system_1'`.

Default: `''`

`Notes`

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, `'System is MIMO'`.

Default: `{}`

`UserData`

Any type of data you want to associate with system, specified as any MATLAB data type.

Default: `[]`

`SamplingGrid`

Sampling grid for model arrays, specified as a data structure.

For arrays of identified linear (IDLTI) models that are derived by sampling one or more independent variables, this property tracks the variable values associated with each model. This information appears when you display or plot the model array. Use this information to trace results back to the independent variables.

Set the field names of the data structure to the names of the sampling variables. Set the field values to the sampled variable values associated with each model in the array. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, if you collect data at various operating points of a system, you can identify a model for each operating point separately and then stack the results together into a single system array. You can tag the individual models in the array with information regarding the operating point:

```
nominal_engine_rpm = [1000 5000 10000];
sys.SamplingGrid = struct('rpm', nominal_engine_rpm)
```

where `sys` is an array containing three identified models obtained at rpms 1000, 5000 and 10000, respectively.

For model arrays generated by linearizing a Simulink model at multiple parameter values or operating points, the software populates `SamplingGrid` automatically with the variable values that correspond to each entry in the array. For example, the Simulink Control Design commands `linearize` and `slLinearizer` populate `SamplingGrid` in this way.

Default: `[]`

More About

Tips

- Although `idpoly` supports continuous-time models, `idtf` and `idproc` enable more choices for estimation of continuous-time models. Therefore, for some continuous-time applications, these model types are preferable.

- "What Are Polynomial Models?"
- "Dynamic System Models"

See Also

`ar` | `armax` | `arx` | `bj` | `idproc` | `idss` | `idtf` | `iv4` | `ivar` | `oe` | `polydata` | `polyest` | `setPolyFormat` | `translatecov`

Introduced before R2006a

idproc

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` 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. K_{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:

$$sys = K_p \frac{1+T_z s}{\left(1+T_{p1}s\right)\left(1+T_{p2}s\right)\left(1+T_{p3}s\right)}e^{-T_d s}.$$

Two of the poles can be a complex conjugate (underdamped) pair. In that case, the general form of the process model is:

$$sys = K_p \frac{1 + T_z s}{\left(1 + 2\zeta T_\omega s + (T_\omega s)^2\right)\left(1 + T_{p3} s\right)} e^{-T_d s}.$$

T_ω is the time constant of the complex pair of poles, and ζ 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`:

$$sys = K_p \frac{1}{s\left(1 + 2\zeta T_\omega s + (T_\omega s)^2\right)} e^{-T_d s}.$$

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 1-599 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 1-599 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

Create 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:

$$sys = \frac{0.01}{1 + 2\left(0.1\right)\left(10\right)s + \left(10s\right)^2}e^{-5s}$$

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 `'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`.

Create a 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)
```

```
sys =
Process model with 3 outputs: y_k = Gk(s)u
  From input 1 to output 1:
                     1+Tz*s
  G1(s) = Kp * ------------------- * exp(-Td*s)
                s(1+Tp1*s)(1+Tp2*s)

         Kp = NaN
        Tp1 = NaN
        Tp2 = NaN
         Td = NaN
         Tz = NaN

  From input 1 to output 2:
                     1+Tz*s
  G1(s) = Kp * ------------------
```

```
                 (1+Tp1*s)(1+Tp2*s)

           Kp = NaN
          Tp1 = NaN
          Tp2 = NaN
           Tz = NaN

  From input 1 to output 3:
                      1+Tz*s
  G1(s) = Kp * -------------------
                 s(1+Tp1*s)(1+Tp2*s)

           Kp = NaN
          Tp1 = NaN
          Tp2 = NaN
           Tz = NaN

Parameterization:
    'P2DIZ'
    'P2Z'
    'P2IZ'
   Number of free coefficients: 13
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

`idproc` creates a MIMO model where each character vector in the `type` array defines the structure of the corresponding I/O pair. Since `type` is a column vector of character vectors, `sys` is a one-input, three-output model having the specified parametrization structure. `type{k,1}` specifies the structure of the subsystem `sys(k,1)`. All identifiable parameters are initialized to `NaN`.

Create Array of Process Models

Create a 3-by-1 array of process models, each containing one output and two input channels.

Specify the structure for each model in the array of process models.

```
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)
```

```
ans =
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:
                            Kp
  G12(s) = ---------------------------------
           s(1+2*Zeta*Tw*s+(Tw*s)^2)(1+Tp3*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`

Model structure, specified as a character vector or cell array of character vectors.

For SISO models, `type` is a character vector made up of one or more of the following 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 `type` with `P0` cannot include `Z` (a process model with no poles cannot include a zero).
`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` character vector must begin with one of `P0`, `P1`, `P2`, or `P3`. All other components are optional. For example:

- `'P1D'` specifies a process model with one pole and a time delay (deadtime) term:

  $$sys = \frac{K_p}{1+T_{p1}s} e^{-T_d s}.$$

  `Kp`, `Tp1`, and `Td` are the identifiable parameters of this model.
- `'P2U'` creates a process model with a pair of complex poles:

  $$sys = \frac{K_p}{\left(1+2\zeta T_\omega s+\left(T_\omega s\right)^2\right)}.$$

  `Kp`, `Tw`, and `Zeta` are the identifiable parameters of this model.

- `'P3ZDI'` creates a process model with three poles. All poles are real, because `U` is not included. The model also includes a zero, a time delay, and an integrator:

  $$sys = K_p \frac{1+T_z s}{s\left(1+T_{p1}s\right)\left(1+T_{p2}s\right)\left(1+T_{p3}s\right)} e^{-T_d s}.$$

  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 character vectors 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. `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`

Model structure, specified as a character vector or cell array of character vectors.

For a SISO model `sys`, the property `sys.Type` contains a character vector specifying the structure of the system. For example, `'P1D'`.

For a MIMO model with `Ny` outputs and `Nu` inputs, `sys.Type` is an `Ny`-by-`Nu` cell array of character vectors specifying the structure of each input/output pair in the model. For example, `type{i,j}` specifies the structure of the subsystem `sys(i,j)` from the *j*th input to the *i*th output.

The character vectors are made up of one or more of the following characters that specify aspects of the model structure:

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 model structure that you specify with the `type` input argument.

If you obtain an `idproc` model by identification using `procest`, then `sys.Type` contains the model structures that you specified for that identification.

In general, you cannot change the type 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 `Ny` outputs and `Nu` inputs, each parameter value is an `Ny`-by-`Nu` cell array of character vectors specifying the corresponding parameter value for each input/output pair in the model. For example, `sys.Kp(i,j)` specifies the `Kp` 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` 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.

  Specify parameter units and labels as character vectors. For example, `'Time'`.

`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

Summary report that contains information about the estimation options and results when the process model is obtained using the `procest` estimation command. Use `Report` to query a model for how it was estimated, including its:

- Estimation method
- Estimation options
- Search termination conditions
- Estimation data fit and other quality metrics

The contents of `Report` are irrelevant if the model was created by construction.

```
m = idproc('P2DU');
m.Report.OptionsUsed
```

```
ans =

     []
```

If you obtain the process model using estimation commands, the fields of `Report` contain information on the estimation data, options, and results.

```
load iddata2 z2;
m = procest(z2,'P2DU');
m.Report.OptionsUsed
```

```
DisturbanceModel: 'estimate'
    InitialCondition: 'auto'
               Focus: 'prediction'
            EstCovar: 1
             Display: 'off'
         InputOffset: [1x1 param.Continuous]
        OutputOffset: []
      Regularization: [1x1 struct]
        SearchMethod: 'auto'
        SearchOption: [1x1 idoptions.search.identsolver]
        OutputWeight: []
            Advanced: [1x1 struct]
```

`Report` is a read-only property.

For more information on this property and how to use it, see the Output Arguments section of the corresponding estimation command reference page and “Estimation Report”.

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`

Sample time. For `idproc`, `Ts` is fixed to zero because all `idproc` models are continuous time.

`TimeUnit`

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as one of the following values:

- `'nanoseconds'`
- `'microseconds'`
- `'milliseconds'`
- `'seconds'`
- `'minutes'`
- `'hours'`
- `'days'`
- `'weeks'`
- `'months'`
- `'years'`

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

Default: `'seconds'`

`InputName`

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, `'controls'`.
- Cell array of character vectors — For multi-input models.

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 input names automatically expand to `{'controls(1)';'controls(2)'}`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `InputName` to `data.InputName`.

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: `''` for all input channels

InputUnit

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, `'seconds'`.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

Default: `''` 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. In this structure, field names are the group names, and field values are the input channels belonging 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, specified as one of the following:

- Character vector — For single-output models. For example, `'measurements'`.
- Cell array of character vectors — For multi-output models.

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 output names automatically expand to `{'measurements(1)';'measurements(2)'}`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `OutputName` to `data.OutputName`.

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: `''` for all output channels

`OutputUnit`

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, `'seconds'`.

- Cell array of character vectors — For multi-output models.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

Default: `''` for all output 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. In this structure, field names are the group names, and field values are the output channels belonging 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, specified as a character vector. For example, `'system_1'`.

Default: `''`

`Notes`

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, `'System is MIMO'`.

Default: `{}`

`UserData`

Any type of data you want to associate with system, specified as any MATLAB data type.

Default: `[]`

`SamplingGrid`

Sampling grid for model arrays, specified as a data structure.

For arrays of identified linear (IDLTI) models that are derived by sampling one or more independent variables, this property tracks the variable values associated with each model. This information appears when you display or plot the model array. Use this information to trace results back to the independent variables.

Set the field names of the data structure to the names of the sampling variables. Set the field values to the sampled variable values associated with each model in the array. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, if you collect data at various operating points of a system, you can identify a model for each operating point separately and then stack the results together into a single system array. You can tag the individual models in the array with information regarding the operating point:

```
nominal_engine_rpm = [1000 5000 10000];
sys.SamplingGrid = struct('rpm', nominal_engine_rpm)
```

where `sys` is an array containing three identified models obtained at rpms 1000, 5000 and 10000, respectively.

For model arrays generated by linearizing a Simulink model at multiple parameter values or operating points, the software populates `SamplingGrid` automatically with the variable values that correspond to each entry in the array. For example, the Simulink Control Design commands `linearize` and `slLinearizer` populate `SamplingGrid` in this way.

Default: `[]`

See Also

`idss` | `idtf` | `pem` | `procest` | `ssest` | `tfest`

Introduced before R2006a

idresamp

Resample time-domain data by decimation or interpolation

Syntax

```
datar = idresamp(data,R)
datar = idresamp(data,R,order,tol)
[datar,res_fact] = idresamp(data,R,order,tol)
```

Description

`datar = idresamp(data,R)` resamples data on a new sample interval `R` and stores the resampled data as `datar`.

`datar = idresamp(data,R,order,tol)` filters the data by applying a filter of specified `order` before interpolation and decimation. Replaces `R` by a rational approximation that is accurate to a tolerance `tol`.

`[datar,res_fact] = idresamp(data,R,order,tol)` returns `res_fact`, which corresponds to the value of `R` approximated by a rational expression.

Input Arguments

`data`

Name of time-domain `iddata` object or a matrix of data. Can be input-output or time-series data.

Data must be sampled at equal time intervals.

`R`

Resampling factor, such that `R>1` results in decimation and `R<1` results in interpolation.

Any positive number you specify is replaced by the rational approximation, `Q/P`.

`order`

Order of the filters applied before interpolation and decimation.

Default: `8`

`tol`

Tolerance of the rational approximation for the resampling factor `R`.

Smaller tolerance might result in larger `P` and `Q` values, which produces more accurate answers at the expense of slower computation.

Default: `0.1`

Output 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`

Introduced in R2007a

idss

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 sample 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 sample 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.

`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 sample 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`

Sample 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 sample time, set `Ts = -1`.

Default: –1 (discrete-time model with unspecified sample time)

`sys0`

Dynamic system.

Any dynamic system to convert to an `idss` model:

- When `sys0` is an identified model, its estimated parameter covariance is lost during conversion. If you want to translate the estimated parameter covariance during the conversion, use `translatecov`.
- When `sys0` is a numeric (non-identified) model, 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`.

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. `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. `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` object 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, specified as one of the following:

- Character vector — For first-order models, for example, `'velocity'`.
- Cell array of character vectors — For models with two or more states
- `''` — For unnamed states.

Default: `''` for all states

`StateUnit`

State units, specified as one of the following:

- Character vector — For first-order models, for example, `'velocity'`.
- Cell array of character vectors — For models with two or more states
- `''` — For unnamed states.

Use `StateUnit` to keep track of the units each state is expressed in. `StateUnit` has no effect on system behavior.

Default: `''` 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.

  Specify parameter units and labels as character vectors. For example, `'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, 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

Summary report that contains information about the estimation options and results when the state-space model is obtained using estimation commands, such as `ssest`, `ssregest`, and `n4sid`. Use `Report` to query a model for how it was estimated, including its:

- Estimation method
- Estimation options
- Search termination conditions
- Estimation data fit and other quality metrics

The contents of `Report` are irrelevant if the model was created by construction.

```
A = [-0.1 0.4; -0.4 -0.1];
B = [1; 0];
C = [1 0];
D = 0;
```

```
m = idss(A,B,C,D);
m.Report.OptionsUsed

ans =

     []
```

If you obtain the state-space model using estimation commands, the fields of `Report` contain information on the estimation data, options, and results.

```
load iddata2 z2;
m = ssest(z2,3);
m.Report.OptionsUsed

InitialState: 'auto'
          N4Weight: 'auto'
         N4Horizon: 'auto'
             Focus: 'prediction'
          EstCovar: 1
           Display: 'off'
       InputOffset: []
      OutputOffset: []
      OutputWeight: []
      SearchMethod: 'auto'
      SearchOption: [1x1 idoptions.search.identsolver]
    Regularization: [1x1 struct]
          Advanced: [1x1 struct]
```

`Report` is a read-only property.

For more information on this property and how to use it, see the Output Arguments section of the corresponding estimation command reference page and “Estimation Report”.

InputDelay

Input delay for each input channel, specified as a scalar value or numeric vector. 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 sample time `Ts`. For example, `InputDelay = 3` means a delay of three sample times.

For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector. Each entry of this vector is a numerical value that represents 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

`OutputDelay`

Output delays.

For identified systems, like `idss`, `OutputDelay` is fixed to zero.

`Ts`

Sample 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 sample 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 sample time of a discrete-time system.

Default: –1 (discrete-time model with unspecified sample time)

`TimeUnit`

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as one of the following values:

- `'nanoseconds'`
- `'microseconds'`
- `'milliseconds'`
- `'seconds'`
- `'minutes'`
- `'hours'`
- `'days'`
- `'weeks'`
- `'months'`
- `'years'`

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

Default: `'seconds'`

`InputName`

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, `'controls'`.
- Cell array of character vectors — For multi-input models.

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 input names automatically expand to `{'controls(1)';'controls(2)'}`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `InputName` to `data.InputName`.

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: `''` for all input channels

`InputUnit`

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, `'seconds'`.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

Default: `''` 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. In this structure, field names are the group names, and field values are the input channels belonging 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, specified as one of the following:

- Character vector — For single-output models. For example, `'measurements'`.
- Cell array of character vectors — For multi-output models.

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 output names automatically expand to `{'measurements(1)';'measurements(2)'}`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `OutputName` to `data.OutputName`.

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: `''` for all output channels

`OutputUnit`

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, `'seconds'`.
- Cell array of character vectors — For multi-output models.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

Default: `''` for all output 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. In this structure, field names are the group names, and field values are the output channels belonging 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, specified as a character vector. For example, `'system_1'`.

Default: `''`

`Notes`

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, `'System is MIMO'`.

Default: `{}`

`UserData`

Any type of data you want to associate with system, specified as any MATLAB data type.

Default: `[]`

`SamplingGrid`

Sampling grid for model arrays, specified as a data structure.

For arrays of identified linear (IDLTI) models that are derived by sampling one or more independent variables, this property tracks the variable values associated with each model. This information appears when you display or plot the model array. Use this information to trace results back to the independent variables.

Set the field names of the data structure to the names of the sampling variables. Set the field values to the sampled variable values associated with each model in the array. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, if you collect data at various operating points of a system, you can identify a model for each operating point separately and then stack the results together into a single system array. You can tag the individual models in the array with information regarding the operating point:

```
nominal_engine_rpm = [1000 5000 10000];
sys.SamplingGrid = struct('rpm', nominal_engine_rpm)
```

where `sys` is an array containing three identified models obtained at rpms 1000, 5000 and 10000, respectively.

For model arrays generated by linearizing a Simulink model at multiple parameter values or operating points, the software populates `SamplingGrid` automatically with the variable values that correspond to each entry in the array. For example, the Simulink Control Design commands `linearize` and `slLinearizer` populate `SamplingGrid` in this way.

Default: [ ]

More About

- "Dynamic System Models"

See Also

`idgrey` | `idpoly` | `idproc` | `idssdata` | `idtf` | `n4sid` | `pem` | `ssest` | `ssestOptions` | `translatecov`

Introduced before R2006a

idssdata

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 (`idss` or `idgrey`), then it is first converted to an `idss` model. This conversion results in a loss of the model uncertainty information.

`sys` can 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);
```

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);
[A,B,C,D,K,x0] = idssdata(sys,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 array `sys`. `x0` is a cell array containing the estimated initial state of the individual entries of the identified model array `sys`.

See Also

`idss` | `polydata` | `ssdata` | `tfdata` | `zpkdata`

Introduced in R2012a

idtf

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 sample 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,

$$G(s) = e^{-\tau s} \frac{b_n s^n + b_{n-1} s^{n-1} + \ldots + b_0}{s^m + a_{m-1} s^{m-1} + \ldots + a_0}.$$

In discrete time,

$$G\left(z^{-1}\right)=z^{-k}\frac{b_n z^{-n}+b_{n-1}z^{-n+1}+\ldots+b_0}{z^{-m}+a_{m-1}z^{-m+1}+\ldots+a_0}.$$

In discrete time, z^{-k} represents a time delay of kT_s, where T_s is the sample time.

For `idtf` models, the denominator coefficients $a_0,\ldots,a_{m-1}$ and the numerator coefficients $b_0,\ldots,b_n$ can be estimable parameters. (The leading denominator coefficient is always fixed to 1.) The time delay τ (or $k$in discrete time) can also be an estimable parameter. The `idtf` model stores the polynomial coefficients $a_0,\ldots,a_{m-1}$ and $b_0,\ldots,b_n$ in the `Denominator` and `Numerator` properties of the model, respectively. The time delay τ 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 `Numerator`, `Denominator`, 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. To do so, use 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.

Note: Unlike `idss` and `idpoly`, `idtf` uses a trivial noise model and does not parameterize the noise.

So, $H = 1$ in $y = Gu + He$.

Examples

Create a Continuous-Time Transfer Function Model

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`.

Create 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. The estimation process 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.

Create 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 sample 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 sample 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).

Create 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 sample 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 cell array 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 sample 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 Transfer Function Display Variable

Specify the following discrete-time transfer function in terms of `q^-1`:

$$H(q^{-1}) = \frac{1 + 0.4q^{-1}}{1 + 0.1q^{-1} - 0.3q^{-2}}$$

Specify the sample time as 0.1 seconds.

```
num = [1 0.4];
den = [1 0.1 -0.3];
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 sample 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.

$$\begin{aligned} \tilde{x}(t) &= \begin{bmatrix} -0.2 & 0 \\ 0 & -0.3 \end{bmatrix} x(t) + \begin{bmatrix} -2 \\ 4 \end{bmatrix} u(t) + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} e(t) \\ y(t) &= \begin{bmatrix} 1 & 1 \end{bmatrix} x(t) \end{aligned}$$

```
A = [-0.2, 0; 0, -0.3];
B = [2;4];
C = [1, 1];
D = 0;
K = [0.1; 0.2];
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`.

Estimate Transfer Function Model By Specifying Number of Poles

Load time-domain system response data and use it to estimate a transfer function for the system.

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

`z1` is an `iddata` object that contains time-domain, input-output data.

`np` specifies the number of poles in the estimated transfer function.

`sys` is an `idtf` model containing the estimated transfer function.

To see the numerator and denominator coefficients of the resulting estimated model `sys`, enter:

```
sys.Numerator
sys.Denominator
```

```
ans =

    2.4554  176.9856


ans =

    1.0000    3.1625   23.1631
```

To view the uncertainty in the estimates of the numerator and denominator and other information, use `tfdata`.

Create 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 preallocates 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 $a = k/10$.

- "Estimate Transfer Function Models With Prior Knowledge of Model Structure and Constraints"

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 `Ny` outputs and `Nu` inputs, `num` is a `Ny`-by-`Nu` cell array of 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 `Ny` outputs and `Nu` inputs, `den` is a `Ny`-by-`Nu` cell array of denominator coefficients for each input/output pair.

`Ts`

Sample time. For continuous-time models, `Ts = 0`. For discrete-time models, `Ts` is a positive scalar representing the sampling period. This value is expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample 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 sample time of a discrete-time system.

Default: `0` (continuous time)

`sys0`

Dynamic system.

Any dynamic system to convert to an `idtf` model.

When `sys0` is an identified model, its estimated parameter covariance is lost during conversion. If you want to translate the estimated parameter covariance during the conversion, use `translatecov`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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:

`Numerator`

Values of transfer function numerator coefficients.

If you create an `idtf` model `sys` using the `idtf` command, `sys.Numerator` 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.Numerator` contains the estimated values of the numerator coefficients.

For an `idtf` model `sys`, the property `sys.Numerator` is an alias for the value of the property `sys.Structure.Numerator.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 `Ny` outputs and `Nu` inputs, `Numerator` is a `Ny`-by-`Nu` cell array of numerator coefficients for each input/output pair.

`Denominator`

Values of transfer function denominator coefficients.

If you create an `idtf` model `sys` using the `idtf` command, `sys.Denominator` 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.Denominator` contains the estimated values of the denominator coefficients.

For an `idtf` model `sys`, the property `sys.Denominator` is an alias for the value of the property `sys.Structure.Denominator.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 `Denominator` is fixed to 1. Any coefficient whose initial value is not known is stored as `NaN`.

For MIMO transfer functions with `Ny` outputs and `Nu` inputs, `Denominator` is a `Ny`-by-`Nu` cell array of denominator coefficients for each input/output pair.

Variable

Transfer function display variable, specified as one of the following values:

- `'s'` — Default for continuous-time models
- `'p'` — Equivalent to `'s'`
- `'z^-1'` — Default for discrete-time models
- `'q^-1'` — Equivalent to `'z^-1'`

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^-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 `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 for 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 sample time `Ts`.

For a MIMO system with `Ny` outputs and `Nu` inputs, set `IODelay` as a `Ny`-by-`Nu` array. Each entry of this array 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.Numerator`, `Structure.Denominator`, 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.Numerator.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.Denominator.Value(1)` is fixed to 1.

  For SISO models, `sys.Numerator`, `sys.Denominator`, and `sys.IODelay` are aliases for `sys.Structure.Numerator.Value`, `sys.Structure.Denominator.Value`, and `sys.Structure.IODelay.Value`, respectively.

  For MIMO models, `sys.Numerator{i,j}` is an alias for `sys.Structure(i,j).Numerator.Value`, and `sys.Denominator{i,j}` is an alias for `sys.Structure(i,j).Denominator.Value`. Additionally, `sys.IODelay(i,j)` is an alias for `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.Denominator.Free = false` fixes all of the denominator coefficients in `sys` to the values specified in `sys.Structure.Denominator.Value`.

  For denominators, the value of `Free` for the leading coefficient, specified by `sys.Structure.Denominator.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.

  Specify parameter units and labels as character vectors. For example, `'Time'`.

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 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

Summary report that contains information about the estimation options and results when the transfer function model is obtained using estimation commands, such as `tfest` and `impulseest`. Use `Report` to query a model for how it was estimated, including its:

- Estimation method
- Estimation options
- Search termination conditions

- Estimation data fit and other quality metrics

The contents of `Report` are irrelevant if the model was created by construction.

```
m = idtf([1 4],[1 20 5]);
m.Report.OptionsUsed
```

```
ans =

     []
```

If you obtain the transfer function model using estimation commands, the fields of `Report` contain information on the estimation data, options, and results.

```
load iddata2 z2;
m = tfest(z2,3);
m.Report.OptionsUsed
```

```
InitMethod: 'iv'
          InitOption: [1x1 struct]
    InitialCondition: 'auto'
               Focus: 'simulation'
            EstCovar: 1
             Display: 'off'
         InputOffset: []
        OutputOffset: []
      Regularization: [1x1 struct]
        SearchMethod: 'auto'
        SearchOption: [1x1 idoptions.search.identsolver]
        OutputWeight: []
            Advanced: [1x1 struct]
```

`Report` is a read-only property.

For more information on this property and how to use it, see the Output Arguments section of the corresponding estimation command reference page and “Estimation Report”.

`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 sample time `Ts`. For example, `InputDelay = 3` means a delay of three sample times.

For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector. Each entry of this vector 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`

Sample time. For continuous-time models, `Ts = 0`. For discrete-time models, `Ts` is a positive scalar representing the sampling period. This value is expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample 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 sample time of a discrete-time system.

Default: `0` (continuous time)

`TimeUnit`

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as one of the following values:

- `'nanoseconds'`
- `'microseconds'`
- `'milliseconds'`
- `'seconds'`
- `'minutes'`

- `'hours'`
- `'days'`
- `'weeks'`
- `'months'`
- `'years'`

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

Default: `'seconds'`

InputName

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, `'controls'`.
- Cell array of character vectors — For multi-input models.

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 input names automatically expand to `{'controls(1)';'controls(2)'}`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `InputName` to `data.InputName`.

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: `''` for all input channels

InputUnit

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, `'seconds'`.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

Default: `''` 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. In this structure, field names are the group names, and field values are the input channels belonging 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, specified as one of the following:

- Character vector — For single-output models. For example, `'measurements'`.
- Cell array of character vectors — For multi-output models.

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 output names automatically expand to `{'measurements(1)';'measurements(2)'}`.

When you estimate a model using an `iddata` object, `data`, the software automatically sets `OutputName` to `data.OutputName`.

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: `''` for all output channels

`OutputUnit`

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, `'seconds'`.
- Cell array of character vectors — For multi-output models.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

Default: `''` for all output 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. In this structure, field names are the group names, and field values are the output channels belonging 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, specified as a character vector. For example, `'system_1'`.

Default: `''`

`Notes`

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, `'System is MIMO'`.

Default: `{}`

`UserData`

Any type of data you want to associate with system, specified as any MATLAB data type.

Default: `[]`

`SamplingGrid`

Sampling grid for model arrays, specified as a data structure.

For arrays of identified linear (IDLTI) models that are derived by sampling one or more independent variables, this property tracks the variable values associated with each model. This information appears when you display or plot the model array. Use this information to trace results back to the independent variables.

Set the field names of the data structure to the names of the sampling variables. Set the field values to the sampled variable values associated with each model in the array. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, if you collect data at various operating points of a system, you can identify a model for each operating point separately and then stack the results together into a single system array. You can tag the individual models in the array with information regarding the operating point:

```
nominal_engine_rpm = [1000 5000 10000];
sys.SamplingGrid = struct('rpm', nominal_engine_rpm)
```

where `sys` is an array containing three identified models obtained at rpms 1000, 5000 and 10000, respectively.

For model arrays generated by linearizing a Simulink model at multiple parameter values or operating points, the software populates `SamplingGrid` automatically with the variable values that correspond to each entry in the array. For example, the Simulink Control Design commands `linearize` and `slLinearizer` populate `SamplingGrid` in this way.

Default: [ ]

More About

- “Dynamic System Models”

See Also

`getcov` | `getpar` | `idfrd` | `idpoly` | `idproc` | `idss` | `oe` | `tfdata` | `tfest` | `translatecov`

Introduced in R2012a

ifft

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 sample time is `Ts`, then

`Datf.Frequency = [0:df:F]`, where `F` is `pi/Ts` if `N` is odd and `F = pi/Ts*(1-1/N)` if `N` is even.

See Also

`iddata` | `fft`

Introduced in R2007a

impulse

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,ysd] = 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 `Ts` and height `1/Ts`, where `Ts` is the sample time of the system. (This pulse approaches $\delta(t)$ as `Ts` 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 = Tfinal`. Express `Tfinal` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time systems with unspecified sample time (`Ts = -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 `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 “Algorithms” on page 1-660). The `impulse` command always applies the impulse at `t=0`, regardless of `Ti`.

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) × (number of outputs) × (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*) × (number of states) × (number of inputs)

`[y,t,x,ysd] = 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.

Examples

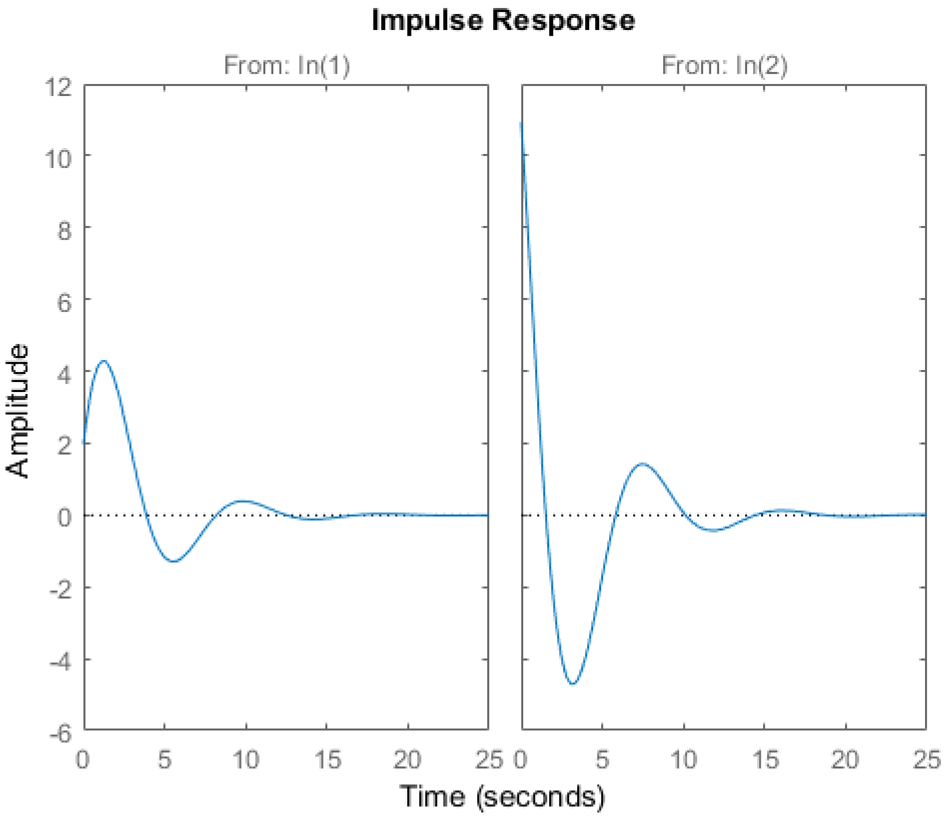
Impulse Response Plot of Second-Order State-Space Model

Plot the impulse response of the second-order state-space model

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -0.5572 & -0.7814 \\ 0.7814 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

$$y = \begin{bmatrix} 1.9691 & 6.4493 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

```
a = [-0.5572 -0.7814;0.7814  0];
b = [1 -1;0 2];
c = [1.9691  6.4493];
sys = ss(a,b,c,0);
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 =

   139     1     2
```

(the first dimension is the length of `t`). The impulse response of the first input channel is then accessed by

```
ch1 = y(:,:,1);
size(ch1)


ans =

   139     1
```

Impulse Data from Identified System

Fetch the impulse response and the corresponding 1 std uncertainty of an identified linear system .

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'dcmotordata'));
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:')
```

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 $t = 0$.

More About

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”.

Algorithms

Continuous-time models are first converted to state space. The impulse response of a single-input state-space model

$$\begin{aligned}\dot{x} &= Ax + bu\\ y &= Cx\end{aligned}$$

is equivalent to the following unforced response with initial state b.

$$\begin{aligned}\dot{x} &= Ax, \quad x(0) = b\\ y &= Cx\end{aligned}$$

To simulate this response, the system is discretized using zero-order hold on the inputs. The sample time is chosen automatically based on the system dynamics, except when a time vector `t = 0:dt:Tf` is supplied (`dt` is then used as sample time).

See Also

`impulseest` | Linear System Analyzer | `lsim` | `step`

Introduced before R2006a

impulseest

Nonparameteric 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 `N`th order impulse response model, corresponding to the time range $0 : Ts : (N-1)*Ts$, where `Ts` is the data sample 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.

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 *Nu* inputs and *Ny* outputs, you can also specify N as an *Ny*-by-*Nu* 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 *Nu* inputs and *Ny* outputs, you can also specify a *Ny*-by-*Nu* 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)*Ts : Ts : (N(ij)+NK(i,j)-1)*Ts`.

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, returned as an `idtf` model, which encapsulates an FIR model.

Information about the estimation results and options used is stored in the `Report` property of the model. `Report` has the following fields:

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Status</code></td><td>Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.</td></tr>
<tr><td><code>Method</code></td><td>Estimation command used.</td></tr>
<tr><td><code>Fit</code></td><td>Quantitative assessment of the estimation, returned as a structure. See “Loss Function and Model Quality Metrics” for more information on these quality metrics. The structure has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>FitP</code></td><td>Normalized root mean squared error (NRMSE) measure of how</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td></td><td>well the response of the model fits the estimation data, expressed as a percentage.</td></tr>
<tr><td><code>Loss</code></td><td>Value of the loss function when the estimation completes.</td></tr>
<tr><td><code>MSE</code></td><td>Mean squared error (MSE) measure of how well the response of the model fits the estimation data.</td></tr>
<tr><td><code>FPE</code></td><td>Final prediction error for the model.</td></tr>
<tr><td><code>AIC</code></td><td>Raw Akaike Information Criteria (AIC) measure of model quality.</td></tr>
<tr><td><code>AICc</code></td><td>Small sample-size corrected AIC.</td></tr>
<tr><td><code>nAIC</code></td><td>Normalized AIC.</td></tr>
<tr><td><code>BIC</code></td><td>Bayesian Information Criteria (BIC).</td></tr>
</table>
</td></tr>
<tr><td><code>Parameters</code></td><td>Estimated values of model parameters.</td></tr>
<tr><td><code>OptionsUsed</code></td><td>Option set used for estimation. If no custom options were configured, this is a set of default options. See <code>impulseestOptions</code> for more information.</td></tr>
<tr><td><code>RandState</code></td><td>State of the random number stream at the start of estimation. Empty, <code>[]</code>, if randomization was not used during estimation. For more information, see <code>rng</code> in the MATLAB documentation.</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation, returned as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
<tr><td><code>Type</code></td><td>Data type.</td></tr>
<tr><td><code>Leng</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time.</td></tr>
<tr><td><code>Inte</code></td><td>Input intersample behavior, returned as one of the following values:
• <code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
• <code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
• <code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.</td></tr>
<tr><td><code>Inpu</code></td><td>Offset removed from time-domain input data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
<tr><td><code>Outp</code></td><td>Offset removed from time-domain output data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
</table></td></tr>
</table>

For more information on using `Report`, see “Estimation Report”.

Examples

Identify Nonparametric Impulse Response Model from Data

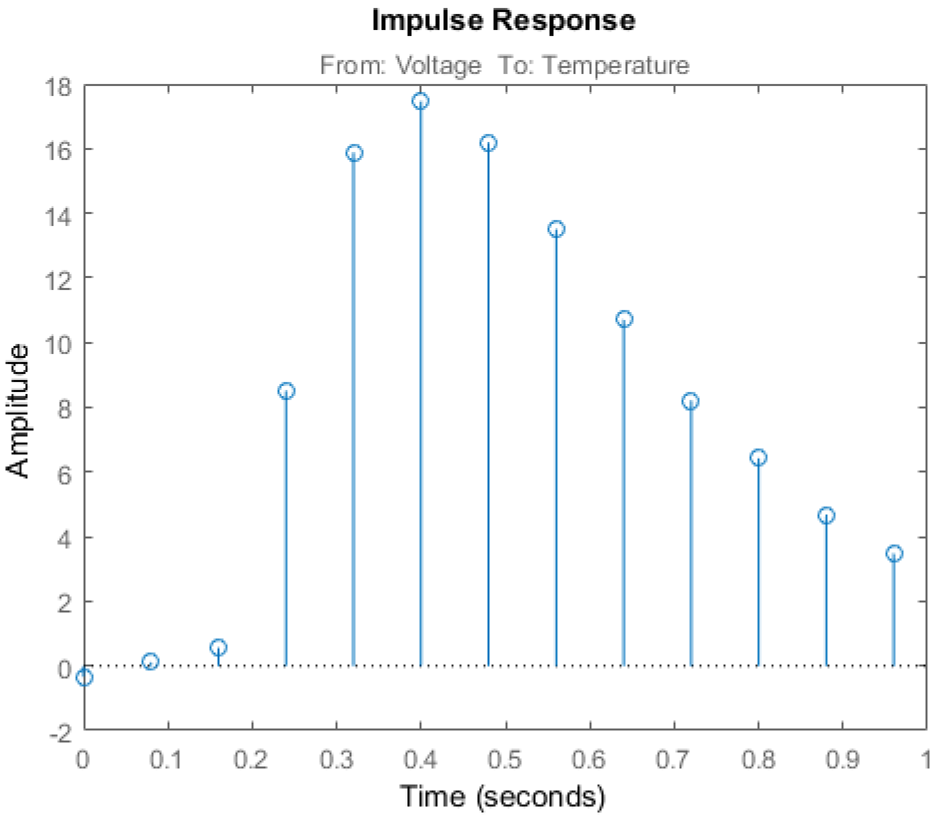
Compute a nonparametric impulse response model using data from a hair dryer. The input is the voltage applied to the heater and the output is the heater temperature. Use the first 500 samples for estimation.

```
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.

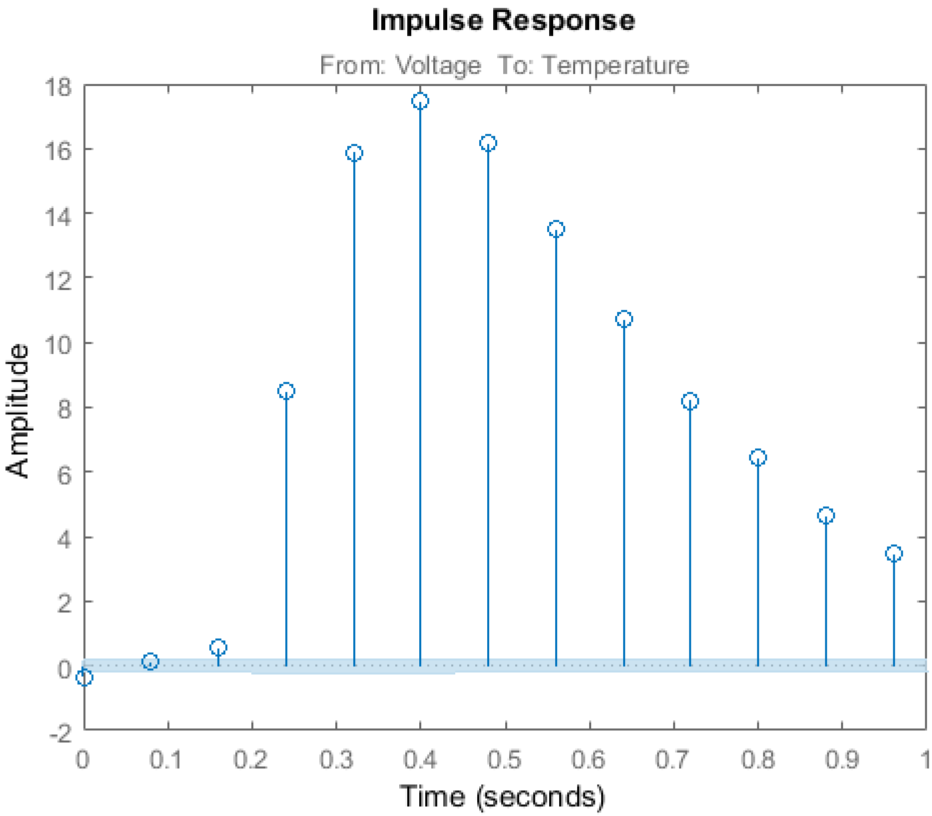
Analyze the impulse response of the identified model from time 0 to 1.

```
h = impulseplot(sys,1);
```



Right-click the plot and select **Characteristics > Confidence Region** to view the statistically zero-response region. Alternatively, you can use the `showConfidence` command.

```
showConfidence(h);
```



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

Load estimation data

```
load iddata3 z3;
```

Estimate a 35th order FIR model.

```
sys = impulseest(z3,35);
```

Specify Transport Delay in FIR Model

Estimate an impulse response model with transport delay of 3 samples.

If you know about the presence of delay in the input/output data in advance, use the value as a transport delay for impulse response estimation.

Generate data with 3-sample input to output lag. Create a random input signal and use an `idpoly` model to simulate the output data.

```
u = rand(100,1);
sys = idpoly([1 .1 .4],[0 0 0 4 -2],[1 1 .1]);
opt = simOptions('AddNoise',true);
y = sim(sys,u,opt);
data = iddata(y,u,1);
```

Estimate a 20th order model with a 3-sample transport delay.

```
model = impulseest(data,20,3);
```

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);
```

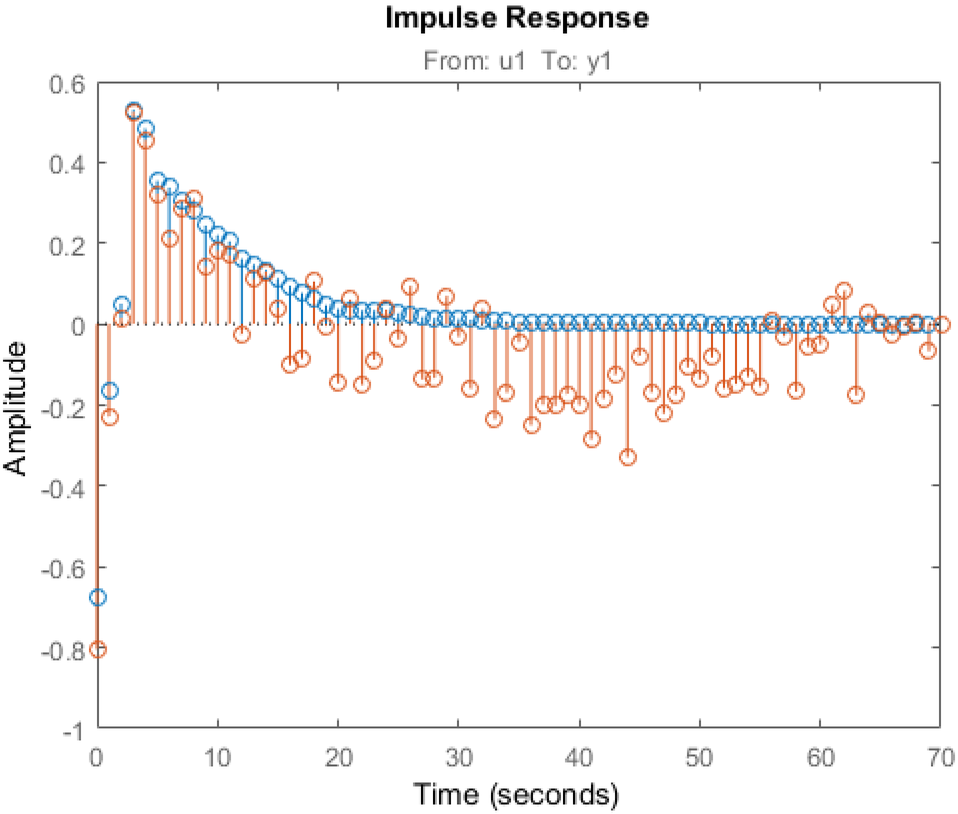
By default, tuned and correlated kernel (`'TC'`) is used for regularization.

Estimate a model with no regularization.

```
opt = impulseestOptions('RegulKernel','none');
sys2 = impulseest(z3,opt);
```

Compare the impulse response of both models.

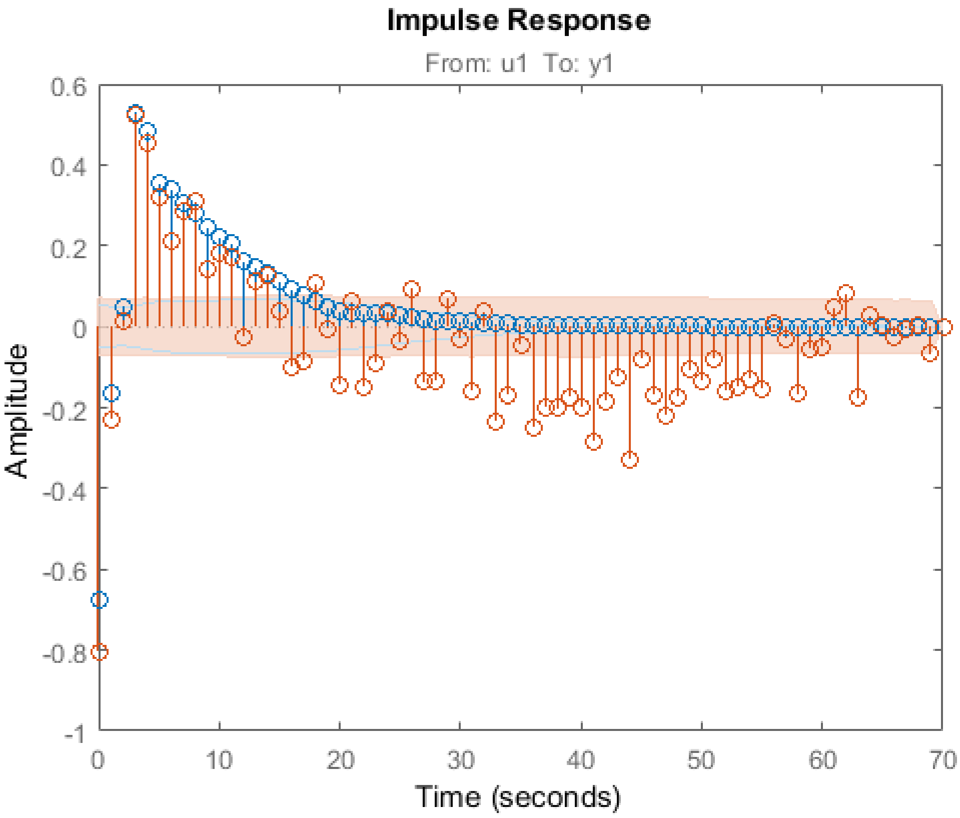
```
h = impulseplot(sys1,sys2,70);
```



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.

Use Regularized Impulse Response Model to Estimate State-Space Model

Load data.

```
load regularizationExampleData eData;
```

Create a transfer function model used for generating the estimation data (true system).

```
trueSys = idtf([0.02008 0.04017 0.02008],[1 -1.561 0.6414],1);
```

Obtain regularized impulse response (FIR) model.

```
opt = impulseestOptions('RegulKernel','DC');
m0 = impulseest(eData,70,opt);
```

Convert the model into a state-space model and reduce the model order.

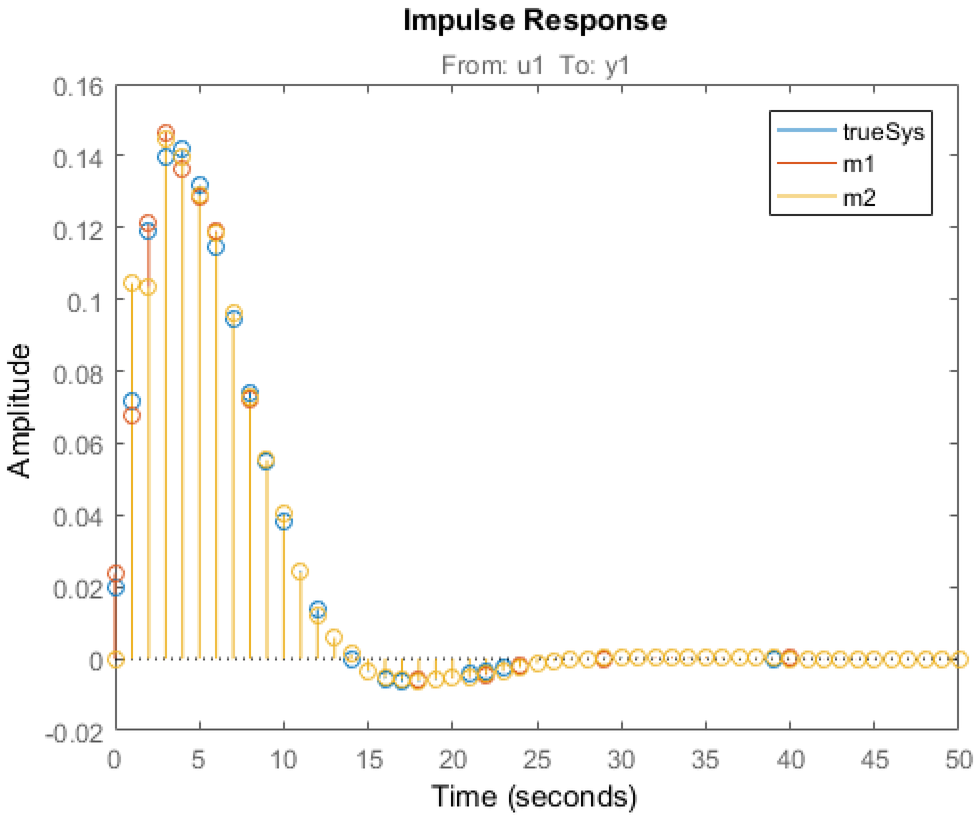
```
m1 = balred(idss(m0),15);
```

Obtain a second state-space model using regularized reduction of an ARX model.

```
m2 = ssregest(eData,15);
```

Compare the impulse responses of the true system and the estimated models.

```
impulse(trueSys,m1,m2,50);
legend('trueSys','m1','m2');
```



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.

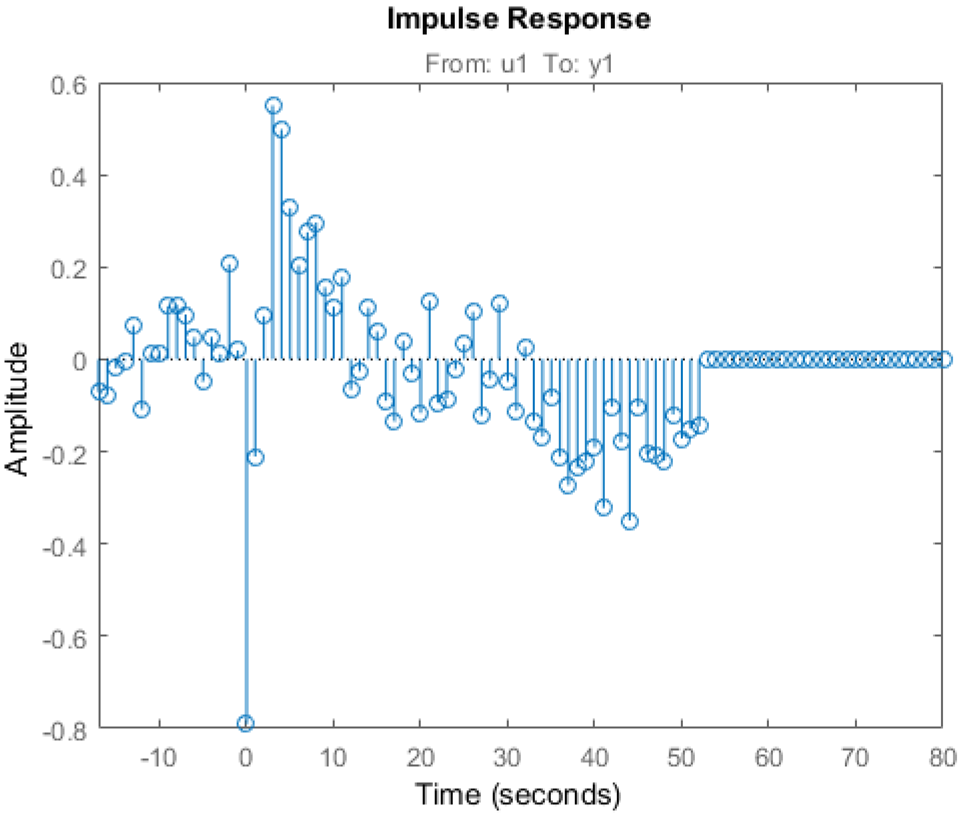
Compute the noncausal impulse response using a fourth-order prewhitening filter, automatically chosen order and negative lag using nonregularized estimation.

```
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.

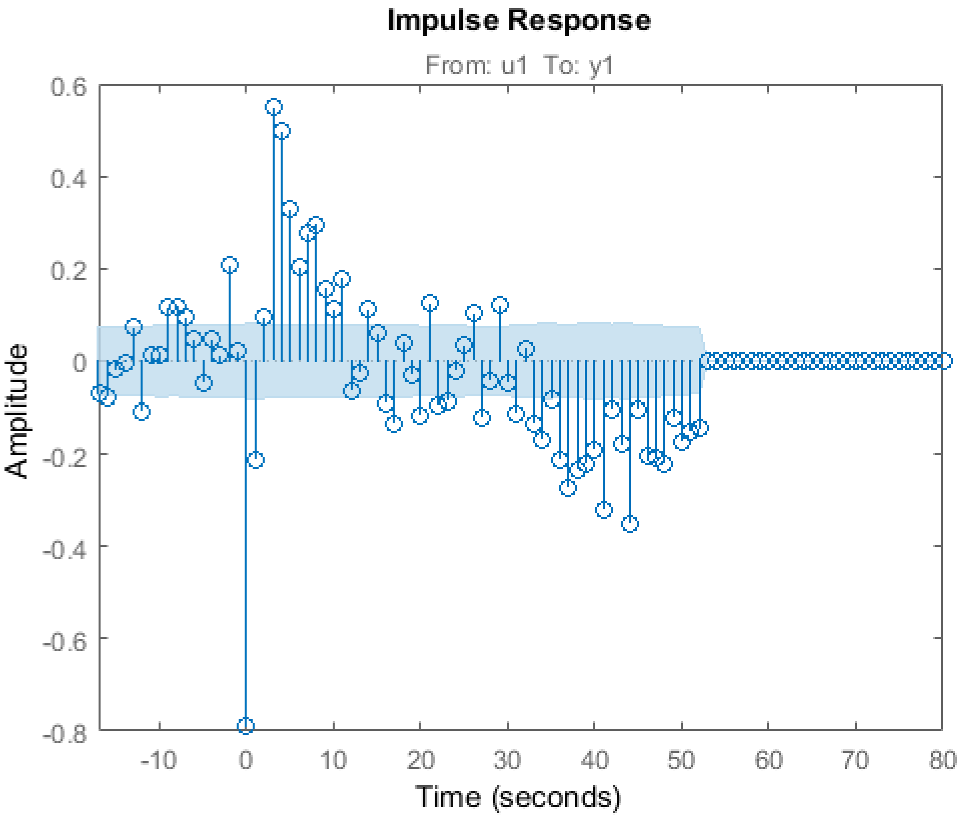
Analyze the impulse response of the identified model.

```
h = impulseplot(sys);
```



View the statistically zero-response region by right-clicking on the plot and selecting **Characteristics > Confidence Region**. Alternatively, you can use the `showConfidence` command.

```
showConfidence(h);
```



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. There could also be 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.

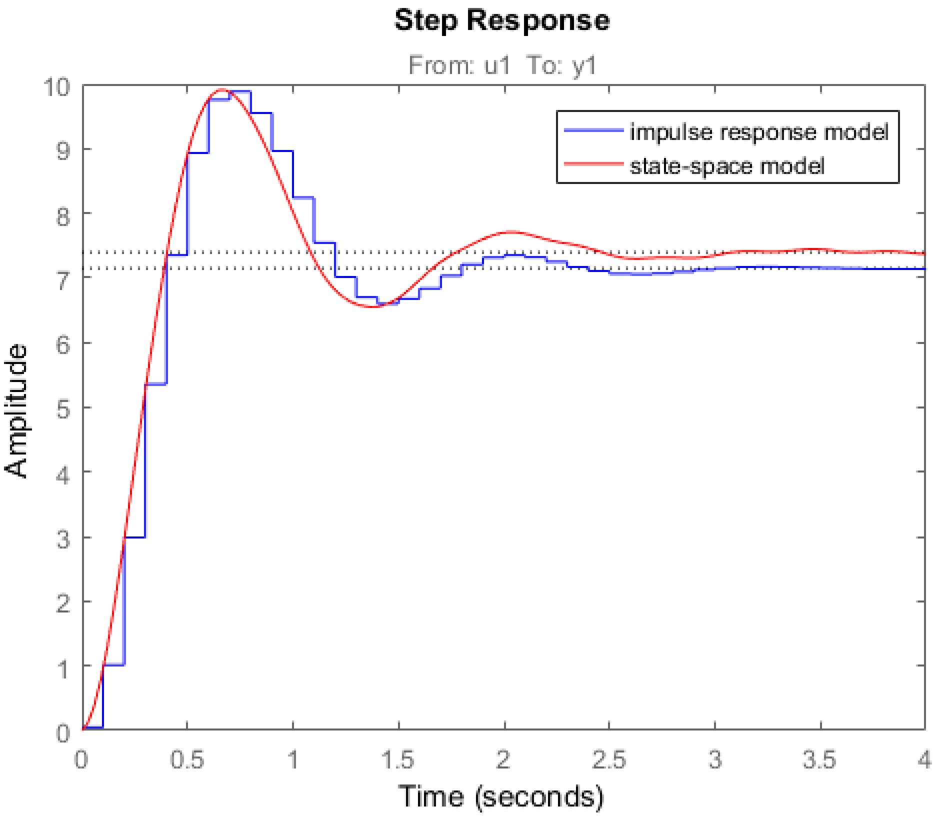
Identify the impulse response model (nonparametric) and state-space model (parametric), based on a data set.

```
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');
```



More About

Tips

- To view the impulse or step response of `sys`, use either `impulseplot` 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.

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`, where `A` is a polynomial and `e` is white noise.
  2. The inputs and outputs are filtered using the filter:

     `uf = Au`, `yf = Ay`

3. The filtered signals `uf` and `yf` are used for estimation.

You can specify prewhitening using the `PW` name-value pair argument of `impulseestOptions`.

- **Regularization** — The least-squares estimate can be regularized. This means that a prior estimate of the decay and mutual correlation among `g(k)` is formed and used to merge with the information about `g` from the observed data. This gives an estimate with less variance, at the price of some bias. You can choose one of the several kernels to encode the prior estimate.

  This option is essential because, often, the model order `n` can be quite large. In cases where there is no regularization, `n` can be automatically decreased to secure a reasonable variance.

  You can specify the regularizing kernel using the `RegulKernel` Name-Value pair argument of `impulseestOptions`.
- **Autoregressive Parameters** — The basic underlying FIR model can be complemented by `NA` autoregressive parameters, making it an ARX model.

  $$y(t) = \sum_{k=1}^{n} g(k)u(t-k) - \sum_{k=1}^{NA} a_k y(t-k)$$

  This gives both better results for small `n` and allows unbiased estimates when data are generated in closed loop. `impulseest` uses NA = 5 for t>0 and NA = 0 (no autoregressive component) for t<0.
- **Noncausal effects** — Response for negative lags. It may happen that the data has been generated partly by output feedback:

  $$u(t) = \sum_{k=0}^{\infty} h(k)y(t-k) + r(t)$$

  where $h(k)$ is the impulse response of the regulator and r is a setpoint or disturbance term. The existence and character of such feedback h can be estimated in the same way as g, simply by trading places between y and u in the estimation call. Using `impulseest` with an indication of negative delays, $\mathrm{mi} \ = \ \mathrm{impulseest}(data, nk, nb),\ nk < 0$, returns a model `mi` with an impulse response

$$\left[h(\text{-}nk), h(\text{-}nk\text{-}1), \ldots, h(0), g(1), g(2), \ldots, g(nb+nk)\right]$$

aligned so that it corresponds to lags $\left[nk, nk+1, .., 0, 1, 2, \ldots, nb+nk\right]$. This is achieved because the input delay (`InputDelay`) of model `mi` is `nk`.

For a multi-input multi-output system, the impulse response $g(k)$ is an *ny*-by-*nu* matrix, where *ny* is the number of outputs and *nu* is the number of inputs. The *i*–*j* element of the matrix $g(k)$ describes the behavior of the *i*th output after an impulse in the *j*th input.

- "What Is Time-Domain Correlation Analysis?"

See Also

`cra` | `impulse` | `impulseestOptions` | `spa` | `step`

Introduced in R2012a

impulseestOptions

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.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

`'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].

Regularization is specified as one of the following values:

- `'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 `Nu`-element column vector, where `Nu` is the number of inputs. For multi-experiment data, specify a `Nu`-by-`Ne` matrix, where `Ne` is the number of experiments. The offset value `InputOffset(i,j)` is subtracted from the i^{th} input signal of the j^{th} experiment.
- `[]` — No offsets.

Default: `[]`

`'OutputOffset'`

Output signal offset level of time-domain estimation data. Must be one of the following:

- An `Ny`-element column vector, where `Ny` is the number of outputs. For multi-experiment data, specify a `Ny`-by-`Ne` matrix, where `Ne` is the number of experiments. The offset value `OputOffset(i,j)` is subtracted from the i^{th} output signal of the j^{th} experiment.
- `[]` — No offsets.

Default: `[]`

`'Advanced'`

Structure, used during regularized estimation, 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, specified as one of the following values:
  - `'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.
  - `'gn'`: Quasi-Newton line search.

  `SearchMethod` is used only when `RegulKernel` is not `'none'`.

  Default: `'fmincon'`

  If you do not have Optimization Toolbox software, the default is `'gn'`.
- `AROrder` — Order of the AR-part in the model from input to output. Specify as a positive integer.

  An order>0 allows more accurate models of the impulse response in case of feedback and non-white output disturbances.

  Default: `5`
- `FeedthroughInSys` — Specify whether the impulse response value at zero lag must be attributed to feedthrough in the system (`true`) or to feedback effects (`false`). Applies only when you compute the response values for negative lags.

  Default: `false`

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;
```

More About

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('InputOffest',in_off,'OuputOffset',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)
```

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`

Introduced in R2012b

impulseplot

Plot impulse response and return plot handle

Syntax

```
impulseplot(sys)
impulseplot(sys,Tfinal)
impulseplot(sys,t)
impulseplot(sys1,sys2,...,sysN)
impulseplot(sys1,sys2,...,sysN,Tfinal)
impulseplot(sys1,sys2,...,sysN,t)
impulseplot(AX,...)
impulseplot(..., plotoptions)
h = impulseplot(...)
```

Description

`impulseplot` 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. `impulseplot` 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.

`impulseplot(sys)` plots the impulse response of the LTI model without returning the plot handle.

`impulseplot(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 sample time (`Ts = -1`), `impulseplot` interprets `Tfinal` as the number of sampling intervals to simulate.

`impulseplot(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 `impulse`). The `impulseplot` command always applies the impulse at `t=0`, regardless of `Ti`.

To plot the impulse response of multiple LTI models `sys1`,`sys2`,... on a single plot, use:

`impulseplot(sys1,sys2,...,sysN)`

`impulseplot(sys1,sys2,...,sysN,Tfinal)`

`impulseplot(sys1,sys2,...,sysN,t)`

You can also specify a color, line style, and marker for each system, as in

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

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

`impulseplot(..., plotoptions)` plots the impulse response with the options specified in `plotoptions`. Type

```
help timeoptions
```

for more detail.

`h = impulseplot(...)` plots the impulse response and returns the plot handle `h`.

Examples

Example 1

Normalize the impulse response of a third-order system.

```
sys = rss(3);
h = impulseplot(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', 'dcmotordata'));
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);
```

More About

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”.

See Also

`showConfidence` | `getoptions` | `impulse` | `setoptions`

Introduced in R2012a

init

Set or randomize initial parameter values

Syntax

```
m = init(m0)
m = init(m0,R,pars,sp)
```

Description

`m = init(m0)` 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`.

`m = init(m0,R,pars,sp)` randomizes parameters around `pars` with variances given by the row vector `R`. Parameter number *k* is randomized as `pars(k)` + `e*sqrt(R(k))`, where `e` is a normal random variable with zero mean and a variance of 1. The default value of `R` is all ones, and the default value of `pars` is the nominal parameter vector in `m0`.

Only models that give stable predictors are accepted. If `sp = 'b'`, only models that are both stable and have stable predictors are accepted.

`sp = 's'` requires stability only of the model, and `sp = 'p'` requires stability only of the predictor. `sp = 'p'` is the default.

Sufficiently free parameterizations can be stabilized by direct means without any random search. To just stabilize such an initial model, set `R = 0`. With `R > 0`, randomization is also done.

For model structures where a random search is necessary to find a stable model/predictor, a maximum of 100 trials is made by `init`. It can be difficult to find a stable predictor for high-order systems by trial and error.

See Also

`idnlarx` | `idnlhw` | `rsample` | `simsd`

Introduced before R2006a

interp

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

`idfrd` | `freqresp` | `frd`

Introduced in R2012a

iopzmap

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.

Examples

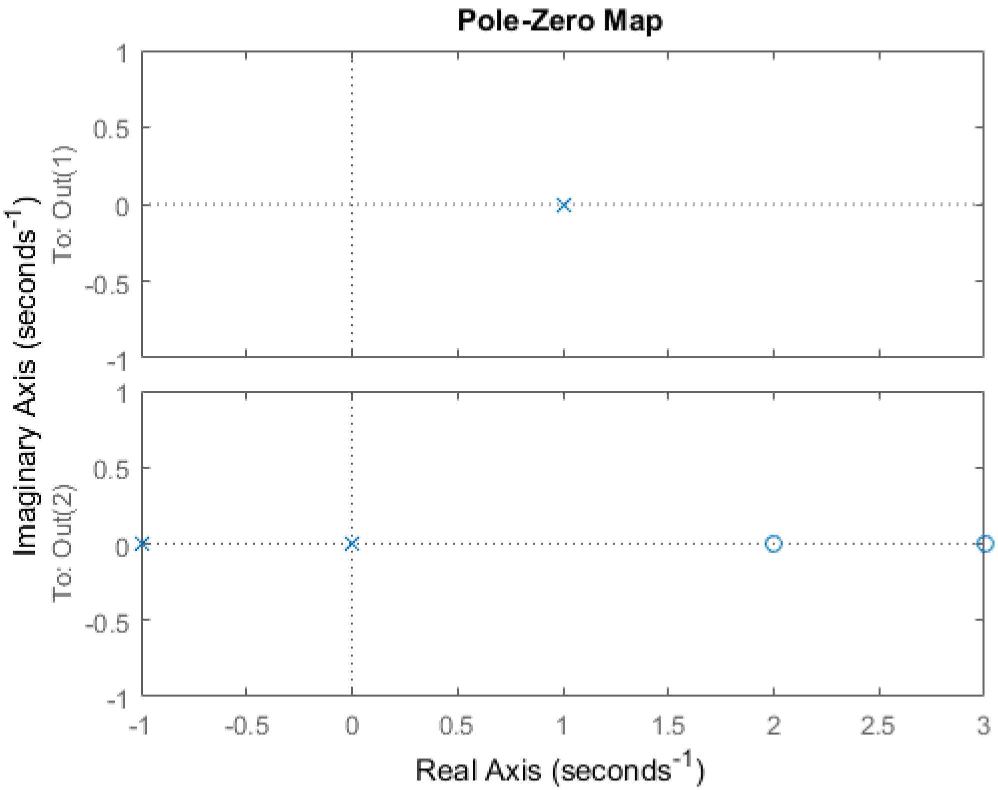
Pole-Zero Map for MIMO System

Create a one-input, two-output dynamic system.

```
H = [tf(-5 ,[1 -1]); tf([1 -5 6],[1 1 0])];
```

Plot a pole-zero map.

```
iopzmap(H)
```

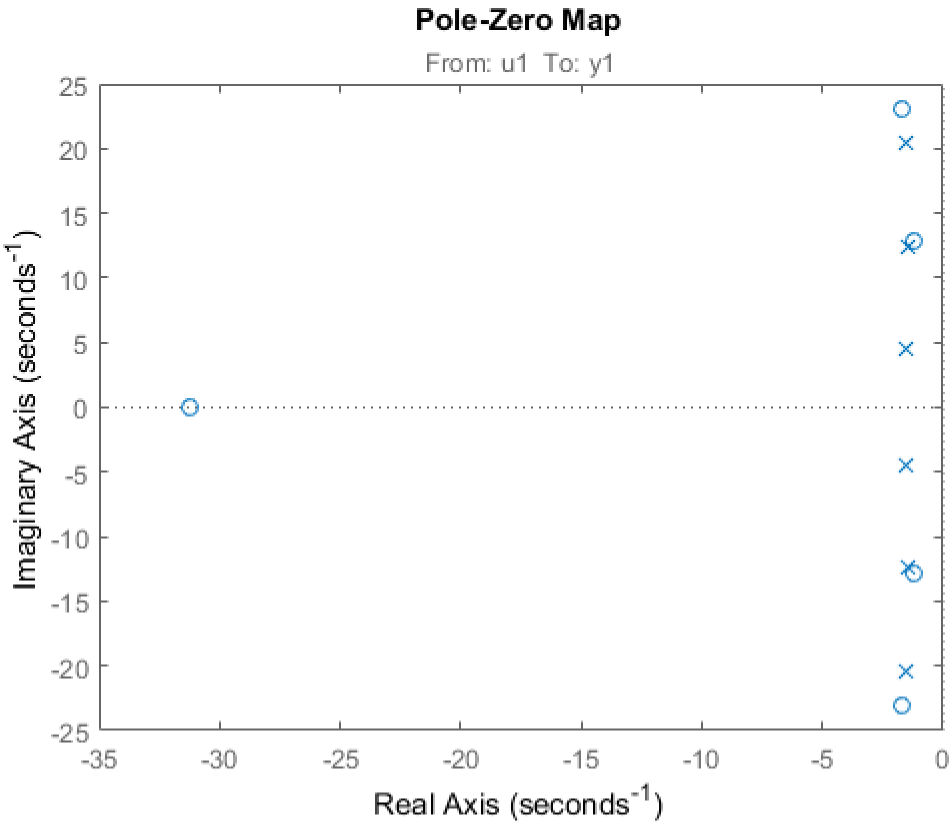


`iopzmap` generates a separate map for each I/O pair in the system.

Pole-Zero Map of Identified Model

View the poles and zeros of an over-parameterized state-space model estimated from input-output data. (Requires System Identification Toolbox™).

```
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.

More About

Tips

For additional options for customizing the appearance of the pole-zero plot, use `iopzplot`.

See Also

`pzmap` | `pole` | `zero` | `sgrid` | `zgrid` | `iopzplot`

Introduced in R2012a

iopzplot

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 dynamic system 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. For more information on the ways to change properties of your plots, see "Ways to Customize Plots".

`iopzplot(sys1,sys2,...)` shows the poles and zeros of multiple dynamic system 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.

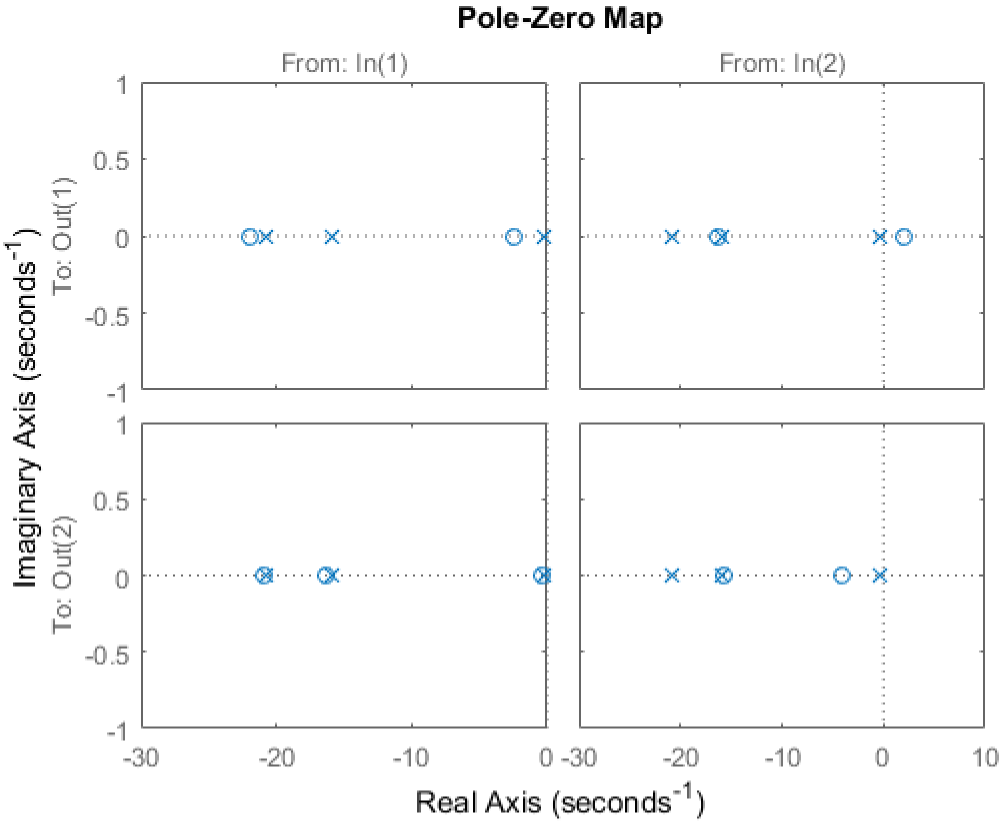
For arrays `sys` of LTI models, `iopzplot` plots the poles and zeros of each model in the array on the same diagram.

Examples

Change I/O Grouping on Pole/Zero Map

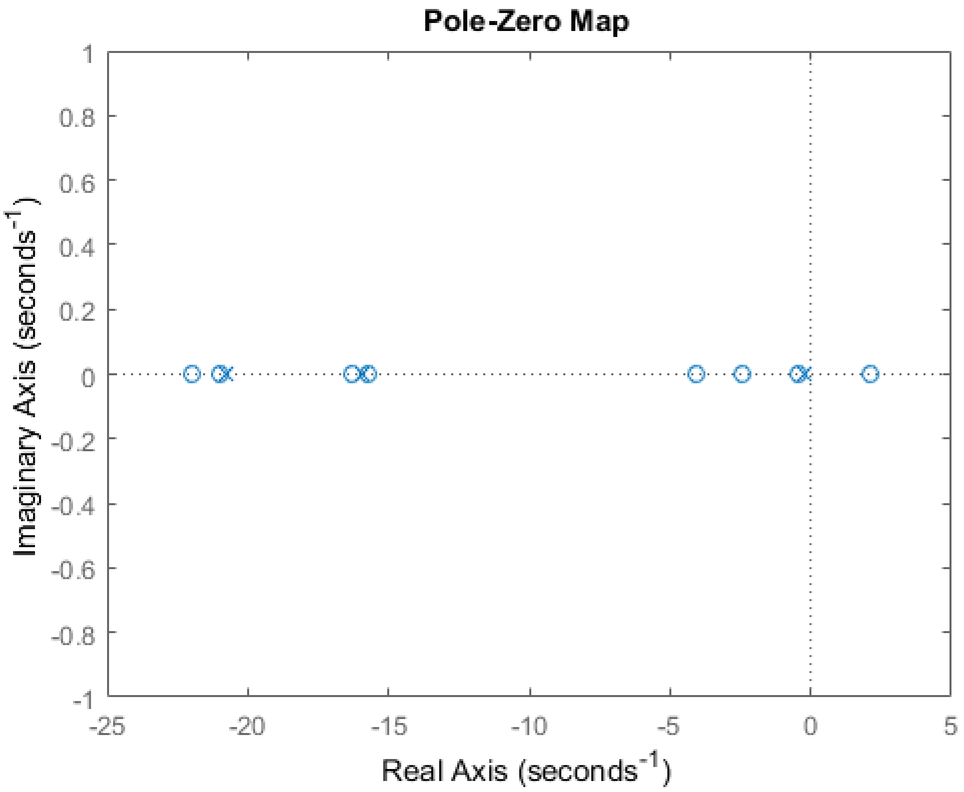
Create a pole/zero map of a two-input, two-output dynamic system.

```
sys = rss(3,2,2);
h = iopzplot(sys);
```



By default, the plot displays the poles and zeros of each I/O pair on its own axis. Use the plot handle to view all I/Os on a single axis.

```
setoptions(h,'IOGrouping','all')
```

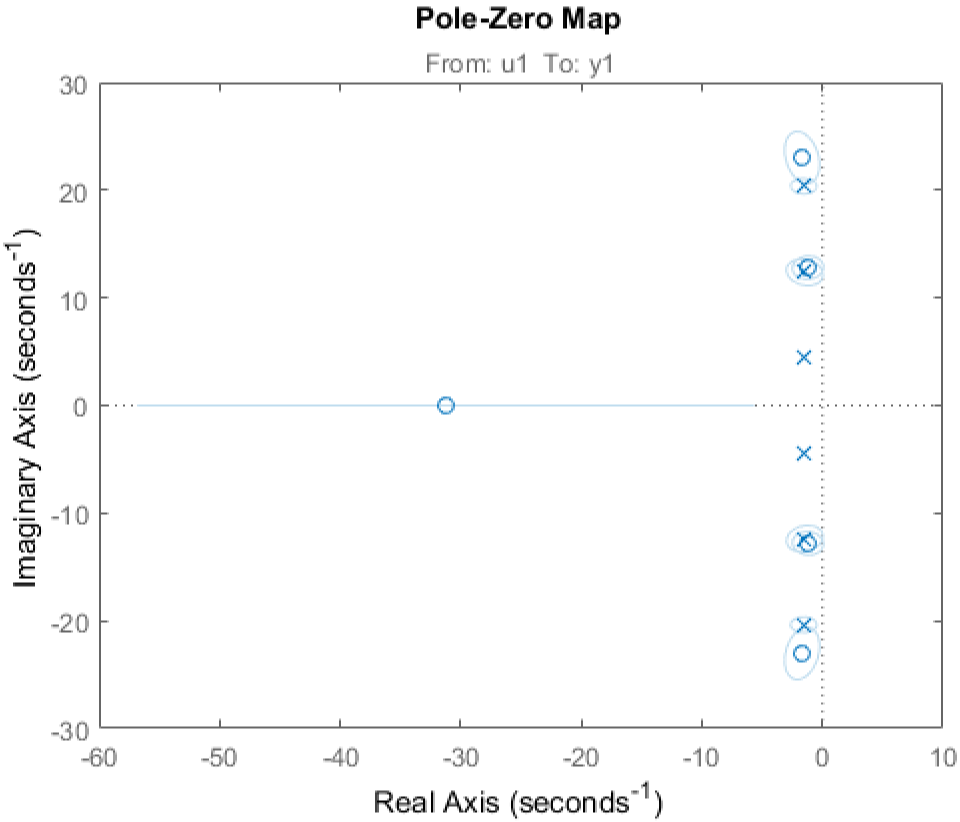


Use Pole-Zero Map to Examine Identified Model

View the poles and zeros of a sixth-order state-space model estimated from input-output data. Use the plot handle to display the confidence intervals of the identified model's pole and zero locations.

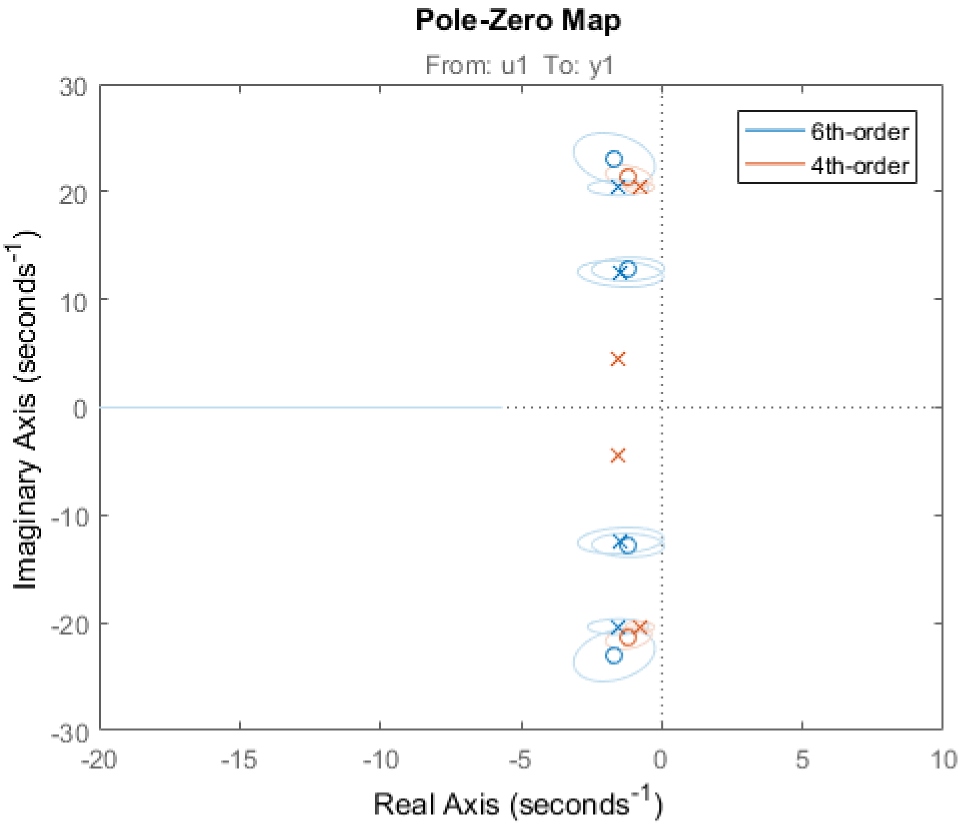
```
load iddata1
sys = ssest(z1,6,ssestOptions('focus','simulation'));
```

```
h = iopzplot(sys);
showConfidence(h)
```



There is at least one pair of complex-conjugate poles whose locations overlap with those of a complex zero, within the 1-σ 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 = iopzplot(sys,sys2);
showConfidence(h)
legend('6th-order','4th-order')
axis([-20, 10 -30 30])
```



The fourth-order model `sys2` shows less variability in the pole-zero locations.

See Also

`showConfidence` | `getoptions` | `iopzmap` | `setoptions`

Introduced in R2012a

isct

Determine if dynamic system model is in continuous time

Syntax

```
bool = isct(sys)
```

Description

`bool = isct(sys)` returns a logical value of `1` (`true`) if the dynamic system model `sys` is a continuous-time model. The function returns a logical value of `0` (`false`) otherwise.

Input Arguments

`sys`

Dynamic system model or array of such models.

Output Arguments

`bool`

Logical value indicating whether `sys` is a continuous-time model.

`bool` = `1` (`true`) if `sys` is a continuous-time model (`sys.Ts = 0`). If `sys` is a discrete-time model, `bool` = `0` (`false`).

For a static gain, both `isct` and `isdt` return `true` unless you explicitly set the sample time to a nonzero value. If you do so, `isdt` returns `true` and `isct` returns `false`.

For arrays of models, `bool` is `true` if the models in the array are continuous.

See Also

`isdt` | `isstable`

Introduced in R2012a

isdt

Determine if dynamic system model is in discrete time

Syntax

```
bool = isdt(sys)
```

Description

`bool = isdt(sys)` returns a logical value of `1` (`true`) if the dynamic system model `sys` is a discrete-time model. The function returns a logical value of `0` (`false`) otherwise.

Input Arguments

`sys`

Dynamic system model or array of such models.

Output Arguments

`bool`

Logical value indicating whether `sys` is a discrete-time model.

`bool` = `1` (`true`) if `sys` is a discrete-time model (`sys.Ts ≠ 0`). If `sys` is a continuous-time model, `bool` = `0` (`false`).

For a static gain, both `isct` and `isdt` return `true` unless you explicitly set the sample time to a nonzero value. If you do so, `isdt` returns `true` and `isct` returns `false`.

For arrays of models, `bool` is `true` if the models in the array are discrete.

See Also

`isct` | `isstable`

Introduced in R2012a

isempty

Determine whether dynamic system model is empty

Syntax

`isempty(sys)`

Description

`isempty(sys)` returns a logical value of `1` (`true`) if the dynamic system model `sys` has no input or no output, and a logical value of `0` (`false`) otherwise. Where `sys` is a `frd` model, `isempty(sys)` returns `1` when the frequency vector is empty. Where `sys` is a model array, `isempty(sys)` returns `1` 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 `1` while

```
isempty(ss(1,2,3,4))
```

returns `0`.

See Also

`issiso` | `size`

Introduced before R2006a

isLocked

Locked status of online parameter estimation System object

Syntax

```
L = isLocked(obj)
```

Description

`L = isLocked(obj)` returns the locked status of online parameter estimation System object, `obj`.

Examples

Check Locked Status of Online Estimation System Object

Create a System object™ for online estimation of an ARMAX model with default properties.

```
obj = recursiveARMAX;
```

Check the locked status of the object.

```
L = isLocked(obj)
```

```
L =

  logical

   0
```

Estimate model parameters online using `step` and input-output data.

```
[A,B,C,EstimatedOutput] = step(obj,1,1);
```

Check the locked status of the object again.

```
L = isLocked(obj)
```

```
L =

  logical

   1
```

`step` puts the object in a locked state.

Input Arguments

`obj` — System object for online parameter estimation

`recursiveAR` object | `recursiveARMA` object | `recursiveARX` object | `recursiveARMAX` object | `recursiveOE` object | `recursiveBJ` object | `recursiveLS` object

System object for online parameter estimation, created using one of the following commands:

- `recursiveAR`
- `recursiveARMA`
- `recursiveARX`
- `recursiveARMAX`
- `recursiveOE`
- `recursiveBJ`
- `recursiveLS`

Output Arguments

`L` — Locked status of online estimation System object

logical

Locked status of online estimation System object, returned as a logical value. `L` is `true` if `obj` is locked.

More About

- “What Is Online Estimation?”

See Also

`clone` | `recursiveAR` | `recursiveARMA` | `recursiveARMAX` | `recursiveARX` | `recursiveBJ` | `recursiveLS` | `recursiveOE` | `release` | `reset` | `step`

Introduced in R2015b

isnlarx

Detect nonlinearity in estimation data

Syntax

```
isnlarx(Data,Orders)
isnlarx(Data,Orders,Ky)
isnlarx(___,Name,Value)

NLHyp = isnlarx(___)
[NLHyp,NLValue,NLRegs,NoiseSigma,DetectRatio] = isnlarx(___)
```

Description

`isnlarx(Data,Orders)` detects nonlinearity in `Data` by testing whether a nonlinear ARX model with the indicated `Orders` produces a better estimate of `Data` than a linear ARX model. The nonlinear model uses a default `treepartition` nonlinearity estimator.

The result of the test is printed to the Command Window and indicates whether a nonlinearity is detected. Use the printed detection ratio to assess the reliability of the nonlinearity detection test:

- Larger values (>`2`) indicate that a significant nonlinearity was detected.
- Smaller values (<`0.5`) indicate that any error unexplained by the linear model is mostly noise. That is, no significant nonlinearity was detected.
- Values close to `1` indicate that the nonlinearity detection test is not reliable and that a weak nonlinearity may be present.

`isnlarx(Data,Orders,Ky)` restricts the nonlinearity test to output channel `Ky` for multi-output data.

`isnlarx(___,Name,Value)` specifies additional nonlinear ARX model options using one or more `Name,Value` pair arguments.

`NLHyp = isnlarx(___)` returns the result of the nonlinearity test and suppresses the command window output.

`[NLHyp,NLValue,NLRegs,NoiseSigma,DetectRatio] = isnlarx(___)` additionally returns the test quantities behind the evaluation.

Examples

Detect Nonlinearity in Estimation Data

Load the signal transmission data set.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'frictiondata'));
```

Construct an `iddata` object from the estimation data.

```
z = iddata(f1,v,1);
```

Specify the model orders and delays.

```
orders = [1 1 0];
```

Run the test to detect nonlinearity.

```
isnlarx(z,orders);
```

```
Nonlinearity is detected in data set z
Detection ratio: 525.05
Estimated discrepancy of the linear model found: 0.0064966
Estimated noise standard deviation: 0.00080938
```

The large detection ratio indicates that the test was robust and a significant nonlinearity was detected. Additionally, the estimated discrepency of the linear model that was found, that is the data explained by the nonlinearity, is significantly greater than the noise error, which can indicate a significant nonlinearity.

Detect Nonlinearity in Estimation Data Output Channel

Load the CSTR data set.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'cstrdata'));
```

Construct an `iddata` object from the estimation data using a sample time of 0.1 seconds .

```
z = iddata(y1,u1,0.1);
```

Specify the model orders and delays.

```
orders = [3*ones(2,2),ones(2,3),2*ones(2,3)];
```

Run the test to detect nonlinearity on the second output channel.

```
isnlarx(z,orders,2);
```

```
ISNLARX results for dataset z

Nonlinearity is not detected in channel (2).
However, the test may be on the edge of detecting the nonlinearity.
Detection ratio: 0.36446
Searching for best nonlinear regressors may provide more reliable results.
-
```

A detection ratio less than `1` indicates that no nonlinearity was detected. However, since this value is near 0.5, there may be a weak nonlinearity that was not detected by the test.

Search for Best Regressors When Detecting Nonlinearity

Load the signal transmission data set.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'signaltransmissiondat
```

Construct an `iddata` object from the estimation data using a sample time of 0.1 seconds.

```
z = iddata(vout,vin,0.1);
```

Specify the model orders and delays.

```
orders = [3 0 2];
```

Display the model regressors for an `idnlarx` model with the given orders.

```
getreg(idnlarx(orders));
```

```
Regressors:
    y1(t-1)
    y1(t-2)
    y1(t-3)
```

Detect nonlinearities in the data, and search for the best nonlinear regressor combination.

```
isnlarx(z,orders,'NonlinearRegressors','search');
```

```
Nonlinearity is detected in data set z
Detection ratio: 1.4691
Estimated discrepancy of the linear model found: 0.42844
Estimated noise standard deviation: 0.92937
Corresponding NonlinearRegressors parameter: [1           2]
```

The regressor search found that using the first two regressors produces the best nonlinear estimation of the given data.

A detection ratio greater than `1` but less than `2` means that a nonlinearity was detected, but the test was not robust. This result may indicate that the detected nonlinearity is not significant. Additionally, the data explained by the nonlinearity is smaller than the noise error, which can be an indication of a weak nonlinearity.

Return Nonlinearity Detection Result

Load the estimation data.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'cstrdata'));
```

Construct an `iddata` object using the estimation data.

```
z = iddata(y1,u1,0.1);
```

Specify the model orders and delays.

```
orders = [3*ones(2,2),ones(2,3),2*ones(2,3)];
```

Detect nonlinearities in the data, and determine the test quantities behind the evaluation.

```
NLHyp = isnlarx(z,orders);
```

Return Nonlinearity Detection Test Quantities

Load the estimation data.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data','narendralidata'))
```

Construct an `iddata` object using the estimation data.

```
z = iddata(u,y1,1);
```

Specify the model orders and delays.

```
orders = [1 1 2];
```

Detect nonlinearities in the data, and determine the test quantities behind the evaluation.

```
[NLHyp,NLValue,NLRegs,NoiseSigma,DetectRatio] = isnlarx(z,orders);
```

Input Arguments

`Data` — Time-domain estimation data

`iddata` object

Time-domain estimation data, specified as an `iddata` object. `Data` can have one or more output channels and zero or more input channels. Data must be uniformly sampled and cannot contain missing (`NaN`) samples.

`Orders` — Model orders and delays

1-by-3 vector of positive integers | 1-by-3 vector of matrices

Model orders and delays for defining the regressor configuration, specified as a 1-by-3 vector, `[na nb nk]`.

For a model with n_y output channels and n_u input channels:

- `na` is an n_y-by-n_y matrix, where `na(i,j)` specifies the number of regressors from the *j*th output used to predict the *i*th output.
- `nb` is an n_y-by-n_u matrix, where `nb(i,j)` specifies the number of regressors from the *j*th input used to predict the *i*th output.
- `nk` is an n_y-by-n_u matrix, where `nk(i,j)` specifies the lag in the *j*th input used to predict the *i*th output.

```
na = [1 2; 2 3]
nb = [1 2 3; 2 3 1];
nk = [2 0 3; 1 0 5];
```

The estimation data for this system has three inputs (`u1`, `u2`, `u3`) and two outputs (`y1`, `y2`). Consider the regressors used to predict output, `y2(t)`:

- Since `na(2,:)` is `[2 3]`, the contributing regressors from the outputs are:

  - `y1(t-1)` and `y1(t-2)`
  - `y2(t-1)`, `y2(t-2)`, and `y2(t-3)`
- Since `nb(2,:)` is `[2 3 1]` and `nk(2,:)` is `[1 0 5]`, the contributing regressors from the inputs are:
  - `u1(t-1)` and `u1(t-2)`
  - `u2(t)`, `u2(t-1)`, and `u2(t-2)`
  - `u3(t-5)`

Note: The minimum lag for regressors based on output variables is always `1`, while the minimum lag for regressors based on input variables is dictated by `nk`. Use `getreg` to view the complete set of regressors used by the nonlinear ARX model.

Ky — Output channel number in estimation data

positive integer in the range [0,n_y]

Output channel number in estimation data, specified as a positive integer in the range [1,n_y], where n_y is the number of output channels.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example: `'NonlinearRegressors','output'` specifies that only the regressors containing output variables are used as inputs to the nonlinear block of the model.

`'TimeVariable'` — Independent variable name

`'t'` (default) | character vector

Independent variable name, specified as the comma-separated pair consisting of `'TimeVariable'` and a character vector. For example, `'t'`.

`'CustomRegressors'` — Regressors constructed from combinations of inputs and outputs

`{}` (default) | cell array of character vectors | array of `customreg` objects

Regressors constructed from combinations of inputs and outputs, specified as the comma-separated pair consisting of `'CustomRegressors'` and one of the following for single-output systems:

- Cell array of character vectors. For example:
  - `{’y1(t-3)^3’,'y2(t-1)*u1(t-3)','sin(u3(t-2))'}`

  Each character vector must represent a valid formula for a regressor contributing towards the prediction of the model output. The formula must be written using the input and output names and the time variable name as variables.
- Array of custom regressor objects, created using `customreg` or `polyreg`.

For a model with n_y outputs, specify an n_y-by-1 cell array of `customreg` object arrays or character arrays.

These regressors are in addition to the standard regressors based on `Orders`.

Example: `'CustomRegressors',{’y1(t-3)^3’,'y2(t-1)*u1(t-3)'}`

Example: `'CustomRegressors',{'sin(u3(t-2))'}`

`'NonlinearRegressors'` — Subset of regressors that enter as inputs to the nonlinear block of the model

`'all'` (default) | `'output'` | `'input'` | `'standard'` | `'custom'` | `'search'` | vector of positive integers | `[]` | cell array

Subset of regressors that enter as inputs to the nonlinear block of the model, specified as the comma-separated pair consisting of `'NonlinearRegressors'` and one of the following values:

- `'all'` — All regressors
- `'output'` — Regressors containing output variables
- `'input'` — Regressors containing input variables
- `'standard'` — Standard regressors
- `'custom'` — Custom regressors
- `'search'` — The estimation algorithm performs a search for the best regressor subset. This is useful when you want to reduce a large number of regressors entering the nonlinear function block of the nonlinearity estimator. This option must be applied to all output models simultaneously.

- `[]` — No regressors. This creates a linear-in-regressor model.
- Vector of regressor indices. To determine the number and order of regressors, use `getreg`.

For a model with multiple outputs, specify a cell array of n_y elements, where n_y is the number of output channels. For each output, specify one of the preceding options. Alternatively, to apply the same regressor subset to all model outputs, specify `[]` or any of the character vector options alone, for example `'standard'`.

Example: `'NonlinearRegressors','search'` performs a best regressor search for the only output of a single output model, or all of the outputs of a multiple output model.

Example: `'NonlinearReg','input'` applies only input regressors to the inputs of the nonlinear function.

Example: `'NonlinearRegressors',{'input','output'}` applies input regressors to the first output, and output regressors to the second output of a model with two outputs.

Output Arguments

`NLHyp` — Result of the nonlinearity test

0 | 1 | logical vector

Result of the nonlinearity test, returned as a logical vector with length equal to the number of output channels. The elements of `NLHyp` are `1` if nonlinearities were detected for the corresponding output. A value of `0` indicates that nonlinearities were not detected.

`NLValue` — Estimated standard deviation of the data explained by the nonlinearity

vector of nonnegative scalars

Estimated standard deviation of the data explained by the nonlinearity, returned as a vector of nonnegative scalars with length equal to the number of output channels. The elements of `NLValue` are `0` if nonlinearities are not detected for the corresponding output.

`NLRegs` — Regressors that should enter nonlinearly in the model

vector of indices | `[]` | cell array

Regressors that should enter nonlinearly in the model, returned as a vector of indices for single output models. For multi-output models, `NLRegs` is returned as a cell array, with elements corresponding to each output channel. `NLRegs` is empty, `[]`, if nonlinearities are not detected.

See the `'NonlinearRegressors'` `Name,Value` argument for more information.

`NoiseSigma` — Estimated standard deviation of the unexplained error
vector of nonnegative scalars

Estimated standard deviation of the unexplained error, returned as a vector of nonnegative scalars with length equal to the number of output channels. The elements of `NoiseSigma` are `0` if nonlinearities are not detected for the corresponding output.

`DetectRatio` — Ratio of the test statistic and the detection threshold
vector

Ratio of the test statistic and the detection threshold, returned as a vector with length equal to the number of output channels. Use the elements of `DetectRatio` to assess the reliability of the nonlinearity detection test for the corresponding output:

- Larger values (`>2`) indicate that a significant nonlinearity was detected.
- Smaller values (`<0.5`) indicate that any error unexplained by the linear model is mostly noise. That is, no significant nonlinearity was detected.
- Values close to `1` indicate that the nonlinearity detection test is not reliable and that a weak nonlinearity may be present.

More About

Algorithms

`isnlarx` estimates a nonlinear ARX model using the given data and a `treepartition` nonlinearity estimator.

The estimation data can be described as $Y(t) = L(t) + F_n(t) + E(t)$, where:

- $L(t)$ is the portion of the data explained by the linear function of the nonlinear ARX model.
- $F_n(t)$ is the portion of the data explained by the nonlinear function of the nonlinear ARX model. The output argument `NLValue` is an estimate of the standard deviation of $F_n(t)$. If the nonlinear function explains a significant portion of the data beyond the data explained by the linear function, a nonlinearity is detected.

- $E(t)$ is the remaining error that is unexplained by the nonlinear ARX model and is typically white noise. The output argument `NoiseSigma` is an estimate of the standard deviation of $E(t)$.
- "Structure of Nonlinear ARX Models"

See Also

`getreg` | `idnlarx` | `nlarx` | `treepartition`

Introduced in R2007a

isproper

Determine if dynamic system model is proper

Syntax

```
B = isproper(sys)
B = isproper(sys,'elem')
[B,sysr] = isproper(sys)
```

Description

`B = isproper(sys)` returns a logical value of `1` (`true`) if the dynamic system model `sys` is proper and a logical value of `0` (`false`) otherwise.

A proper model has relative degree ≤ 0 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.

If `sys` is a model array, then `B` is `1` if all models in the array are proper.

`B = isproper(sys,'elem')` checks each model in a model array `sys` and returns a logical array of the same size as `sys`. The logical array indicates which models in `sys` are proper.

`[B,sysr] = isproper(sys)` also returns an equivalent model `sysr` with fewer states (reduced order) and a non-singular `E` matrix, if `sys` is a proper descriptor state-space model with a non-invertible `E` matrix. If `sys` is not proper, `sysr = sys`.

Examples

Examine Whether Models are Proper

The following commands

```
B1 = isproper(tf([1 0],1))         % transfer function s
B2 = isproper(tf([1 0],[1 1]))    % transfer function s/(s+1)
```

return `0` (`false`) and `1` (`true`), respectively.

Compute Equivalent Lower-Order Model

Combining state-space models sometimes yields 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;
size(H)
```

```
State-space model with 1 outputs, 1 inputs, and 4 states.
```

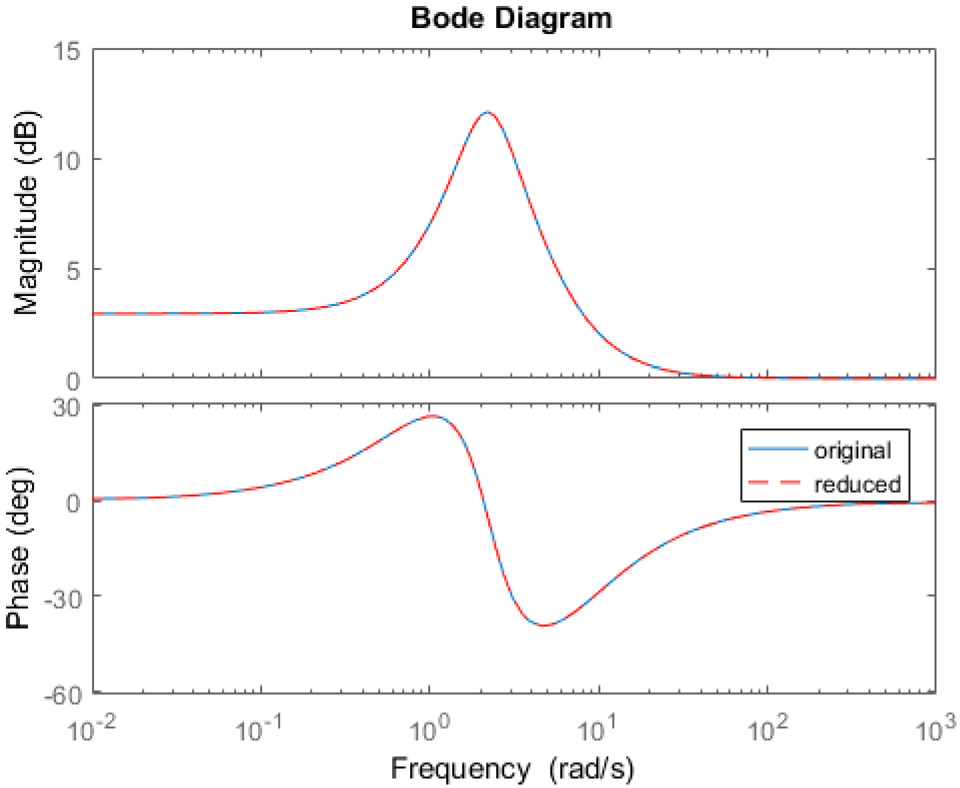
`H` is proper and reducible. `isproper` returns the reduced model.

```
[isprop,Hr] = isproper(H);
size(Hr)
```

```
State-space model with 1 outputs, 1 inputs, and 2 states.
```

`H` and `Hr` are equivalent, as a Bode plot demonstrates.

```
bodeplot(H,Hr,'r--')
legend('original','reduced')
```



See Also

`ss` | `dss`

Introduced before R2006a

isreal

Determine whether model parameters or data values are real

Syntax

```
isreal(Data)
isreal(Model)
```

Description

`isreal(Data)` returns `1` if all signals of the data set are real. `Data` is an `iddata` object.

`isreal(Model)` returns `1` if all parameters of the model are real. `Model` is any linear identified model.

See Also

`realdata`

Introduced before R2006a

issiso

Determine if dynamic system model is single-input/single-output (SISO)

Syntax

`issiso(sys)`

Description

`issiso(sys)` returns a logical value of `1` (`true`) if the dynamic system model `sys` is SISO and a logical value of `0` (`false`) otherwise.

See Also

`isempty` | `size`

Introduced in R2012a

isstable

Determine whether system is stable

Syntax

```
B = isstable(sys)
B = isstable(sys,'elem')
```

Description

`B = isstable(sys)` returns a logical value of `1` (`true`) if the dynamic system model `sys` has stable dynamics, and a logical value of `0` (`false`) otherwise. If `sys` is a model array, then `B` = `1` only if all models in `sys` are stable.

`B = isstable(sys,'elem')` returns a logical array of the same dimensions 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.

Examples

Determine Stability of Models in Model Array

Create an array of SISO transfer function models with poles varying from -2 to 2. To do so, first initialize an array of dimension `[1,length(a)]` with zero-valued SISO transfer functions.

```
a = [-2:2];
sys = tf(zeros(1,1,1,length(a)));
```

Populate this array with transfer functions of the form `1/(s-a)`.

```
for j = 1:length(a)
    sys(1,1,1,j) = tf(1,[1 -a(j)]);
end
sys.SamplingGrid = struct('a',a);
```

Examine the stability of the model array.

```
B_all = isstable(sys)
```

```
B_all =

  logical

   0
```

By default, `isstable` returns a single Boolean value that is 1 (`true`) only if all models in the array are stable. `sys` contains some models with nonnegative poles, which are not stable. Therefore, `isstable` returns 0 (`false`) for the entire array.

Examine stability of each model in the array, element by element.

```
B_elem = isstable(sys,'elem')
```

```
B_elem =

  1×5 logical array

   1   1   0   0   0
```

The `'elem'` flag causes `isstable` to return an array of Boolean values, which indicate the stability of the corresponding entry in the model array. For example, `B_elem(2)` = 1, which indicates that `sys(1,1,1,2)` is stable. This result is expected, because `sys(1,1,1,2)` has `a` = -1.

See Also

`pole`

Introduced in R2012a

ivar

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` | `spectrum` | `step`

Introduced before R2006a

ivstruc

Compute loss functions for sets of ARX model structures using instrumental variable method

Syntax

```
v = ivstruc(ze,zv,NN)
v = ivstruc(ze,zv,NN,p,maxsize)
```

Description

`v = ivstruc(ze,zv,NN)` computes the loss functions for sets of single-output ARX model structures. `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 input-output data. Only time-domain data is supported. Models for each model structure defined in `NN` are estimated using the instrumental variable (IV) method on data set `ze`. The estimated models are simulated using the inputs from data set `zv`. The normalized quadratic fit between the simulated output and the measured output in `zv` is formed and returned in `v`. The rows below the first row in `v` are the transpose of `NN`, and the last row contains the logarithms of the condition numbers of the IV matrix

$$\sum \varsigma(t)\varphi^T(t)$$

A large condition number indicates that the structure is of unnecessarily high order (see 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`.

The routine is for single-output systems only.

`v = ivstruc(ze,zv,NN,p,maxsize)` specifies the computation of condition numbers and the size of largest matrix formed during computations. If `p` is equal to zero, the computation of condition numbers is suppressed. `maxsize` affects the speed/memory trade-off.

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

Generate Model-Order Combinations and Estimate ARX Model Using IV Method

Create estimation and validation data sets

```
load iddata1;
ze = z1(1:150);
zv = z1(151:300);
```

Generate model-order combinations for estimation, specifying ranges for model orders and delays.

```
NN = struc(1:3,1:2,2:4);
```

Estimate ARX models using the instrumental variable method, and compute the loss function for each model order combination.

```
V = ivstruc(ze,zv,NN);
```

Select the model order with the best fit to the validation data.

```
order = selstruc(V,0);
```

Estimate an ARX model of selected order.

```
M = iv4(ze,order);
```

Suppress Condition Number Computation When Determining ARX Loss Functions

Create estimation and validation data sets.

```
load iddata1;
ze = z1(1:150);
zv = z1(151:300);
```

Generate model-order combinations for estimation, specifying ranges for model orders and a delay of 2 for all model configurations.

```
NN = struc(2:3,1:2,2);
```

Compute the loss function for each model order combination. Suppress the computation of condition numbers.

```
V = ivstruc(ze,zv,NN,0);
```

More About

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-Hall PTR, 1999.

See Also

`arxstruc` | `struc` | `iv4` | `selstruc`

Introduced before R2006a

ivx

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.

Input Arguments

`data`

Estimation data. The data can be:

- Time- or frequency-domain input-output data
- Time-series data
- Frequency-response data

`data` must be an `iddata`, `idfrd`, or `frd` object.

When 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`

ARX model that fits the estimation data, returned as a discrete-time `idpoly` object. This model is created using the specified model orders, delays, and estimation options. `ivx` does not return any estimated covariance information for `sys`.

Information about the estimation results and options used is stored in the `Report` property of the model. `Report` has the following fields:

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Status</code></td><td>Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.</td></tr>
<tr><td><code>Method</code></td><td>Estimation command used.</td></tr>
<tr><td><code>InitialC</code></td><td>Handling of initial conditions during model estimation, returned as one of the following values:
• <code>'zero'</code> — The initial conditions were set to zero.
• <code>'estimate'</code> — The initial conditions were treated as independent estimation parameters.
• <code>'backcast'</code> — The initial conditions were estimated using the best least squares fit.
This field is especially useful to view how the initial conditions were handled when the <code>InitialCondition</code> option in the estimation option set is <code>'auto'</code>.</td></tr>
<tr><td><code>Fit</code></td><td>Quantitative assessment of the estimation, returned as a structure. See “Loss Function and Model Quality Metrics” for more information on these quality metrics. The structure has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>FitPerc</code></td><td>Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as a percentage.</td></tr>
<tr><td><code>LossFcn</code></td><td>Value of the loss function when the estimation completes.</td></tr>
<tr><td><code>MSE</code></td><td>Mean squared error (MSE) measure of how well the response of the model fits the estimation data.</td></tr>
<tr><td><code>FPE</code></td><td>Final prediction error for the model.</td></tr>
<tr><td><code>AIC</code></td><td>Raw Akaike Information Criteria (AIC) measure of model quality.</td></tr>
<tr><td><code>AICc</code></td><td>Small sample-size corrected AIC.</td></tr>
<tr><td><code>nAIC</code></td><td>Normalized AIC.</td></tr>
<tr><td><code>BIC</code></td><td>Bayesian Information Criteria (BIC).</td></tr>
</table></td></tr>
<tr><td><code>Parameter</code></td><td>Estimated values of model parameters.</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>OptionsUs</code></td><td>Option set used for estimation. If no custom options were configured, this is a set of default options. See <code>arxOptions</code> for more information.</td></tr>
<tr><td><code>RandState</code></td><td>State of the random number stream at the start of estimation. Empty, <code>[]</code>, if randomization was not used during estimation. For more information, see <code>rng</code> in the MATLAB documentation.</td></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation, returned as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
<tr><td><code>Type</code></td><td>Data type.</td></tr>
<tr><td><code>Length</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time.</td></tr>
<tr><td><code>InterSam</code></td><td>Input intersample behavior, returned as one of the following values:
• <code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
• <code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
• <code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.</td></tr>
<tr><td><code>InputOff</code></td><td>Offset removed from time-domain input data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
<tr><td><code>OutputOf</code></td><td>Offset removed from time-domain output data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
</table></td></tr>
</table>

For more information on using `Report`, see “Estimation Report”.

More About

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.

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`

Introduced before R2006a

iv4

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(___,Name,Value)
sys = iv4(___,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:

$$A(q)y(t) = B(q)u(t-nk) + v(t)$$

`sys = iv4(data,'na',na,'nb',nb,'nk',nk)` alternatively specify the ARX model orders separately.

`sys = iv4(___,Name,Value)` estimates an ARX polynomial with additional options specified by one or more `Name,Value` pair arguments.

`sys = iv4(___,opt)` uses the option set, `opt`, to configure the estimation behavior.

Input Arguments

`data`

Estimation data. The data can be:

- Time- or frequency-domain input-output data
- Time-series data
- Frequency-response data

`data` must be an `iddata`, `idfrd`, or `frd` object.

`data` must be discrete-time (Ts>0) for frequency domain.

`[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 *Ny*-by-*Ny* matrix, where each entry is the polynomial order relating the corresponding output pair. Here, *Ny* is the number of outputs. Specify `nb` and `nk` as *Ny*-by-*Nu* matrices, where *Nu* 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. `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 delay for each input channel, specified as a scalar value or numeric vector. 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 sample time `Ts`. For example, `InputDelay = 3` means a delay of three sample times.

For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector. Each entry of this vector is a numerical value that represents 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

`'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 in integer multiples of the sample time, `Ts`.

For a MIMO system with `Ny` outputs and `Nu` inputs, set `IODelay` to a `Ny`-by-`Nu` array. Each entry of this array is a numerical value that represents 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'`

Specify integrators in the noise channels.

Adding an integrator creates an ARIX model represented by:

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

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

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

Default: `false(Ny,1)`, where `Ny` is the number of outputs

Output Arguments

`sys`

ARX model that fits the estimation data, returned as a discrete-time `idpoly` object. This model is created using the specified model orders, delays, and estimation options.

Information about the estimation results and options used is stored in the `Report` property of the model. `Report` has the following fields:

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Status</code></td><td>Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.</td></tr>
<tr><td><code>Method</code></td><td>Estimation command used.</td></tr>
<tr><td><code>InitialC</code></td><td>Handling of initial conditions during model estimation, returned as one of the following values:
<code>'zero'</code> — The initial conditions were set to zero.
<code>'estimate'</code> — The initial conditions were treated as independent estimation parameters.
This field is especially useful to view how the initial conditions were handled when the <code>InitialCondition</code> option in the estimation option set is <code>'auto'</code>.</td></tr>
<tr><td><code>Fit</code></td><td>Quantitative assessment of the estimation, returned as a structure. See “Loss Function and Model Quality Metrics” for more information on these quality metrics. The structure has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>FitPerc</code></td><td>Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as a percentage.</td></tr>
<tr><td><code>LossFcn</code></td><td>Value of the loss function when the estimation completes.</td></tr>
<tr><td><code>MSE</code></td><td>Mean squared error (MSE) measure of how well the response of the model fits the estimation data.</td></tr>
<tr><td><code>FPE</code></td><td>Final prediction error for the model.</td></tr>
<tr><td><code>AIC</code></td><td>Raw Akaike Information Criteria (AIC) measure of model quality.</td></tr>
</table></td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td></td><td><table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>AICc</code></td><td>Small sample-size corrected AIC.</td></tr>
<tr><td><code>nAIC</code></td><td>Normalized AIC.</td></tr>
<tr><td><code>BIC</code></td><td>Bayesian Information Criteria (BIC).</td></tr>
</table></td></tr>
<tr><td><code>Parameter</code></td><td>Estimated values of model parameters.</td></tr>
<tr><td><code>OptionsUs</code></td><td>Option set used for estimation. If no custom options were configured, this is a set of default options. See <code>iv4Options</code> for more information.</td></tr>
<tr><td><code>RandState</code></td><td>State of the random number stream at the start of estimation. Empty, <code>[]</code>, if randomization was not used during estimation. For more information, see <code>rng</code> in the MATLAB documentation.</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation, returned as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
<tr><td><code>Type</code></td><td>Data type.</td></tr>
<tr><td><code>Length</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time.</td></tr>
<tr><td><code>InterSam</code></td><td>Input intersample behavior, returned as one of the following values:
• <code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
• <code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
• <code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.</td></tr>
<tr><td><code>InputOff</code></td><td>Offset removed from time-domain input data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
<tr><td><code>OutputOf</code></td><td>Offset removed from time-domain output data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
</table>
</td></tr>
</table>

For more information on using `Report`, see “Estimation Report”.

Examples

Estimate ARX Model Using Four-Stage Instrumental Variable Method

Load estimation data.

```
load iddata7;
```

This data has two inputs, `u1` and `u2`, and one output, `y1`.

Specify the ARX model orders, using the same orders for both inputs.

```
na = 2;
nb = [2 2];
```

Specify a delay of `2` samples for input `u2` and no delay for input `u1`.

```
nk = [0 2];
```

Estimate an ARX model using the four-stage instrumental variable method.

```
m = iv4(z7,[na nb nk]);
```

More About

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

`armax` | `arx` | `bj` | `idpoly` | `iv4Options` | `ivx` | `n4sid` | `oe` | `polyest`

Introduced before R2006a

iv4Options

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. `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'` — Handling of initial conditions
`'auto'` (default) | `'zero'` | `'estimate'`

Handling of initial conditions during estimation, specified as one of the following values:

- `'zero'` — The initial condition is set to zero.
- `'estimate'` — The initial condition is treated as an independent estimation parameter.
- `'auto'` — The software chooses the initial condition handling method based on the estimation data.

`'Focus'` — Error to be minimized

`'prediction'` (default) | `'simulation'`

Error to be minimized in the loss function during estimation, specified as the comma-separated pair consisting of `'Focus'` and one of the following values:

- `'prediction'` — The one-step ahead prediction error between measured and predicted outputs is minimized during estimation. As a result, the estimation focuses on producing a good predictor model.
- `'simulation'` — The simulation error between measured and simulated outputs is minimized during estimation. As a result, the estimation focuses on making a good fit for simulation of model response with the current inputs.

The `Focus` option can be interpreted as a weighting filter in the loss function. For more information, see “Loss Function and Model Quality Metrics”.

`'WeightingFilter'` — Weighting prefilter

`[]` (default) | vector | matrix | cell array | linear system

Weighting prefilter applied to the loss function to be minimized during estimation. To understand the effect of `WeightingFilter` on the loss function, see “Loss Function and Model Quality Metrics”.

Specify `WeightingFilter` as one of the following values:

- `[]` — No weighting prefilter is used.
- Passbands — Specify a row vector or matrix containing frequency values that define desired passbands. You select a frequency band where the fit between estimated model and estimation data is optimized. For example, `[wl,wh]` where `wl` and `wh` represent lower and upper limits of a passband. For a matrix with several rows defining frequency passbands, `[w1l,w1h;w2l,w2h;w3l,w3h;...]`, the estimation algorithm uses the union of the frequency ranges to define the estimation passband.

  Passbands are expressed in `rad/TimeUnit` for time-domain data and in `FrequencyUnit` for frequency-domain data, where `TimeUnit` and `FrequencyUnit` are the time and frequency units of the estimation data.
- SISO filter — Specify a single-input-single-output (SISO) linear filter in one of the following ways:
  - A SISO LTI model

  - `{A,B,C,D}` format, which specifies the state-space matrices of a filter with the same sample time as estimation data.
  - `{numerator,denominator}` format, which specifies the numerator and denominator of the filter as a transfer function with same sample time as estimation data.

    This option calculates the weighting function as a product of the filter and the input spectrum to estimate the transfer function.
- Weighting vector — Applicable for frequency-domain data only. Specify a column vector of weights. This vector must have the same length as 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.

`'EnforceStability'` — Control whether to enforce stability of model

`false` (default) | `true`

Control whether to enforce stability of estimated model, specified as the comma-separated pair consisting of `'EnforceStability'` and either `true` or `false`.

Data Types: `logical`

`'EstCovar'` — Control whether to generate parameter covariance data

`true` (default) | `false`

Controls whether parameter covariance data is generated, specified as `true` or `false`.

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

`'Display'` — Specify whether to display the estimation progress

`'off'` (default) | `'on'`

Specify whether to display the estimation progress, specified as one of the following values:

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

`'InputOffset'` — Removal of offset from time-domain input data during estimation

`[ ]` (default) | vector of positive integers | matrix

Removal of offset from time-domain input data during estimation, specified as the comma-separated pair consisting of `'InputOffset'` and one of the following:

- A column vector of positive integers of length *Nu*, where *Nu* is the number of inputs.
- `[]` — Indicates no offset.
- *Nu*-by-*Ne* matrix — For multi-experiment data, specify `InputOffset` as an *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.

`'OutputOffset'` — Removal of offset from time-domain output data during estimation

[ ] (default) | vector | matrix

Removal of offset from time-domain output data during estimation, specified as the comma-separated pair consisting of `'OutputOffset'` and one of the following:

- A column vector of length *Ny*, where *Ny* is the number of outputs.
- `[]` — Indicates no offset.
- *Ny*-by-*Ne* matrix — For multi-experiment 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.

`'Advanced'` — Additional advanced options

structure

Additional advanced options, specified as 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(eps)`

Output Arguments

`opt` — Options set for `iv4`

`iv4Options` option set

Option set for `iv4`, returned as an `iv4Options` option set.

Examples

Create Default Options Set for ARX Model Estimation Using 4-Stage Instrument Variable Method

```
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. Set `Display` to `'on'`.

```
opt = iv4Options('InitialCondition','backcast','Display','on');
```

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

```
opt = iv4Options;
opt.InitialCondition = 'backcast';
opt.Display = 'on';
```

More About

- “Loss Function and Model Quality Metrics”

See Also

`iv4`

Introduced in R2012a

linapp

Linear approximation of nonlinear ARX and Hammerstein-Wiener models for given input

Syntax

```
lm = linapp(nlmodel,u)
lm = linapp(nlmodel,umin,umax,nsample)
```

Description

`lm = linapp(nlmodel,u)` computes a linear approximation of a nonlinear ARX or Hammerstein-Wiener model by simulating the model output for the input signal `u`, and estimating a linear model `lm` from `u` and the simulated output signal. `lm` is an `idpoly` model.

`lm = linapp(nlmodel,umin,umax,nsample)` 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 `umin` and `umax` and (optionally) the number of samples.

Input Arguments

`nlmodel`

Name of the `idnlarx` or `idnlhw` model object you want to linearize.

`u`

Input signal as an `iddata` object or a real matrix.

Dimensions of `u` must match the number of inputs in `nlmodel`.

`[umin,umax]`

Minimum and maximum input values for generating white-noise input with a magnitude in this rectangular range. The sample length of this signal is `nsample`.

`nsample`

Optional argument when you specify `[umin,umax]`. Specifies the length of the white-noise input.

Default: `1024`.

More About

- "Linear Approximation of Nonlinear Black-Box Models"

See Also

`idnlarx` | `idnlhw` | `idnlarx/findop` | `idnlhw/findop` | `idnlhw/linearize` | `idnlarx/linearize`

Introduced in R2007a

linear

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 1-752.

linear Properties

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

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

```
% List Parameters values
get(lin)
% Get value of Parameters property
lin.Parameters
```

Property Name	Description
`Parameters`	Structure containing the following fields: • `LinearCoef`: `m`-*by*-1 vector *L*. • `OutputOffset`: Scalar `d`.

Examples

Estimate a nonlinear ARX model using the `linear` estimator with custom regressors for the following system:

$y(t) = a1y(t–1) + a2y(t–2) + a3u(t–1) + a4y(t–1)u(t–2) + a5|u(t)|u(t–3) + a6,$

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.

Tutorials

“Estimate Nonlinear ARX Models at the Command Line”

More About

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,[]);
```

Algorithms

When the `Focus` option in `nlarxOptions` 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='Simulation'`, `linear` uses an iterative technique for estimating parameters.

See Also

`customreg` | `nlarx`

Introduced in R2007a

idnlarx/linearize

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 1-528.

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` command.

Output Arguments

- `SYS` is an `idss` model.

  When the Control System Toolbox product is installed, `SYS` is an LTI object.

Examples

Linearize Nonlinear ARX Model at Simulation Snapshot

Linearize a nonlinear ARX model around an operating point corresponding to a simulation snapshot at a specific time.

Load sample data.

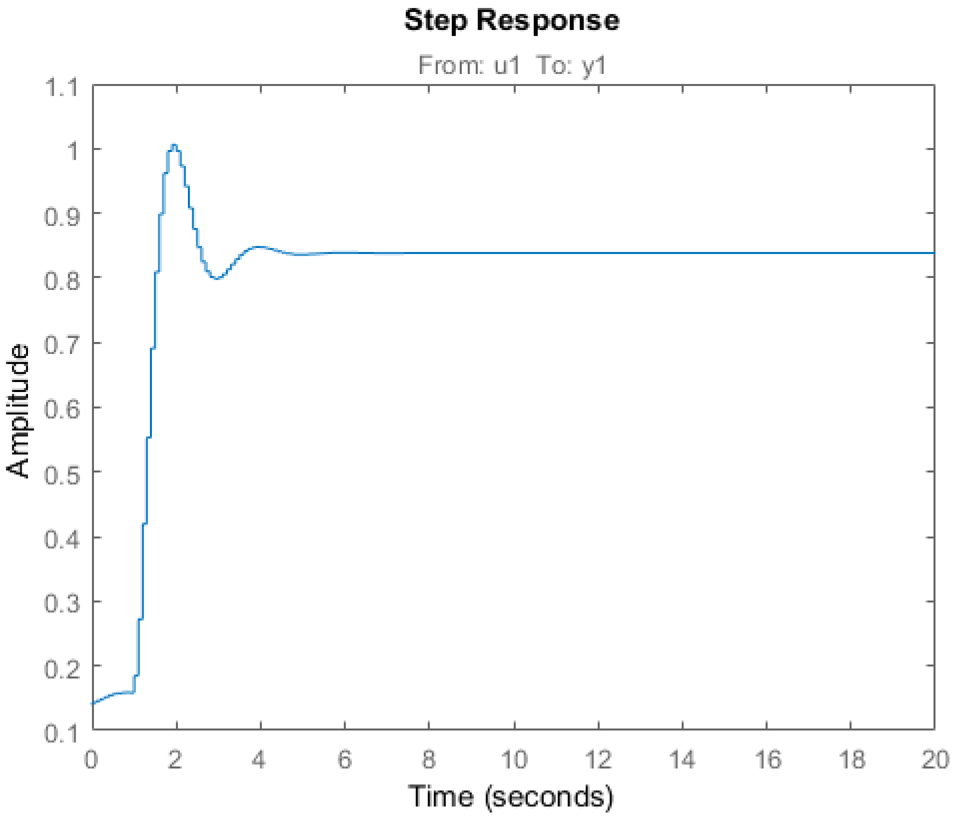
```
load iddata2
```

Estimate nonlinear ARX 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]);
```

Plot the response of the model for a step input.

```
step(nlsys, 20)
```



The step response is a steady-state value of `0.8383` at `T = 20` seconds.

Compute the operating point corresponding to `T = 20`.

```
stepinput = iddata([],[zeros(10,1);ones(200,1)],nlsys.Ts);
[x,u] = findop(nlsys,'snapshot',20,stepinput);
```

Linearize the model about the operating point corresponding to the model snapshot at `T = 20`.

```
sys = linearize(nlsys,u,x);
```

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 perturbation step signal with amplitude 0.1.

```
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` computed previously as initial conditions.

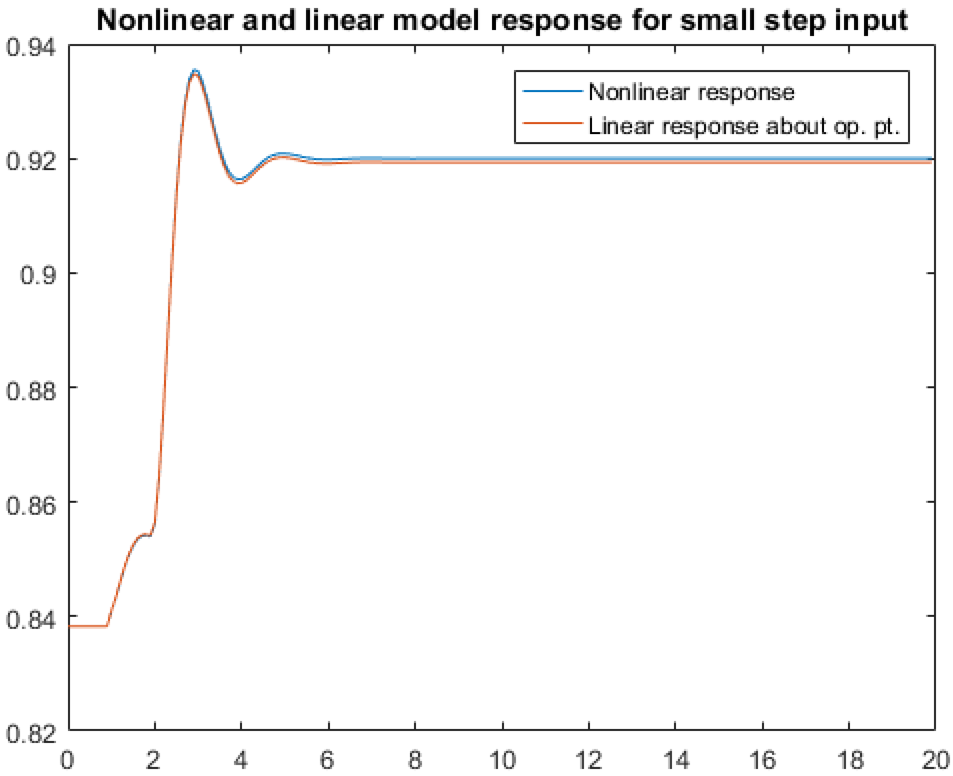
```
u_nl = 1 + delta_u;
y_nl = sim(nlsys,u_nl,x);
```

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'])
```



More About

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

$y^* = f(X^*, u^*)$

where X^* and u^* are the state vector and input at the operating point.

The linear approximation of the model response is as follows:

$$\begin{aligned}\Delta X(t+1) &= (A + B_1 f_X)\Delta X(t) + (B_1 f_u + B_2)\Delta u(t)\\ \Delta y(t) &= f_X \Delta X(t) + f_u \Delta u(t)\end{aligned}$$

where

- $\Delta X(t) = X(t) - X^*(t)$
- $\Delta u(t) = u(t) - u^*(t)$
- $\Delta y(t) = y(t) - y^*(t)$
- $B\tilde{U} = [B_1, B_2]\begin{bmatrix} Y \\ U \end{bmatrix} = B_1 Y + B_2 U$
- $f_X = \dfrac{\partial}{\partial X} f(X,U)\Big|_{X^*,U^*}$
- $f_U = \dfrac{\partial}{\partial U} f(X,U)\Big|_{X^*,U^*}$

Note: For linear approximations over larger input ranges, use `linapp`.

- “Linear Approximation of Nonlinear Black-Box Models”

See Also

`idnlarx/findop` | `idnlarx` | `linapp` | `getDelayInfo`

Introduced in R2014b

idnlhw/linearize

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 "Definition of `idnlhw` States" on page 1-567.

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` command.

Output Arguments

- `SYS` is an `idss` model.

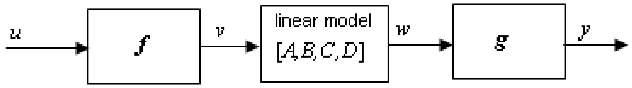
  When the Control System Toolbox product is installed, `SYS` is an LTI object.

Examples

More About

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:

$$\begin{aligned}\Delta X(t+1) &= A\Delta X(t) + Bf_u\Delta u(t)\\ \Delta y(t) &\approx g_w C\Delta X(t) + g_w Df_u\Delta u(t)\end{aligned}$$

where

- $\Delta X(t) = X(t) - X^*(t)$
- $\Delta u(t) = u(t) - u^*(t)$
- $\Delta y(t) = y(t) - y^*(t)$
- $f_u = \frac{\partial}{\partial u} f(u)\Big|_{u=u^*}$
- $g_w = \frac{\partial}{\partial w} g(w)\Big|_{w=w^*}$

where y^* is the output of the model corresponding to input u^* and state vector X^*, $v^* = f(u^*)$, and w^* is the response of the linear model for input v^* and state X^*.

Note: For linear approximations over larger input ranges, use `linapp`. For more information, see the `linapp` reference page.

- “Linear Approximation of Nonlinear Black-Box Models”

See Also

`idnlhw/findop` | `idnlhw` | `linapp`

Introduced in R2014b

lsim

Simulate time response of dynamic system to arbitrary inputs

Syntax

```
lsim(sys,u,t)
lsim(sys,u,t,x0)
lsim(sys,u,t,x0,method)
lsim(sys1,...,sysn,u,t)
lsim(sys1,PlotStyle1,...,sysN,PlotStyleN,u,t)
y = lsim( ___ )
[y,t,x] = lsim( ___ )
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 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 input `u` is an array having as many rows as time samples (`length(t)`) and as many columns as system inputs. For instance, if `sys` is a SISO system, then `u` is a t-by-1 vector. If `sys` has three inputs, then `u` is a t-by-3 array. Each row `u(i,:)` specifies the input value(s) at the time sample `t(i)`. The signal `u` also appears on the plot.

The model `sys` can be continuous or discrete, SISO or MIMO. In discrete time, `u` must be sampled at the same rate as the system. In this case, the input `t` is redundant and can be omitted or set to an 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 1-768 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 when `sys` is a state-space model. `x0` is a vector whose entries are the initial values of the corresponding states of `sys`.

`lsim(sys,u,t,x0,method)` explicitly specifies how the input values should be interpolated between samples, when `sys` is a continuous-time system. Specify `method` as one of the following values:

- `'zoh'` — Use zero-order hold
- `'foh'` — Use linear interpolation (first-order hold)

If you do not specify a method, `lsim` selects the interpolation method automatically based on the smoothness of the signal `u`.

`lsim(sys1,...,sysn,u,t)` simulates the responses of several dynamic system models to the same input history `t`,`u` and plots these responses on a single figure. You can also use the `x0` and `method` input arguments when computing the responses of multiple models.

`lsim(sys1,PlotStyle1,...,sysN,PlotStyleN,u,t)` specifies the line style, marker, and color of each of the system responses in the plot. You can also use the `x0` and `method` input arguments with this syntax. Each `PlotStyle` argument is specified as a vector of one, two, or three characters. The characters can appear in any order. For example, the following code plots the response of `sys1` as a yellow dotted line and the response of `sys2` as a green dashed line:

```
lsim(sys1,'y:',sys2,'g--',u,t,x0)
```

For more information about configuring the `PlotStyle` argument, see “Specify Line Style, Color, and Markers” in the MATLAB documentation.

`y = lsim(___)` returns the system response `y`, sampled at the same times as the input (`t`). The output `y` is an array having as many rows as time samples (`length(t)`) and as many columns as system outputs. No plot is drawn on the screen. You can use this syntax with any of the input arguments described in previous syntaxes except the `PlotStyle` arguments.

`[y,t,x] = lsim(___)` also returns the time vector `t` used for simulation and the state trajectories `x` (for state-space models only). The output `x` has as many rows as time samples (`length(t)`) and as many columns as system states. You can use this syntax with any of the input arguments described in previous syntaxes except the `PlotStyle` arguments.

`lsim(sys)` opens the Linear Simulation Tool GUI. For more information about working with this GUI, see Working with the Linear Simulation Tool.

Examples

Simulate Response to Square Wave

Simulate and plot the response of the following system to a square wave with period of four seconds:

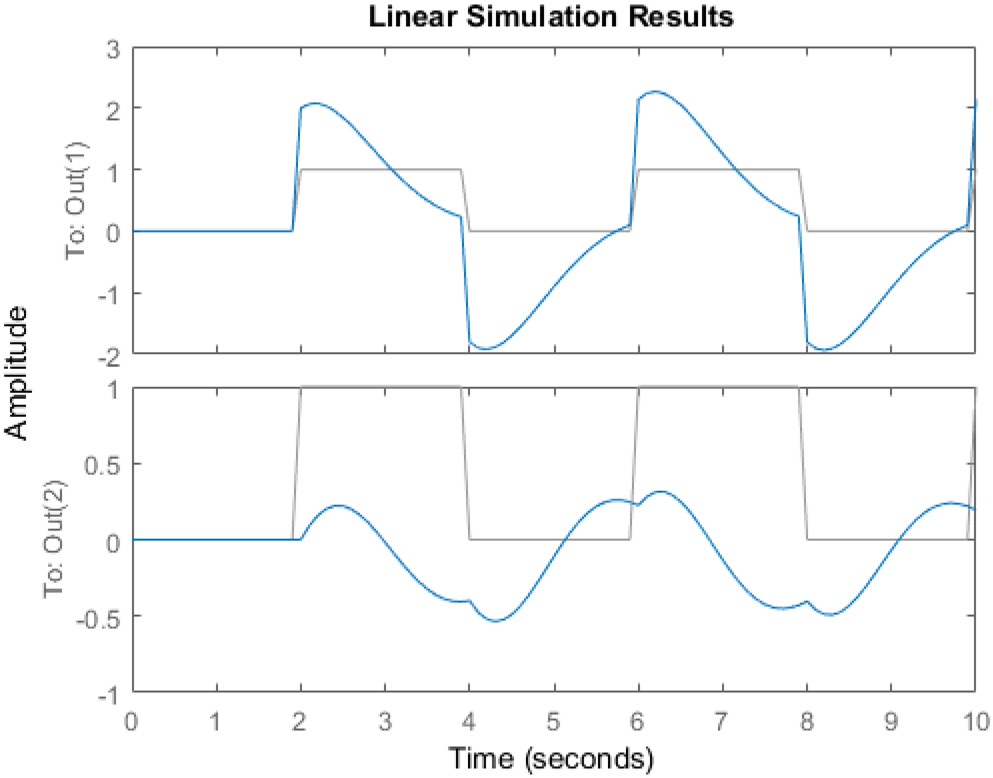
$$H(s) = \begin{bmatrix} \dfrac{2s^2+5s+1}{s^2+2s+3} \\ \dfrac{s-1}{s^2+s+5} \end{bmatrix}.$$

Create the transfer function, and generate the square wave with `gensig`. Sample every 0.1 second during 10 seconds.

```
H = [tf([2 5 1],[1 2 3]);tf([1 -1],[1 1 5])];
[u,t] = gensig('square',4,10,0.1);
```

Then simulate with `lsim`.

```
lsim(H,u,t)
```



The plot displays both the applied signal and the response.

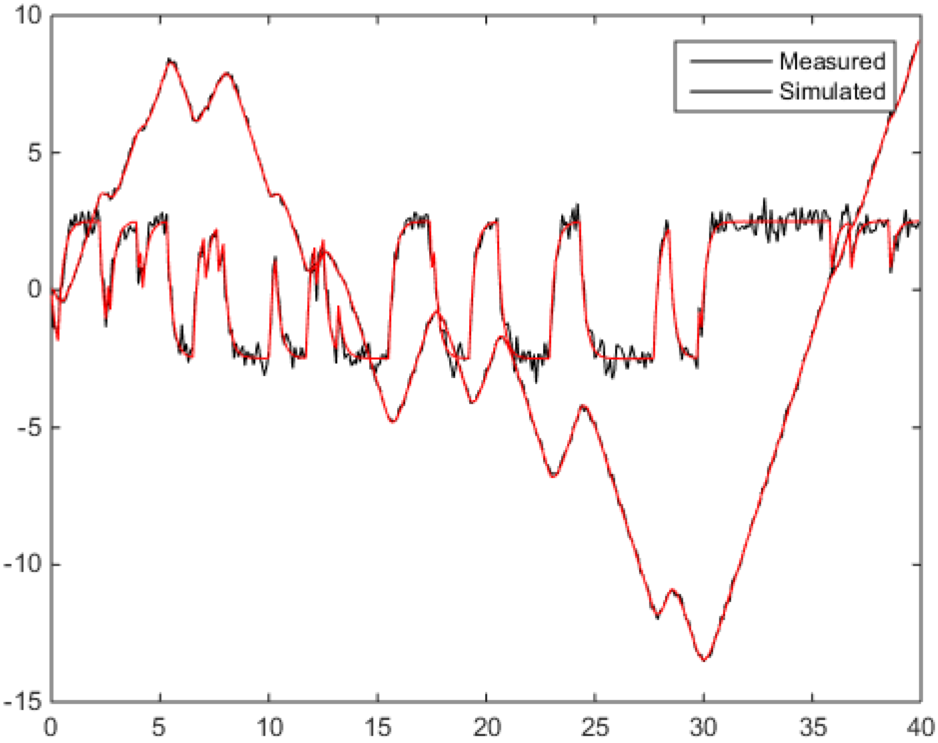
Simulate Response of Identified Model

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','dcmotordata'));
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')
```



More About

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 sample time is set to the spacing `dt` between the user-supplied time samples `t`.

The choice of sample time can drastically affect simulation results. To illustrate why, consider the second-order model

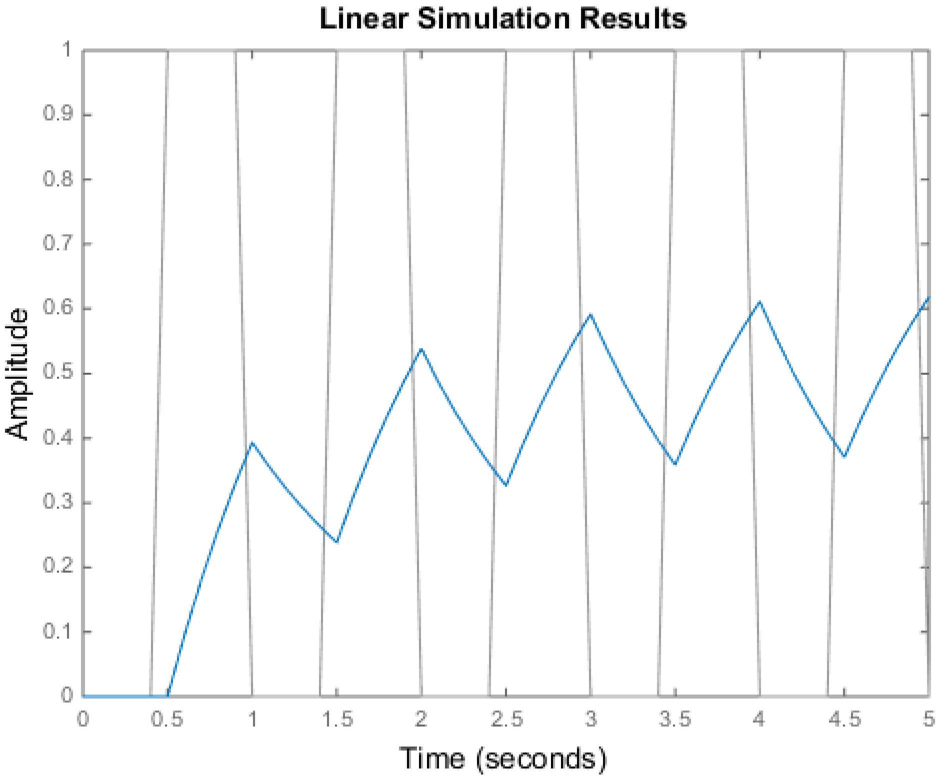
$$H(s) = \frac{\omega^2}{s^2 + 2s + \omega^2}, \quad \omega = 62.83$$

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)
```

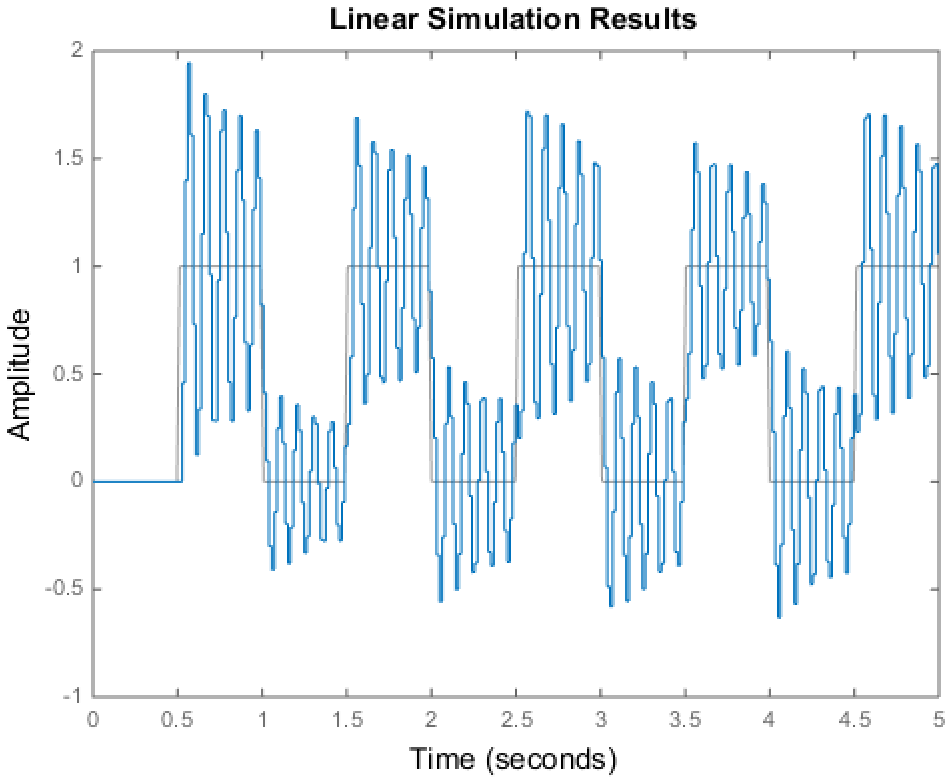
`lsim` evaluates the specified sample time, and issues a warning:

```
Warning: Input signal is undersampled. Sample every 0.016 sec or
faster.
```



To improve on this response, discretize $H(s)$ using the recommended sample time:

```
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 that is hidden in the undersampled version.

See Also

`gensig` | `impulse` | `initial` | Linear System Analyzer | `lsiminfo` | `sim` | `step`

Introduced in R2012a

lsiminfo

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) - yfinal|` 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`

Introduced in R2012a

lsimplot

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`

Introduced in R2012a

mag2db

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`

Introduced in R2008a

merge (iddata)

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 `NaN`s to conform with these rules.
- If the `ExperimentNames` of `datk` have been specified as something other than the default `'Exp1'`, `'Exp2'`, etc., they must all be unique. If default names overlap, they are modified so that `dat` will have a list of unique `ExperimentNames`.

The sampling intervals, the number of observations, and the input properties (`Period`, `InterSample`) might be different in the different experiments.

You can retrieve the individual experiments by using the command `getexp`. You can also retrieve them by subreferencing with a fourth index.

```
dat1 = dat(:,:,:,ExperimentNumber)
```

or

```
dat1 = dat(:,:,:,ExperimentName)
```

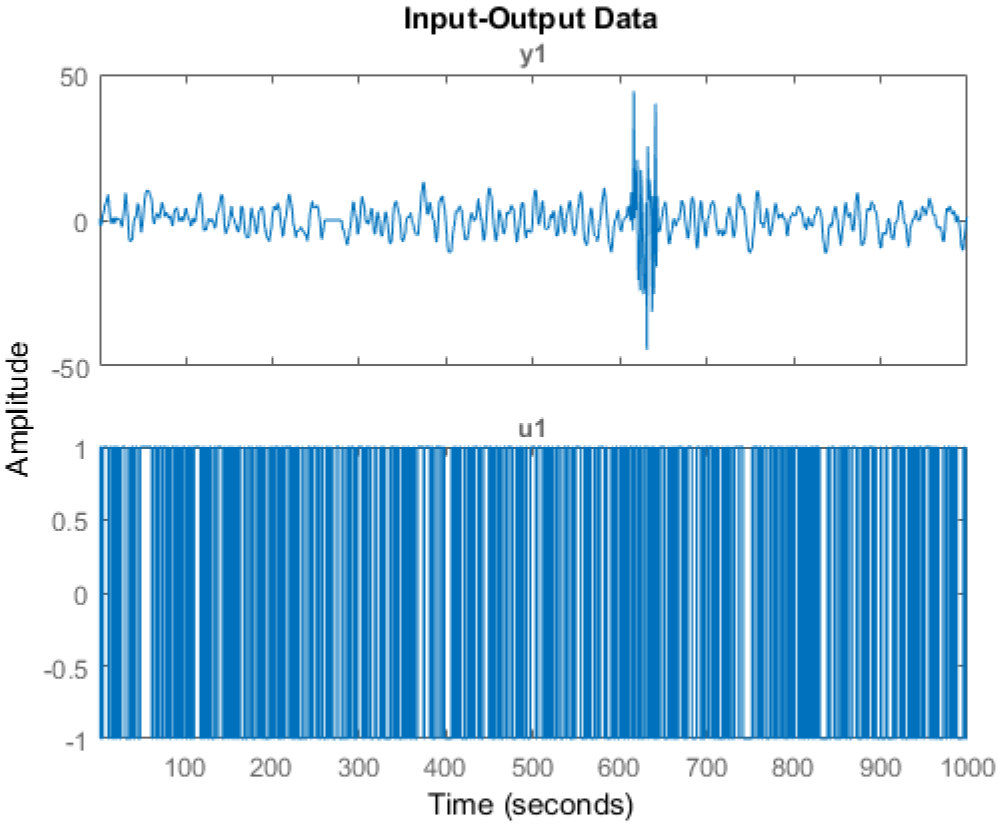
Storing multiple experiments as one `iddata` object can be very useful for handling experimental data that has been collected on different occasions, or when a data set has been split up to remove “bad” portions of the data. All the toolbox routines accept multiple-experiment data.

Examples

Merge Multiple Data Sets

Remove bad portions of data to estimate models without the bad data destroying the estimate.

```
load iddemo8;
plot(dat);
```



Bad portions of data are detected around sample 250 to 280 and between samples 600 to 650. Cut out these bad portions to form a multiple-experiment data set and merge the data.

```
dat = merge(dat(1:250),dat(281:600),dat(651:1000));
```

You can use the first two experiments to estimate a model and the third experiment to validate the model.

```
dat_est = getexp(dat,[1,2]);
m = ssest(dat_est,2);
dat_val = getexp(dat,3);
```

- “Dealing with Multi-Experiment Data and Merging Models”
- “Create Multiexperiment Data at the Command Line”

See Also

`getexp` | `iddata` | `merge`

Introduced before R2006a

merge

Merge estimated models

Syntax

```
m = merge(m1,m2,....,mN)
[m,tv] = merge(m1,m2)
```

Description

`m = merge(m1,m2,....,mN)` merges estimated models. The models `m1,m2,...,mN` must all be of the same structure, just differing in parameter values and covariance matrices. Then `m` is the merged model, where the parameter vector is a statistically weighted mean (using the covariance matrices to determine the weights) of the parameters of `mk`.

`[m,tv] = merge(m1,m2)` returns a test variable `tv`. When two models are merged,

```
[m, tv] = merge(m1,m2)
```

`tv` is χ^2 distributed with `n` degrees of freedom, if the parameters of `m1` and `m2` have the same means. Here `n` is the length of the parameter vector. A large value of `tv` thus indicates that it might be questionable to merge the models.

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]);
```

result in models `ma` and `mb` that are related and should be close. The difference is that merging the data sets assumes that the signal-to-noise ratios are about the same in the two experiments. Merging the models allows one model to be much more uncertain, for example, due to more disturbances in that experiment. If the conditions are about the same, we recommend that you merge data rather than models, since this is more efficient and typically involves better conditioned calculations.

See Also

`append`

Introduced in R2007a

midprefs

Specify location for file containing System Identification app startup information

Syntax

```
midprefs
midprefs(path)
```

Description

The System Identification app 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 the **System Identification** app. 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

Apps
System Identification

Introduced before R2006a

misdata

Reconstruct missing input and output data

Syntax

```
Datae = misdata(Data)
Datae = misdata(Data,Model)
Datae = misdata(Data,Maxiter,Tol)
```

Description

`Datae = misdata(Data)` reconstructs missing input and output data. `Data` is time-domain input-output data in the `iddata` object format. Missing data samples (both in inputs and in outputs) are entered as `NaN`s. `Datae` is an `iddata` object where the missing data has been replaced by reasonable estimates.

`Datae = misdata(Data,Model)` specifies a model used for the reconstruction of missing data. `Model` is any linear identified model (`idtf`, `idproc`, `idgrey`, `idpoly`, `idss`). If no suitable model is known, it is estimated in an iterative fashion using default order state-space models.

`Datae = misdata(Data,Maxiter,Tol)` specifies maximum number of iterations and tolerance. `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`.

More About

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

`advice` | `arx` | `pexcit` | `tfest`

Introduced before R2006a

n4sid

Estimate state-space model using subspace method

Syntax

```
sys = n4sid(data,nx)
sys = n4sid(data,nx,Name,Value)
sys = n4sid(___,opt)
[sys,x0] = n4sid(___)
```

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:

$$\begin{aligned}\dot{x}(t) &= Ax(t) + Bu(t) + Ke(t)\\ y(t) &= Cx(t) + Du(t) + e(t)\end{aligned}$$

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 free estimation parameters by default. D is fixed to zero by default, 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. Use the `Form`, `Feedthrough`, and `DisturbanceModel` name-value pair arguments to modify the default behavior of the A, B, C, D, and K matrices.

`sys = n4sid(___,opt)` specifies estimation options, `opt`, that configure the initial states, estimation objective, and subspace algorithm related choices to be used for estimation.

`[sys,x0] = n4sid(___)` 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 multiexperiment data, the sample times and intersample 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,..,10`.

`opt`

Estimation options.

`opt` is an options set, created using `n4sidOptions`, which specifies options including:

- Estimation objective
- Handling of initial conditions
- Subspace algorithm related choices

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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'` — Sample time

sample time of data (`data.Ts`) (default) | positive scalar | 0

Sample time, specified as a positive scalar. 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 sample time.

`'Form'` — Type of canonical form

`'free'` (default) | `'modal'` | `'companion'` | `'canonical'`

Type of canonical form of `sys`, specified as 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 estimated.
- `'canonical'` — Obtain `sys` in observability canonical form [1].

Use the `Form`, `Feedthrough`, and `DisturbanceModel` name-value pair arguments to modify the default behavior of the A, B, C, D, and K matrices.

`'Feedthrough'` — Direct feedthrough from input to output

`false(1,Nu)` (default) | vector | scalar

Logical specifying direct feedthrough from input to output, specified as a logical vector of length Nu, where Nu is the number of inputs.

If `Feedthrough` is specified as a logical scalar, this value is applied to all the inputs. If the model has no states, then `Feedthrough` is `true(1,Nu)`.

`'DisturbanceModel'` — Specify whether to estimate the *K* matrix

`'estimate'` (For time-domain data) (default) | `'none'`

Specify estimation of the noise component (*K* matrix), specified as 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.

`'InputDelay'` — Input delays

0 (default) | scalar | vector

Input delay for each input channel, specified as a numeric vector. 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 sample time `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. Each entry of this vector is a numerical value that represents 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.

Output Arguments

`sys`

Identified state-space model, returned as a `idss` model. This model is created using the specified model orders, delays, and estimation options.

Information about the estimation results and options used is stored in the `Report` property of the model. `Report` has the following fields:

Report Field	Description
`Status`	Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.
`Method`	Estimation command used.

Report Field	Description
`InitialSt`	How initial states were handled during estimation, returned as one of the following values: • `'zero'` — The initial state is set to zero. • `'estimate'` — The initial state is treated as an independent estimation parameter. This field is especially useful when the `InitialState` option in the estimation option set is `'auto'`.
`N4Weight`	Weighting scheme used for singular-value decomposition by the N4SID algorithm, returned as one of the following values: • `'MOESP'` — Uses the MOESP algorithm. • `'CVA'` — Uses the Canonical Variable Algorithm. • `'SSARX'` — A subspace identification method that uses an ARX estimation based algorithm to compute the weighting. This option is especially useful when the `N4Weight` option in the estimation option set is `'auto'`.
`N4Horizor`	Forward and backward prediction horizons used by the N4SID algorithm, returned as a row vector with three elements — `[r sy su]`, where `r` is the maximum forward prediction horizon. `sy` is the number of past outputs, and `su` is the number of past inputs that are used for the predictions.

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Fit</code></td><td>Quantitative assessment of the estimation, returned as a structure. See “Loss Function and Model Quality Metrics” for more information on these quality metrics. The structure has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>FitPerc</code></td><td>Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as a percentage.</td></tr>
<tr><td><code>LossFcn</code></td><td>Value of the loss function when the estimation completes.</td></tr>
<tr><td><code>MSE</code></td><td>Mean squared error (MSE) measure of how well the response of the model fits the estimation data.</td></tr>
<tr><td><code>FPE</code></td><td>Final prediction error for the model.</td></tr>
<tr><td><code>AIC</code></td><td>Raw Akaike Information Criteria (AIC) measure of model quality.</td></tr>
<tr><td><code>AICc</code></td><td>Small sample-size corrected AIC.</td></tr>
<tr><td><code>nAIC</code></td><td>Normalized AIC.</td></tr>
<tr><td><code>BIC</code></td><td>Bayesian Information Criteria (BIC).</td></tr>
</table></td></tr>
<tr><td><code>Parameter</code></td><td>Estimated values of model parameters.</td></tr>
<tr><td><code>OptionsUs</code></td><td>Option set used for estimation. If no custom options were configured, this is a set of default options. See <code>n4sidOptions</code> for more information.</td></tr>
<tr><td><code>RandState</code></td><td>State of the random number stream at the start of estimation. Empty, <code>[]</code>, if randomization was not used during estimation. For more information, see <code>rng</code> in the MATLAB documentation.</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation, returned as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
<tr><td><code>Type</code></td><td>Data type.</td></tr>
<tr><td><code>Length</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time.</td></tr>
<tr><td><code>InterSam</code></td><td>Input intersample behavior, returned as one of the following values:
• <code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
• <code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
• <code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.</td></tr>
<tr><td><code>InputOff</code></td><td>Offset removed from time-domain input data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
<tr><td><code>OutputOf</code></td><td>Offset removed from time-domain output data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
</table>
</td></tr>
</table>

For more information on using `Report`, see “Estimation Report”.

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.

Examples

Estimate State-Space Model and Specify Estimation Options

Load estimation data.

```
load iddata2 z2
```

Specify the estimation options.

```
opt = n4sidOptions('Focus','simulation','Display','on');
```

Estimate the model.

```
nx = 3;
sys = n4sid(z2,nx,opt);
```

`sys` is a third-order, state-space model.

Estimate State-Space Model from Closed-Loop Data

Estimate a state-space model from closed-loop data using the subspace algorithm `SSARX`. This algorithm is better at capturing feedback effects than other weighting algorithms.

Generate closed-loop estimation data for a second-order system corrupted by white noise.

```
N = 1000;
K = 0.5;
rng('default');
w = randn(N,1);
z = zeros(N,1);
u = zeros(N,1);
y = zeros(N,1);
e = randn(N,1);
v = filter([1 0.5],[1 1.5 0.7],e);
for k = 3:N
   u(k-1) = -K*y(k-2) + w(k);
   u(k-1) = -K*y(k-1) + w(k);
   z(k) = 1.5*z(k-1) - 0.7*z(k-2) + u(k-1) + 0.5*u(k-2);
   y(k) = z(k) + 0.8*v(k);
end
dat = iddata(y, u, 1);
```

Specify the weighting scheme used by the N4SID algorithm. In one options set, specify the algorithm as CVA and in the other, specify as SSARX.

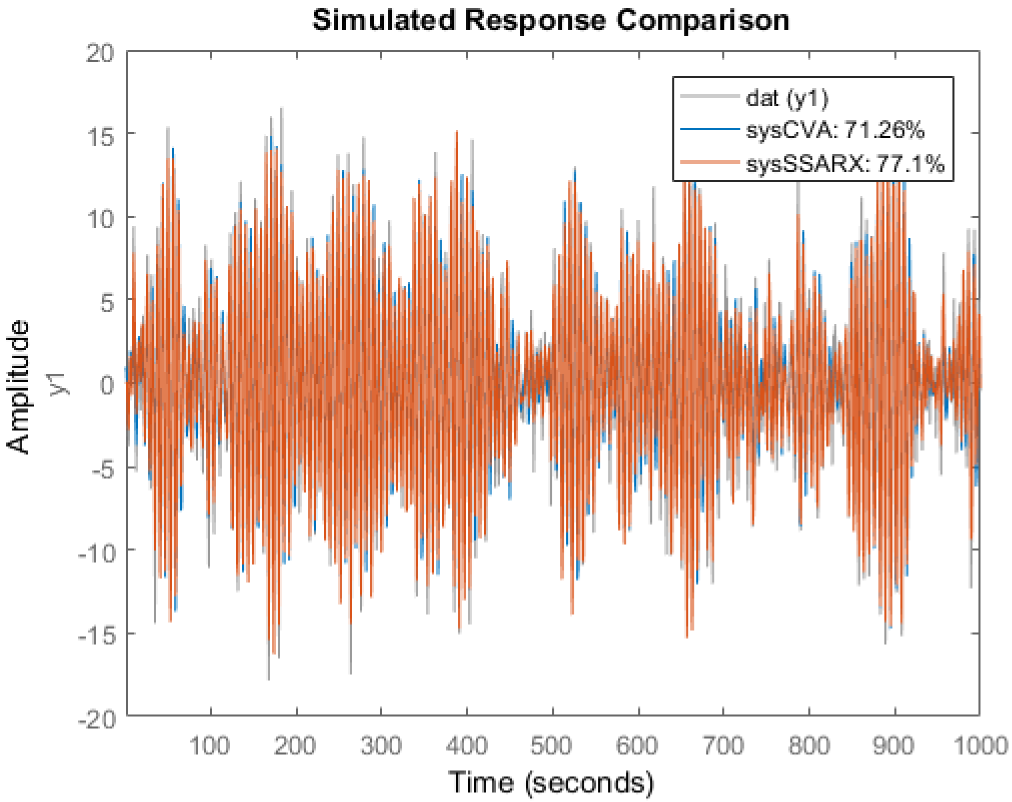
```
optCVA = n4sidOptions('N4weight','CVA');
optSSARX = n4sidOptions('N4weight','SSARX');
```

Estimate state-space models using the options sets.

```
sysCVA = n4sid(dat,2,optCVA);
sysSSARX = n4sid(dat,2,optSSARX);
```

Compare the fit of the two models with the estimation data.

```
compare(dat,sysCVA,sysSSARX);
```



From the plot, you see that the model estimated using the SSARX algorithm produces a better fit than the CVA algorithm.

Estimate a Canonical-Form, Continuous-Time Model

Estimate a continuous-time, canonical-form model.

Load estimation data.

```
load iddata1 z1
```

Specify the estimation options.

```
opt = n4sidOptions('Focus','simulation','Display','on');
```

Estimate the model.

```
nx = 2;
sys = n4sid(z1,nx,'Ts',0,'Form','canonical',opt);
```

`sys` is a second-order, continuous-time, state-space model in the canonical form.

More About

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 right-most column of the A matrix. For a system with characteristic polynomial

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

the corresponding companion A matrix is

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

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.

References

[1] Ljung, L. *System Identification: Theory for the User*, Appendix 4A, Second Edition, pp. 132–134. Upper Saddle River, NJ: Prentice Hall PTR, 1999.

[2] van Overschee, P., and B. De Moor. *Subspace Identification of Linear Systems: Theory, Implementation, Applications.* Springer Publishing: 1996.

[3] Verhaegen, M. "Identification of the deterministic part of MIMO state space models." *Automatica*, 1994, Vol. 30, pp. 61—74.

[4] Larimore, W.E. "Canonical variate analysis in identification, filtering and adaptive control." *Proceedings of the 29th IEEE Conference on Decision and Control*, 1990, pp. 596–604.

See Also

`canon` | `iddata` | `idfrd` | `idgrey` | `idss` | `n4sidOptions` | `pem` | `polyest` | `procest` | `ssest` | `tfest`

Introduced before R2006a

n4sidOptions

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. `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'` — Handling of initial states

`'estimate'` (default) | `'zero'`

Handling of initial states during estimation, specified as one of the following values:

- `'zero'` — The initial state is set to zero.
- `'estimate'` — The initial state is treated as an independent estimation parameter.

`'N4Weight'` — Weighting scheme used for singular-value decomposition by the `N4SID` algorithm

`'auto'` (default) | `'MOESP'` | `'CVA'` | `'SSARX'`

Weighting scheme used for singular-value decomposition by the N4SID algorithm, specified as one of the following values:

- `'MOESP'` — Uses the MOESP algorithm by Verhaegen [2].
- `'CVA'` — Uses the Canonical Variable Algorithm by Larimore [1].

  Estimation using frequency-domain data always uses `'CVA'`.
- `'SSARX'` — A subspace identification method that uses an ARX estimation based algorithm to compute the weighting.

  Specifying this option allows unbiased estimates when using data that is collected in closed-loop operation. For more information about the algorithm, see [4].
- `'auto'` — The estimating function chooses between the `MOESP`, `CVA` and `SSARX` algorithms.

`'N4Horizon'` — Forward- and backward-prediction horizons used by the `N4SID` algorithm

`'auto'` (default) | vector `[r sy su]` | `k`-by-3 matrix

Forward- and backward-prediction horizons used by the N4SID algorithm, specified as 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 more information. 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`.

`'Focus'` — Error to be minimized

`'prediction'` (default) | `'simulation'`

Error to be minimized in the loss function during estimation, specified as the comma-separated pair consisting of `'Focus'` and one of the following values:

- `'prediction'` — The one-step ahead prediction error between measured and predicted outputs is minimized during estimation. As a result, the estimation focuses on producing a good predictor model.
- `'simulation'` — The simulation error between measured and simulated outputs is minimized during estimation. As a result, the estimation focuses on making a good fit for simulation of model response with the current inputs.

The `Focus` option can be interpreted as a weighting filter in the loss function. For more information, see “Loss Function and Model Quality Metrics”.

`'WeightingFilter'` — Weighting prefilter

[ ] (default) | vector | matrix | cell array | linear system

Weighting prefilter applied to the loss function to be minimized during estimation. To understand the effect of `WeightingFilter` on the loss function, see “Loss Function and Model Quality Metrics”.

Specify `WeightingFilter` as one of the following values:

- `[]` — No weighting prefilter is used.
- Passbands — Specify a row vector or matrix containing frequency values that define desired passbands. You select a frequency band where the fit between estimated model and estimation data is optimized. For example, `[wl,wh]` where `wl` and `wh` represent lower and upper limits of a passband. For a matrix with several rows defining frequency passbands, `[w1l,w1h;w2l,w2h;w3l,w3h;...]`, the estimation algorithm uses the union of the frequency ranges to define the estimation passband.

  Passbands are expressed in `rad/TimeUnit` for time-domain data and in `FrequencyUnit` for frequency-domain data, where `TimeUnit` and `FrequencyUnit` are the time and frequency units of the estimation data.
- SISO filter — Specify a single-input-single-output (SISO) linear filter in one of the following ways:
  - A SISO LTI model
  - `{A,B,C,D}` format, which specifies the state-space matrices of a filter with the same sample time as estimation data.
  - `{numerator,denominator}` format, which specifies the numerator and denominator of the filter as a transfer function with same sample time as estimation data.

This option calculates the weighting function as a product of the filter and the input spectrum to estimate the transfer function.

- Weighting vector — Applicable for frequency-domain data only. Specify a column vector of weights. This vector must have the same length as 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.

`'EnforceStability'` — Control whether to enforce stability of model

`false` (default) | `true`

Control whether to enforce stability of estimated model, specified as the comma-separated pair consisting of `'EnforceStability'` and either `true` or `false`.

Data Types: `logical`

`'EstCovar'` — Control whether to generate parameter covariance data

`true` (default) | `false`

Controls whether parameter covariance data is generated, specified as `true` or `false`.

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

`'Display'` — Specify whether to display the estimation progress

`'off'` (default) | `'on'`

Specify whether to display the estimation progress, specified as one of the following values:

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

`'InputOffset'` — Removal of offset from time-domain input data during estimation

[ ] (default) | vector of positive integers | matrix

Removal of offset from time-domain input data during estimation, specified as the comma-separated pair consisting of `'InputOffset'` and one of the following:

- A column vector of positive integers of length *Nu*, where *Nu* is the number of inputs.

- `[]` — Indicates no offset.
- *Nu*-by-*Ne* matrix — For multi-experiment data, specify `InputOffset` as an *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.

`'OutputOffset'` — Removal of offset from time-domain output data during estimation

[ ] (default) | vector | matrix

Removal of offset from time-domain output data during estimation, specified as the comma-separated pair consisting of `'OutputOffset'` and one of the following:

- A column vector of length *Ny*, where *Ny* is the number of outputs.
- `[]` — Indicates no offset.
- *Ny*-by-*Ne* matrix — For multi-experiment 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.

`'OutputWeight'` — Weighting of prediction errors in multi-output estimations

[ ] (default) | `'noise'` | positive semidefinite symmetric matrix

Weighting of prediction errors in multi-output estimations, specified as one of the following values:

- `'noise'` — Minimize $\det(E'^* E / N)$, where E represents the prediction error and `N` is the number of data samples. 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 `trace(E'*E*W/N)` where:
  - `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-output models, or the reliability of corresponding data.
  - `N` is the number of data samples.

- `[]` — The software chooses between the `'noise'` or using the identity matrix for `W`.

This option is relevant only for multi-output models.

`'Advanced'` — Additional advanced options
structure

Additional advanced options, specified as 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 for `n4sid`
`n4sidOptions` option set

Option set for `n4sid`, returned as an `n4sidOptions` option set.

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');
```

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

```
opt = n4sidOptions;
opt.InitialState = 'zero';
```

```
opt.Display = 'on';
```

More About

- “Loss Function and Model Quality Metrics”

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.

[4] Jansson, M. “Subspace identification and ARX modeling.” *13th IFAC Symposium on System Identification* , Rotterdam, The Netherlands, 2003.

See Also

`idfilt` | `idpar` | `n4sid`

Introduced in R2012a

ndims

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`

Introduced in R2012a

neuralnet

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 1- 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 the `Focus` option in `nlarxOptions` 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 Nonlinearity Estimator Using Feed-Forward Neural Network

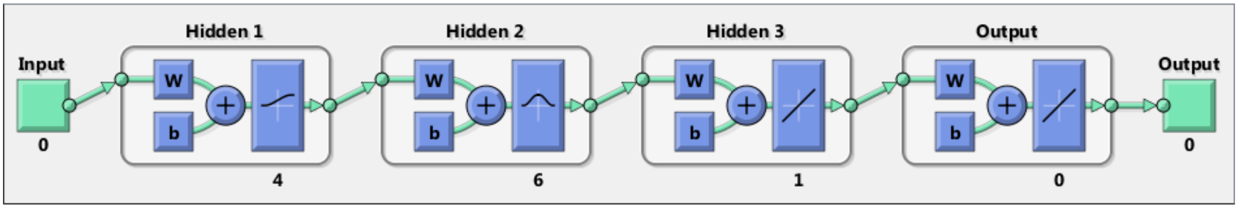
Create a neural network nonlinearity estimator using a feed-forward neural network with three hidden layers; transfer functions of types `logsig`, `radbas`,and `purelin`; and unknown input and output sizes.

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 neural network estimator.

```
net_estimator = neuralnet(net);
```

Estimate Nonlinear ARX Model Using a Neural Network Nonlinearity Estimator

Create a single-layer, cascade-forward network with unknown input and output sizes and use this network for nonlinear ARX model estimation.

Create a cascade-forward neural network with 20 neurons and unknown input-output sizes.

```
net = cascadeforwardnet(20);
```

Create a neural network nonlinearity estimator.

```
net_estimator = neuralnet(net);
```

Load estimation data.

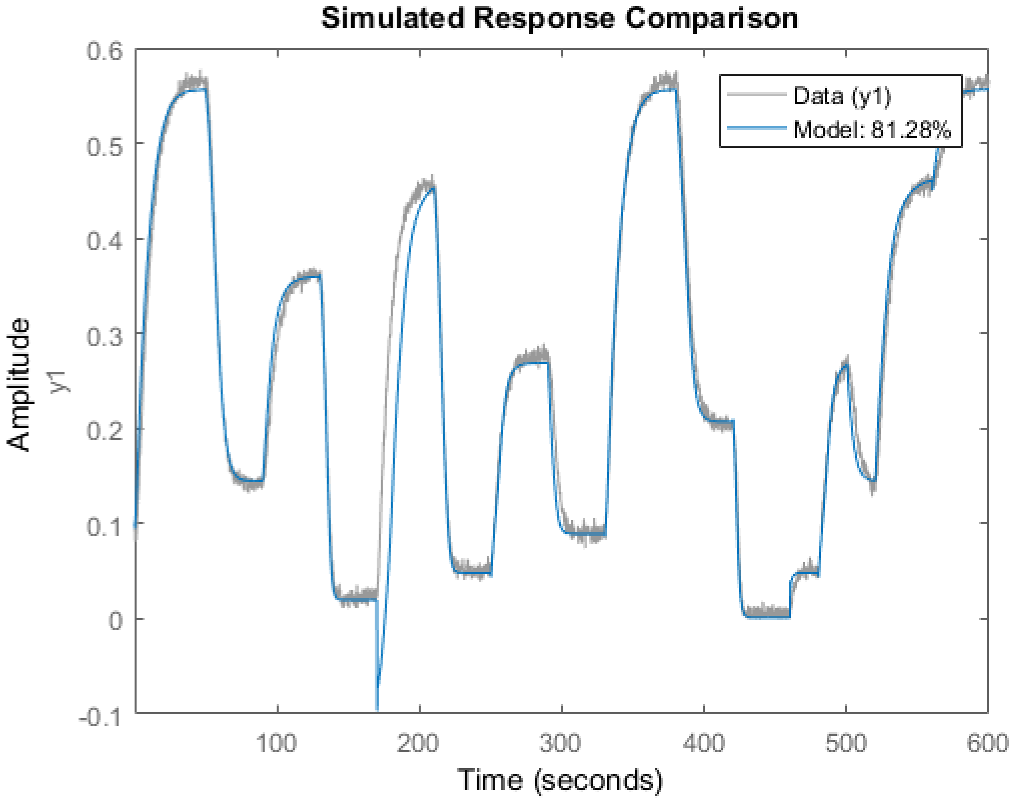
```
load twotankdata
Data = iddata(y,u,0.2);
```

Estimate nonlinear ARX model.

```
Model = nlarx(Data,[2 2 1],net_estimator);
```

Compare model response to measured output signal.

```
compare(Data,Model)
```



Initialize Input-Output Sizes of Neural Network Nonlinearity Estimator

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.

Load estimation data.

```
load iddata7 z7
z7 = z7(1:200);
```

Create a template nonlinear ARX model with no nonlinearity.

```
model = idnlarx([4 4 4 1 1],[]);
```

This model has six regressors and is 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.

Obtain the model regressor values.

```
R = getreg(model,'all',z7);
```

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 a neural network nonlinearity estimator.

```
net_estimator = neuralnet(net);
```

Specify the nonlinearity estimator in the model.

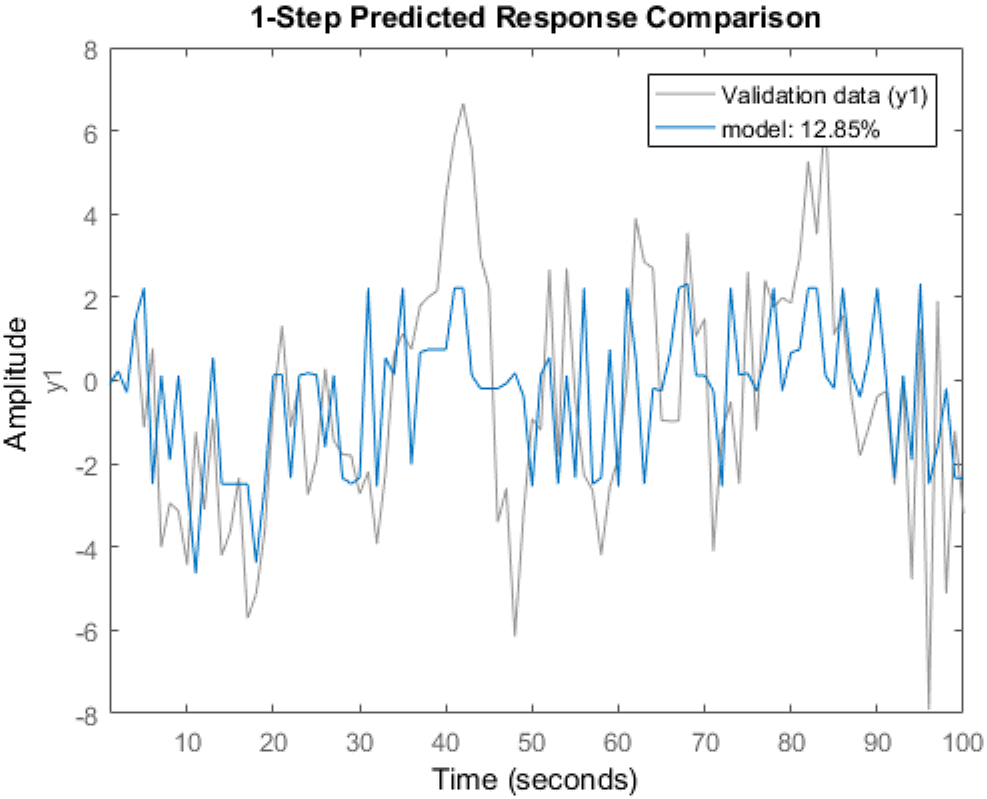
```
model.Nonlinearity = net_estimator;
```

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)
```



- “Identifying Nonlinear ARX Models”

More About

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`

Introduced in R2007a

nkshift

Shift data sequences

Syntax

```
Datas = nkshift(Data,nk)
```

Description

`Data` contains input-output data in the `iddata` format.

`nk` is a row vector with the same length as the number of input channels in `Data`.

`Datas` is an `iddata` object where the input channels in `Data` have been shifted according to `nk`. A positive value of `nk(ku)` means that input channel number `ku` is delayed `nk(ku)` samples.

`nkshift` supports both frequency- and time-domain data. For frequency-domain data it multiplies with $e^{ink\omega T}$ to obtain the same effect as shifting in the time domain. For continuous-time frequency-domain data (`Ts = 0`), `nk` should be interpreted as the shift in seconds.

`nkshift` lives in symbiosis with the `InputDelay` property of linear identified models:

```
m1 = ssest(dat,4,'InputDelay',nk)
```

is related to

```
m2 = ssest(nkshift(dat,nk),4);
```

such that `m1` and `m2` are the same models, but `m1` stores the delay information and uses this information when computing the frequency response, for example. When using `m2`, the delay value must be accounted for separately when computing time and frequency responses.

See Also

`idpoly` | `absorbDelay` | `idss` | `delayest`

Introduced before R2006a

nlarx

Estimate parameters of nonlinear ARX model

Syntax

```
sys = nlarx(Data,Orders)
sys = nlarx(Data,Orders,Nonlinearity)
sys = nlarx(Data,Orders,Nonlinearity,Name,Value)

sys = nlarx(Data,LinModel)
sys = nlarx(Data,LinModel,Nonlinearity)
sys = nlarx(Data,LinModel,Nonlinearity,Name,Value)

sys = nlarx(Data,sys0)

sys = nlarx(___,Options)
```

Description

`sys = nlarx(Data,Orders)` estimates a nonlinear ARX model to fit the given estimation data using the specified orders and a default wavelet network nonlinearity estimator.

`sys = nlarx(Data,Orders,Nonlinearity)` specifies the nonlinearity to use for model estimation.

`sys = nlarx(Data,Orders,Nonlinearity,Name,Value)` specifies additional attributes of the estimated model using one or more `Name,Value` pair arguments. These attributes include the nonlinear and custom regressor structure, and the data properties of the `idnlarx` model.

`sys = nlarx(Data,LinModel)` uses a linear ARX model, `LinModel`, to specify the model orders and the initial values of the linear coefficients of the model.

`sys = nlarx(Data,LinModel,Nonlinearity)` specifies the nonlinearity to use for model estimation.

`sys = nlarx(Data,LinModel,Nonlinearity,Name,Value)` specifies additional attributes of the `idnlarx` model structure using one or more `Name,Value` pair arguments.

`sys = nlarx(Data,sys0)` refines the parameters of the nonlinear ARX model, `sys0`.

Use this syntax to:

- Update the parameters of a previously estimated model to improve the fit to the estimation data. In this case, the estimation algorithm uses the parameters of `sys0` as initial guesses.
- Estimate the parameters of a model previously created using the `idnlarx` constructor. Prior to estimation, you can configure the model properties using dot notation.

`sys = nlarx( ___ ,Options)` specifies additional configuration options for the model estimation. Use `Options` with any of the previous syntaxes.

Examples

Estimate Nonlinear ARX Model with Default Settings

Load the estimation data.

```
load twotankdata;
```

Create an `iddata` object from the estimation data with a sample time of 0.2 min.

```
Ts = 0.2;
z = iddata(y,u,Ts);
```

Estimate the nonlinear ARX model.

```
sys = nlarx(z,[4 4 1]);
```

Estimate Nonlinear ARX Model from Time Series Data

Create time and data arrays.

```
dt = 0.01;
t = 0:dt:10;
```

```
y = 10*sin(2*pi*t)+rand(size(t));
```

Create an `iddata` object with no input signal specified.

```
z = iddata(y',[],dt);
```

Estimate the nonlinear ARX model.

```
sys = nlarx(z,2);
```

Estimate Nonlinear ARX Model with Specific Nonlinearity

Load the estimation data.

```
load twotankdata;
```

Create an `iddata` object from the estimation data.

```
z = iddata(y,u,0.2);
```

Create a wavelet network nonlinearity estimator with 5 units.

```
NL = wavenet('NumberOfUnits',5);
```

Estimate the nonlinear ARX model.

```
sys = nlarx(z,[4 4 1],NL);
```

Estimate Nonlinear ARX Model Using Custom Network Nonlinearity

Generating a custom network nonlinearity requires the definition of a user-defined unit function.

Define the unit function and save it as `gaussunit.m`.

```
% Copyright 2015 The MathWorks, Inc.

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

```
end
```

Create a custom network nonlinearity using the `gaussunit` function.

```
H = @gaussunit;
CNet = customnet(H);
```

Load the estimation data.

```
load iddata1;
```

Estimate a nonlinear ARX model using the custom network.

```
sys = nlarx(z1,[1 2 1],CNet);
```

Estimate MIMO Nonlinear ARX Model

Load the estimation data.

```
load motorizedcamera;
```

Create an `iddata` object.

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

`z` is an `iddata` object with 6 inputs and 2 outputs.

Specify the model orders.

```
Orders = [ones(2,2),2*ones(2,6),ones(2,6)];
```

Specify different nonlinearity estimators for each output channel.

```
NL = [wavenet('NumberOfUnits',2),linear];
```

Estimate the nonlinear ARX model.

```
sys = nlarx(z,Orders,NL);
```

Estimate MIMO Nonlinear ARX Model with Same Nonlinearity for All Outputs

Load the estimation data and create an `iddata` object.

```
load motorizedcamera;
```

```
z = iddata(y,u,0.02);
```

Specify the model orders.

```
Orders = [ones(2,2),2*ones(2,6),ones(2,6)];
```

Estimate a nonlinear ARX model using a sigmoidnet nonlinearity with 4 units for all output channels.

```
m = nlarx(z,Orders,sigmoidnet('numberOfUnits',4));
```

Estimate Nonlinear ARX Model with Custom Regressors

Load the estimation data.

```
load iddata1;
```

Create a cell array with two custom regressors.

```
C = {'y1(t-1)^2','y1(t-2)*u1(t-3)'};
```

Estimate a nonlinear ARX model with custom regressors and no standard regressors.

```
sys = nlarx(z1,[0 0 0],'linear','CustomRegressors',C);
```

Estimate Nonlinear ARX Model with Custom Regressor Objects

Load the estimation data.

```
load iddata1;
```

Define a custom regressor object for `y1(t-1)^2`.

```
C1 = customreg(@(x)x^2,{'y1'},[1]);
```

Define a custom regressor object for `y1(t-2)*u1(t-3)`.

```
C2 = customreg(@(x,y)x*y,{'y1','u1'},[2 3]);
```

Create a custom regressor object array.

```
C = [C1,C2];
```

Estimate a nonlinear ARX model with custom regressors.

```
sys = nlarx(z1,[0 0 0],'linear','CustomRegressors',C);
```

List the model regressors.

```
getreg(sys);
```

```
Regressors:
    y1(t-1)^2
    y1(t-2)*u1(t-3)
```

Estimate Nonlinear ARX Model Searching for Optimum Nonlinear Regressors

Load the estimation data.

```
load iddata1;
```

Estimate a Nonlinear ARX model using the `'search'` option.

```
sys = nlarx(z1,[4 4 1],'sigmoidnet','NonlinearRegressors','search');
```

List the model nonlinear regressor indices.

```
sys.NonlinearRegressors
```

```
ans =

     3     5     6     7
```

List all of the model regressors.

```
getreg(sys)
```

```
Regressors:
    y1(t-1)
    y1(t-2)
    y1(t-3)
    y1(t-4)
    u1(t-1)
    u1(t-2)
    u1(t-3)
    u1(t-4)
```

The optimum set of nonlinear regressors for this model includes `y1(t-3)`, `u1(t-1)`, `u1(t-2)`, and `u1(t-3)`.

Estimate Nonlinear ARX Model with No Linear Term in Nonlinearity Estimator

Load the estimation data.

```
load iddata1;
```

Create a sigmoid network nonlinearity estimator with no linear term.

```
SNL = sigmoidnet('LinearTerm','off');
```

Estimate the nonlinear ARX model.

```
sys = nlarx(z1,[2 2 1],SNL);
```

Specify Nonlinear ARX Orders and Linear Parameters Using Linear ARX Model

Load the estimation data.

```
load throttledata;
```

Detrend the data.

```
Tr = getTrend(ThrottleData);
Tr.OutputOffset = 15;
DetrendedData = detrend(ThrottleData,Tr);
```

Estimate the linear ARX model.

```
LinearModel = arx(DetrendedData,[2 1 1]);
```

Estimate the nonlinear ARX model using the linear model. The model orders, delays, and linear parameters of `NonlinearModel` are derived from `LinearModel`.

```
NonlinearModel = nlarx(ThrottleData,LinearModel);
```

Estimate Nonlinear ARX Model Using Constructed idnlarx Object

Load the estimation data.

```
load iddata1;
```

Create an `idnlarx` model.

```
sys = idnlarx([2 2 1]);
```

Configure the model using dot notation to set the following parameters:

- Use a sigmoid network nonlinearity
- Search for an opimum nonlinear regressor subset

```
sys.Nonlinearity = 'sigmoidnet';
sys.NonlinearRegressors = 'search';
```

Estimate a nonlinear ARX model with the structure and properties specified in the `idnlarx` object.

```
sys = nlarx(z1,sys);
```

Estimate Nonlinear ARX Model and Avoid Local Minima

If an estimation stops at a local minimum, you can perturb the model using `init` and reestimate the model.

Load the estimation data.

```
load iddata1;
```

Estimate the initial nonlinear model using specific nonlinear regressors.

```
sys1 = nlarx(z1,[4 2 1],'sigmoidnet','NonlinearRegressors',[1:3]);
```

Randomly perturb the model parameters to avoid local minima.

```
sys2 = init(sys1);
```

Estimate the new nonlinear model with the perturbed values.

```
sys2 = nlarx(z1,sys2);
```

Estimate Nonlinear ARX Model Using Specific Options

Load the estimation data.

```
load twotankdata;
```

Create an `iddata` object from the estimation data.

```
z = iddata(y,u,0.2);
```

Create an `nlarxOptions` option set specifying a simulation error minimization objective and a maximum of 50 estimation iterations.

```
opt = nlarxOptions;
opt.Focus = 'simulation';
opt.SearchOption.MaxIter = 50;
```

Estimate the nonlinear ARX model.

```
sys = nlarx(z,[4 4 1],'sigmoidnet',opt);
```

Estimate Regularized Nonlinear ARX Model with Large Number of Units

Load the regularization example data.

```
load regularizationExampleData.mat nldata;
```

Create a `sigmoidnet` nonlinearity with 30 units, and specify the model orders.

```
NL = sigmoidnet('NumberOfUnits',30);
Orders = [1 2 1];
```

Create an estimation option set and set the estimation search method to

```
opt = nlarxOptions('SearchMethod','lm');
```

Estimate an unregularized model.

```
sys = nlarx(nldata,Orders,NL,opt);
```

Configure the regularization `Lambda` parameter.

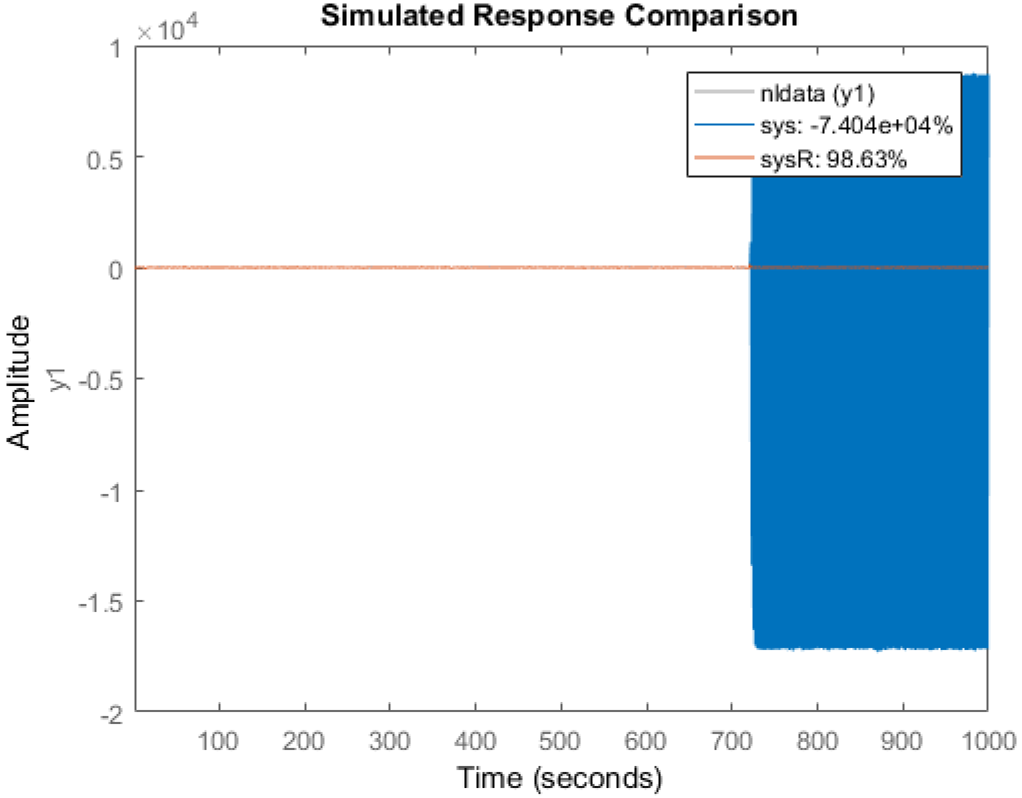
```
opt.Regularization.Lambda = 1e-8;
```

Estimate a regularized model.

```
sysR = nlarx(nldata,Orders,NL,opt);
```

Compare the two models.

```
compare(nldata,sys,sysR)
```



The large negative fit result for the unregularized model indicates a poor fit to the data. Estimating a regularized model produces a significantly better result.

- “Estimate Nonlinear ARX Models at the Command Line”
- “Estimate Nonlinear ARX Models Using Linear ARX Models”

Input Arguments

`Data` — Time-domain estimation data

`iddata` object

Time-domain estimation data, specified as an `iddata` object. `Data` can have one or more output channels and zero or more input channels. Data must be uniformly sampled and cannot contain missing (`NaN`) samples.

`Orders` — Model orders and delays

1-by-3 vector of positive integers | 1-by-3 vector of matrices

Model orders and delays for defining the regressor configuration, specified as a 1-by-3 vector, `[na nb nk]`.

For a model with n_y output channels and n_u input channels:

- `na` is an n_y-by-n_y matrix, where `na(i,j)` specifies the number of regressors from the *j*th output used to predict the *i*th output.
- `nb` is an n_y-by-n_u matrix, where `nb(i,j)` specifies the number of regressors from the *j*th input used to predict the *i*th output.
- `nk` is an n_y-by-n_u matrix, where `nk(i,j)` specifies the lag in the *j*th input used to predict the *i*th output.

```
na = [1 2; 2 3]
nb = [1 2 3; 2 3 1];
nk = [2 0 3; 1 0 5];
```

The estimation data for this system has three inputs (`u1`, `u2`, `u3`) and two outputs (`y1`, `y2`). Consider the regressors used to predict output, `y2(t)`:

- Since `na(2,:)` is `[2 3]`, the contributing regressors from the outputs are:
  - `y1(t-1)` and `y1(t-2)`
  - `y2(t-1)`, `y2(t-2)`, and `y2(t-3)`
- Since `nb(2,:)` is `[2 3 1]` and `nk(2,:)` is `[1 0 5]`, the contributing regressors from the inputs are:
  - `u1(t-1)` and `u1(t-2)`
  - `u2(t)`, `u2(t-1)`, and `u2(t-2)`
  - `u3(t-5)`

Note: The minimum lag for regressors based on output variables is always `1`, while the minimum lag for regressors based on input variables is dictated by `nk`. Use `getreg` to view the complete set of regressors used by the nonlinear ARX model.

`Nonlinearity` — Nonlinearity estimator

`'wavenet'` (default) | `'sigmoidnet'` | `'treepartition'` | `'linear'` | nonlinearity estimator object | array of nonlinearity estimator objects

Nonlinearity estimator, specified as one of the following:

`'wavenet'` or `wavenet` object	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 — Requires Neural Network Toolbox.
`customnet` object	Custom network — Similar to `sigmoidnet`, but with a user-defined replacement for the sigmoid function.

For more information, see “Nonlinearity Estimators for Nonlinear ARX Models”.

Specifying a character vector, for example `'sigmoidnet'`, creates a nonlinearity estimator object with default settings. Alternatively, you can specify nonlinearity estimator settings in two ways:

- Use the associated nonlinearity estimator function with Name-Value pair arguments.

  ```
  NL = sigmoidnet('NumberOfUnits',10);
  ```

- Create and modify a default nonlinearity estimator object.

  ```
  NL = sigmoidnet;
  NL.NumberOfUnits = 10;
  ```

For `ny` output channels, you can specify nonlinear estimators individually for each channel by setting `Nonlinearity` to an `ny`-by-1 array of nonlinearity estimator objects. To specify the same nonlinearity for all outputs, specify `Nonlinearity` as a character vector or a single nonlinearity estimator object.

Example: `’sigmoidnet’` specifies a sigmoid network nonlinearity with a default configuration.

Example: `treepartition('NumberOfUnits',5)` specifies a binary-tree nonlinearity with `5` terms in the binary tree expansion.

Example: `[wavenet('NumberOfUnits',10);sigmoidnet]` specifies different nonlinearity estimators for two output channels.

`LinModel` — Discrete time input-output polynomial model of ARX structure

`idpoly` model

Discrete time input-output polynomial model of ARX structure, specified as an `idpoly` model. Create this object using the `idpoly` constructor or estimate it using the `arx` command.

`sys0` — Nonlinear ARX model

`idnlarx` model

Nonlinear ARX model, specified as an `idnlarx` model. `sys0` can be:

- A model previously estimated using `nlarx`. The estimation algorithm uses the parameters of `sys0` as initial guesses. In this case, use `init` to slightly perturb the model properties to avoid being trapped in local minima.

  ```
  sys = init(sys);
  sys = nlarx(data,sys);
  ```

- A model previously created using `idnlarx` and with properties set using dot notation. Use this method to avoid complicated Name-Value pair syntax when configuring multiple model properties. For example, use

  ```
  sys1 = idnlarx([4 3 1]);
  sys1.Nonlinearity = 'treepartition';
  sys1.CustomRegressors = {'sin(u1(t-1))'};
  sys1.NonlinearRegressors = 'search';
  sys2 = nlarx(data,sys1);
  ```

  in place of the equivalent

  ```
  sys2 = nlarx(data,[4,3,1],'treepartition','CustomRegressors',...
      {'sin(u1(t-1))'},'NonlinearRegressors','search');
  ```

`Options` — Estimation options

`nlarxOptions` option set

Estimation options for nonlinear ARX model identification, specified as an `nlarxOptions` option set.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example: `'NonlinearRegressors','output'` specifies that only the regressors containing output variables are used as inputs to the nonlinear block of the model.

`'TimeVariable'` — Independent variable name

`'t'` (default) | character vector

Independent variable name, specified as the comma-separated pair consisting of `'TimeVariable'` and a character vector. For example, `'t'`.

`'CustomRegressors'` — Regressors constructed from combinations of inputs and outputs

`{}` (default) | cell array of character vectors | array of `customreg` objects

Regressors constructed from combinations of inputs and outputs, specified as the comma-separated pair consisting of `'CustomRegressors'` and one of the following for single-output systems:

- Cell array of character vectors. For example:
  - `{’y1(t-3)^3’,'y2(t-1)*u1(t-3)','sin(u3(t-2))'}`

  Each character vector must represent a valid formula for a regressor contributing towards the prediction of the model output. The formula must be written using the input and output names and the time variable name as variables.
- Array of custom regressor objects, created using `customreg` or `polyreg`.

For a model with n_y outputs, specify an n_y-by-1 cell array of `customreg` object arrays or character arrays.

These regressors are in addition to the standard regressors based on `Orders`.

Example: `'CustomRegressors',{’y1(t-3)^3’,'y2(t-1)*u1(t-3)'}`

Example: `'CustomRegressors',{'sin(u3(t-2))'}`

`'NonlinearRegressors'` — Subset of regressors that enter as inputs to the nonlinear block of the model

`'all'` (default) | `'output'` | `'input'` | `'standard'` | `'custom'` | `'search'` | vector of positive integers | `[]` | cell array

Subset of regressors that enter as inputs to the nonlinear block of the model, specified as the comma-separated pair consisting of `'NonlinearRegressors'` and one of the following values:

- `'all'` — All regressors
- `'output'` — Regressors containing output variables
- `'input'` — Regressors containing input variables
- `'standard'` — Standard regressors
- `'custom'` — Custom regressors
- `'search'` — The estimation algorithm performs a search for the best regressor subset. This is useful when you want to reduce a large number of regressors entering the nonlinear function block of the nonlinearity estimator. This option must be applied to all output models simultaneously.
- `[]` — No regressors. This creates a linear-in-regressor model.
- Vector of regressor indices. To determine the number and order of regressors, use `getreg`.

For a model with multiple outputs, specify a cell array of n_y elements, where n_y is the number of output channels. For each output, specify one of the preceding options. Alternatively, to apply the same regressor subset to all model outputs, specify `[]` or any of the character vector options alone, for example `'standard'`.

Example: `'NonlinearRegressors','search'` performs a best regressor search for the only output of a single output model, or all of the outputs of a multiple output model.

Example: `'NonlinearReg','input'` applies only input regressors to the inputs of the nonlinear function.

Example: `'NonlinearRegressors',{'input','output'}` applies input regressors to the first output, and output regressors to the second output of a model with two outputs.

Output Arguments

`sys` — Nonlinear ARX model

`idnlarx` object

Nonlinear ARX model that fits the given estimation data, returned as an `idnlarx` object. This model is created using the specified model orders, nonlinearity estimator, and estimation options.

Information about the estimation results and options used is stored in the `Report` property of the model. The contents of `Report` depend upon the choice of nonlinearity and estimation focus you specified for `nlarx`. `Report` has the following fields:

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Status</code></td><td>Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.</td></tr>
<tr><td><code>Method</code></td><td>Estimation command used.</td></tr>
<tr><td><code>Fit</code></td><td>Quantitative assessment of the estimation, returned as a structure. See “Loss Function and Model Quality Metrics” for more information on these quality metrics. The structure has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>FitPerc</code></td><td>Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as a percentage.</td></tr>
<tr><td><code>LossFcn</code></td><td>Value of the loss function when the estimation completes.</td></tr>
<tr><td><code>MSE</code></td><td>Mean squared error (MSE) measure of how well the response of the model fits the estimation data.</td></tr>
<tr><td><code>FPE</code></td><td>Final prediction error for the model.</td></tr>
<tr><td><code>AIC</code></td><td>Raw Akaike Information Criteria (AIC) measure of model quality.</td></tr>
<tr><td><code>AICc</code></td><td>Small sample-size corrected AIC.</td></tr>
<tr><td><code>nAIC</code></td><td>Normalized AIC.</td></tr>
<tr><td><code>BIC</code></td><td>Bayesian Information Criteria (BIC).</td></tr>
</table>
</td></tr>
<tr><td><code>Parameter</code></td><td>Estimated values of model parameters.</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>OptionsUs</code></td><td>Option set used for estimation. If no custom options were configured, this is a set of default options. See <code>nlarxOptions</code> for more information.</td></tr>
<tr><td><code>RandState</code></td><td>State of the random number stream at the start of estimation. Empty, <code>[]</code>, if randomization was not used during estimation. For more information, see <code>rng</code> in the MATLAB documentation.</td></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation, returned as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
<tr><td><code>Type</code></td><td>Data type.</td></tr>
<tr><td><code>Length</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time.</td></tr>
<tr><td><code>InterSam</code></td><td>Input intersample behavior, returned as one of the following values:
• <code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
• <code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
• <code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.</td></tr>
<tr><td><code>InputOff</code></td><td>Offset removed from time-domain input data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
<tr><td><code>OutputOf</code></td><td>Offset removed from time-domain output data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Terminati</code></td><td>Termination conditions for the iterative search used for prediction error minimization. Structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>WhyStop</code></td><td>Reason for terminating the numerical search.</td></tr>
<tr><td><code>Iteratio</code></td><td>Number of search iterations performed by the estimation algorithm.</td></tr>
<tr><td><code>FirstOrd</code></td><td>∞ -norm of the gradient search vector when the search algorithm terminates.</td></tr>
<tr><td><code>FcnCount</code></td><td>Number of times the objective function was called.</td></tr>
<tr><td><code>UpdateNo</code></td><td>Norm of the gradient search vector in the last iteration. Omitted when the search method is <code>'lsqnonlin'</code>.</td></tr>
<tr><td><code>LastImpr</code></td><td>Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is <code>'lsqnonlin'</code>.</td></tr>
<tr><td><code>Algorith</code></td><td>Algorithm used by <code>'lsqnonlin'</code> search method. Omitted when other search methods are used.</td></tr>
</table>
For estimation methods that do not require numerical search optimization, the <code>Termination</code> field is omitted.</td></tr>
</table>

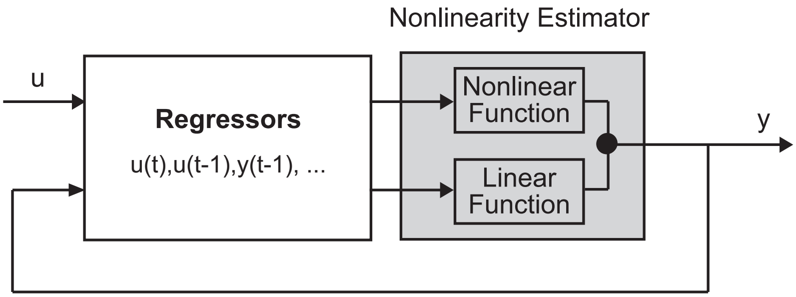
For more information on using `Report`, see “Estimation Report”.

More About

Algorithms

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 multilayer 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:

$$F(x) = L^T(x-r) + d + g\big(Q(x-r)\big)$$

x is a vector of the regressors. $L^T(x)+d$ is the output of the linear function block and is affine when d≠0. d is a scalar offset. $g\big(Q(x-r)\big)$ 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.

- “Structure of Nonlinear ARX Models”
- “Ways to Configure Nonlinear ARX Estimation”
- “Validating Nonlinear ARX Models”
- “Using Nonlinear ARX Models”
- “Loss Function and Model Quality Metrics”
- “Regularized Estimates of Model Parameters”
- “Estimation Report”

See Also

`aic` | `fpe` | `goodnessofFit` | `idnlarx` | `isnlarx` | `nlarxOptions`

Introduced in R2007a

nlarxOptions

Option set for `nlarx`

Syntax

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

Description

`opt = nlarxOptions` creates the default option set for `nlarx`. Use dot notation to modify this option set for your specific application. Any options that you do not modify retain their default values.

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

Examples

Create Default Option Set for Nonlinear ARX Estimation

```
opt = nlarxOptions;
```

Create and Modify Default Nonlinear ARX Option Set

Create a default option set for `nlarx`, and use dot notation to modify specific options.

```
opt = nlarxOptions;
```

Turn on the estimation progress display.

```
opt.Display = 'on';
```

Minimize the norm of the simulation error.

```
opt.Focus = 'simulation';
```

Use a subspace Gauss-Newton least squares search with a maximum of 25 iterations.

```
opt.SearchMethod = 'gn';
opt.SearchOption.MaxIter = 25;
```

Specify Options for Nonlinear ARX Estimation

Create an option set for `nlarx` specifying the following options:

- Turn off iterative estimation for the default wavelet network estimation.
- Turn on the estimation progress-viewer display.

```
opt = nlarxOptions('IterWavenet','off','Display','on');
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example: `'Focus','simulation','SearchMethod','grad'` specifies that the norm of the simulation error is minimized using a steepest descent least squares search.

`'Focus'` — Minimization objective

`'prediction'` (default) | `'simulation'`

Minimization objective, specified as the comma-separated pair consisting of `'Focus'` and one of the following:

- `'prediction'` — Minimize the norm of the prediction error, which is defined as the difference between the measured output and the one-step ahead predicted response of the model.
- `'simulation'` — Minimize the norm of the simulation error, which is defined as the difference between the measured output and simulated response of the model.

`'Display'` — Estimation progress display setting

`'off'` (default) | `'on'`

Estimation progress display setting, specified as the comma-separated pair consisting of `'Display'` and one of the following:

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

`'OutputWeight'` — Weighting of prediction error in multi-output estimations

`'noise'` (default) | positive semidefinite matrix

Weighting of prediction error in multi-output model estimations, specified as the comma-separated pair consisting of `'OutputWeight'` and one of the following:

- `'noise'` — Optimal weighting is automatically computed as the inverse of the estimated noise variance. This weighting minimizes `det(E'*E)`, where `E` is the matrix of prediction errors. This option is not available when using `'lsqnonlin'` as a `'SearchMethod'`.
- A positive semidefinite matrix, `W`, of size equal to the number of outputs. This weighting minimizes `trace(E'*E*W/N)`, where `E` is the matrix of prediction errors and `N` is the number of data samples.

`'IterWavenet'` — Iterative `wavenet` estimation setting

`'auto'` (default) | `'on'` | `'off'`

Iterative `wavenet` estimation setting, specified as the comma-separated pair consisting of `'IterWavenet'` and one of the following:

- `'auto'` — First estimation is noniterative and subsequent estimations are iterative.
- `'on'` — Perform iterative estimation only.
- `'off'` — Perform noniterative estimation only.

This option applies only when using a `wavenet` nonlinearity estimator.

`'Regularization'` — Options for regularized estimation of model parameters

structure

Options for regularized estimation of model parameters, specified as the comma-separated pair consisting of `'Regularization'` and a structure with fields:

Field Name	Description	Default
`Lambda`	Bias versus variance trade-off constant, specified as a nonnegative scalar.	`0` — Indicates no regularization.

Field Name	Description	Default
`R`	Weighting matrix, specified as a vector of nonnegative scalars or a square positive semidefinite matrix. The length must be equal to the number of free parameters in the model, `np`. Use the `nparams` command to determine the number of model parameters.	`1` — Indicates a value of `eye(np)`.
`Nomina`	The nominal value towards which the free parameters are pulled during estimation, specified as one of the following: • `'zero'` — Pull parameters towards zero. • `'model'` — Pull parameters towards preexisting values in the initial model. Use this option only when you have a well-initialized `idnlarx` model with finite parameter values.	`'zero'`

To specify field values in `Regularization`, create a default `nlarxOptions` set and modify the fields using dot notation. Any fields that you do not modify retain their default values.

```
opt = nlarxOptions;
opt.Regularization.Lambda = 1.2;
opt.Regularization.R = 0.5*eye(np);
```

Regularization is a technique for specifying model flexibility constraints, which reduce uncertainty in the estimated parameter values. For more information, see “Regularized Estimates of Model Parameters”.

`'SearchMethod'` — Numerical search method used for iterative parameter estimation

`'auto'` (default) | `'gn'` | `'gna'` | `'lm'` | `'grad'` | `'lsqnonlin'`

Numerical search method used for iterative parameter estimation, specified as the comma-separated pair consisting of `'SearchMethod'` and one of the following:

- `'auto'` — A combination of the line search algorithms, `'gn'`, `'lm'`, `'gna'`, and `'grad'` methods is tried at each iteration. The descent direction leading to the largest reduction in estimation cost is used.
- `'gn'` — Subspace Gauss-Newton least squares search. Singular values of the Jacobian matrix less than `GnPinvConst*eps*max(size(J))*norm(J)` are

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

- `'gna'` — Adaptive subspace Gauss-Newton search. Eigenvalues less than `gamma*max(sv)` of the Hessian are ignored, where sv contains the singular values of the Hessian. The Gauss-Newton direction is computed in the remaining subspace. gamma has the initial value `InitGnaTol` (see `Advanced` in `'SearchOption'` 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 five bisections. This value is decreased by the factor `2*LMStep` each time a search is successful without any bisections.
- `'lm'` — Levenberg-Marquardt least squares search, where the next parameter value is `-pinv(H+d*I)*grad` from the previous one. 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.
- `'grad'` — Steepest descent least squares search.
- `'lsqnonlin'` — Trust region reflective algorithm provided by Optimization Toolbox. This method cannot be used with the `'OutputWeight'` option`'noise'`. See `lsqnonlin` for more information.

`'SearchOption'` — Options set for the search algorithm

search option set

Options set for the search algorithm, specified as the comma-separated pair consisting of `'SearchOption'` and a search option set with fields that depend on the value of `SearchMethod`:

`SearchOption` Structure When `SearchMethod` Is Specified as `'gn'`, `'gna'`, `'lm'`, `'grad'`, or `'auto'`

Field Name	Description	Default
`Toleran`	Minimum percentage difference between the current value of the loss function and its expected improvement after the next iteration, specified as a positive scalar. When the percentage of expected improvement is less than `Tolerance`, the iterations stop. The estimate of the expected loss-function improvement at the next iteration is based on the Gauss-Newton vector computed for the current parameter value.	`0.01`

Field Name	Description	Default
`MaxIter`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIter` is reached or another stopping criterion is satisfied, such as `Tolerance`. Setting `MaxIter = 0` returns the result of the start-up procedure. Use `sys.Report.Termination.Iterations` to get the actual number of iterations during an estimation, where *sys* is an `idtf` model.	`20`
`Advance`	Advanced search settings, specified as a structure with the following fields:	

Field Name	Description	Default
`GnPinvCons`	Jacobian matrix singular value threshold, specified as a positive scalar. Singular values of the Jacobian matrix that are smaller than `GnPinvConst*max(size(J)*norm(J)*eps)` are discarded when computing the search direction. Applicable when `SearchMethod` is `'gn'`.	`10000`
`InitGnaTol`	Initial value of *gamma*, specified as a positive scalar. Applicable when `SearchMethod` is `'gna'`.	`0.0001`
`LMStartVal`	Starting value of search-direction length *d* in the Levenberg-Marquardt method, specified as a positive scalar. Applicable when `SearchMethod` is `'lm'`.	`0.001`
`LMStep`	Size of the Levenberg-Marquardt step, specified as a positive integer. 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'`.	`2`
`MaxBisecti`	Maximum number of bisections used for line search along the search direction, specified as a positive integer.	`25`

<table>
<tr><th>Field Name</th><th>Description</th><th>Default</th></tr>
<tr><td></td><td colspan="2">
<table>
<tr><th>Field Name</th><th>Description</th><th>Default</th></tr>
<tr><td><code>MaxFunEval</code></td><td>Maximum number of calls to the model file, specified as a positive integer. Iterations stop if the number of calls to the model file exceeds this value.</td><td><code>Inf</code></td></tr>
<tr><td><code>MinParChan</code></td><td>Smallest parameter update allowed per iteration, specified as a nonnegative scalar.</td><td><code>0</code></td></tr>
<tr><td><code>RelImprove</code></td><td>Relative improvement threshold, specified as a nonnegative scalar. Iterations stop if the relative improvement of the criterion function is less than this value.</td><td><code>0</code></td></tr>
<tr><td><code>StepReduct</code></td><td>Step reduction factor, specified as a positive scalar that is greater than 1. The suggested parameter update is reduced by the factor <code>StepReduction</code> after each try. This reduction continues until <code>MaxBisections</code> tries are completed or a lower value of the criterion function is obtained.
<code>StepReduction</code> is not applicable for <code>SearchMethod</code> <code>'lm'</code> (Levenberg-Marquardt method).</td><td><code>2</code></td></tr>
</table>
</td></tr>
</table>

`SearchOption` Structure When `SearchMethod` Is Specified as `'lsqnonlin'`

Field Name	Description	Default
`TolFun`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar. The value of `TolFun` is the same as that of `opt.SearchOption.Advanced.TolFun`.	`1e-5`
`TolX`	Termination tolerance on the estimated parameter values, specified as a positive scalar.	`1e-6`

Field Name	Description	Default
	The value of `TolX` is the same as that of `opt.SearchOption.Advanced.TolX`.	
`MaxIter`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIter` is reached or another stopping criterion is satisfied, such as `TolFun`. The value of `MaxIter` is the same as that of `opt.SearchOption.Advanced.MaxIter`.	`20`
`Advance`	Advanced search settings, specified as an option set for `lsqnonlin`. For more information, see the Optimization Options table in “Optimization Options”.	Use `optimset('lsqnonlin')` to create a default option set.

To specify field values in `SearchOption`, create a default `nlarxOptions` set and modify the fields using dot notation. Any fields that you do not modify retain their default values.

```
opt = nlarxOptions;
opt.SearchOption.MaxIter = 15;
opt.SearchOption.Advanced.RelImprovement = 0.5;
```

`'Advanced'` — Additional advanced options

structure

Additional advanced options, specified as the comma-separated pair consisting of `'Advanced'` and a structure with fields:

Field Name	Description	Default
`ErrorThre`	Threshold for when to adjust the weight of large errors from quadratic to linear, specified as a nonnegative scalar. Errors larger than `ErrorThreshold` times the estimated standard deviation have a linear weight in the loss function. The standard deviation is estimated robustly as the median of the absolute deviations from the median of the prediction errors, divided by 0.7. If	`0` — Leads to a purely quadratic loss function.

Field Name	Description	Default
	your estimation data contains outliers, try setting `ErrorThreshold` to `1.6`.	
`MaxSize`	Maximum number of elements in a segment when input-output data is split into segments, specified as a positive integer.	`250000`

To specify field values in `Advanced`, create a default `nlarxOptions` set and modify the fields using dot notation. Any fields that you do not modify retain their default values.

```
opt = nlarxOptions;
opt.Advanced.ErrorThreshold = 1.2;
```

Output Arguments

`opt` — Option set for `nlarx` command

`nlarxOptions` option set

Option set for `nlarx` command, returned as an `nlarxOptions` option set.

See Also

`nlarx`

Introduced in R2015a

nlgreyest

Estimate nonlinear grey-box model parameters

Syntax

```
sys= nlgreyest(data,init_sys)
sys= nlgreyest(data,init_sys,options)
```

Description

`sys= nlgreyest(data,init_sys)` estimates the parameters of a nonlinear grey-box model, `init_sys`, using time-domain data, `data`.

`sys= nlgreyest(data,init_sys,options)` specifies additional model estimation options.

Examples

Selectively Estimate Parameters of Nonlinear Grey-Box Model

Load data.

```
load(fullfile(matlabroot,'toolbox','ident','iddemos','data','twotankdata'));
z = iddata(y,u,0.2,'Name','Two tanks');
```

The data contains 3000 input-output data samples of a two tank system. The input is the voltage applied to a pump, and the output is the liquid level of the lower tank.

Specify file describing the model structure for a two-tank system. The file specifies the state derivatives and model outputs as a function of time, states, inputs, and model parameters.

```
FileName = 'twotanks_c';
```

Specify model orders [ny nu nx].

```
Order = [1 1 2];
```

Specify initial parameters (Np = 6).

```
Parameters = {0.5;0.0035;0.019; ...
    9.81;0.25;0.016};
```

Specify initial initial states.

```
InitialStates = [0;0.1];
```

Specify as continuous system.

```
Ts = 0;
```

Create `idnlgrey` model object.

```
nlgr = idnlgrey(FileName,Order,Parameters,InitialStates,Ts, ...
    'Name','Two tanks');
```

Set some parameters as constant.

```
nlgr.Parameters(1).Fixed = true;
nlgr.Parameters(4).Fixed = true;
nlgr.Parameters(5).Fixed = true;
```

Estimate the model parameters.

```
nlgr = nlgreyest(z,nlgr);
```

Estimate a Nonlinear Grey-Box Model Using Specific Options

Create estimation option set for `nlgreyest` to view estimation progress, and to set the maximum iteration steps to 50.

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

Load data.

```
load(fullfile(matlabroot,'toolbox','ident','iddemos','data','dcmotordata'));
z = iddata(y,u,0.1,'Name','DC-motor');
```

The data is from a linear DC motor with one input (voltage), and two outputs (angular position and angular velocity). The structure of the model is specified by `dcmotor_m.m` file.

Create a nonlinear grey-box model.

```
file_name = 'dcmotor_m';
Order = [2 1 2];
Parameters = [1;0.28];
InitialStates = [0;0];

init_sys = idnlgrey(file_name,Order,Parameters,InitialStates,0, ...
    'Name','DC-motor');
```

Estimate the model parameters using the estimation options.

```
sys = nlgreyest(z,init_sys,opt);
```

- "Creating IDNLGREY Model Files"
- "Represent Nonlinear Dynamics Using MATLAB File for Grey-Box Estimation"

Input Arguments

`data` — Time domain data

`iddata` object

Time-domain estimation data, specified as an `iddata` object. `data` has the same input and output dimensions as `init_sys`.

`init_sys` — Constructed nonlinear grey-box model

`idnlgrey` object

Constructed nonlinear grey-box model that configures the initial parameterization of `sys`, specified as an `idnlgrey` object. `init_sys` has the same input and output dimensions as `data`. Create `init_sys` using `idnlgrey`.

`options` — Estimation options

`nlgreyestOptions` option set

Estimation options for nonlinear grey-box model identification, specified as an `nlgreyestOptions` option set.

Output Arguments

`sys` — Estimated nonlinear grey-box model

`idnlgrey` object

Nonlinear grey-box model with the same structure as `init_sys`, returned as an `idnlgrey` object. The parameters of `sys` are estimated such that the response of `sys` matches the output signal in the estimation data.

Information about the estimation results and options used is stored in the `Report` property of the model. `Report` has the following fields:

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Status</code></td><td>Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.</td></tr>
<tr><td><code>Method</code></td><td>Name of the simulation solver and the search method used during estimation.</td></tr>
<tr><td><code>Fit</code></td><td>Quantitative assessment of the estimation, returned as a structure. See “Loss Function and Model Quality Metrics” for more information on these quality metrics. The structure has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>FitPerc</code></td><td>Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as a percentage.</td></tr>
<tr><td><code>LossFcn</code></td><td>Value of the loss function when the estimation completes.</td></tr>
<tr><td><code>MSE</code></td><td>Mean squared error (MSE) measure of how well the response of the model fits the estimation data.</td></tr>
<tr><td><code>FPE</code></td><td>Final prediction error for the model.</td></tr>
<tr><td><code>AIC</code></td><td>Raw Akaike Information Criteria (AIC) measure of model quality.</td></tr>
<tr><td><code>AICc</code></td><td>Small sample-size corrected AIC.</td></tr>
<tr><td><code>nAIC</code></td><td>Normalized AIC.</td></tr>
<tr><td><code>BIC</code></td><td>Bayesian Information Criteria (BIC).</td></tr>
</table>
</td></tr>
<tr><td><code>Parameter</code></td><td>Estimated values of the model parameters. Structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>InitialValues</code></td><td>Structure with values of parameters and initial states before estimation.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>ParVector</code></td><td>Value of parameters after estimation.</td></tr>
<tr><td><code>Free</code></td><td>Logical vector specifying the fixed or free status of parameters during estimation</td></tr>
<tr><td><code>FreeParCovarian</code></td><td>Covariance of the free parameters.</td></tr>
<tr><td><code>X0</code></td><td>Value of initial states after estimation.</td></tr>
<tr><td><code>X0Covariance</code></td><td>Covariance of the initial states.</td></tr>
</table>
</td></tr>
<tr><td><code>OptionsUs</code></td><td>Option set used for estimation. If no custom options were configured, this is a set of default options. See <code>nlgreyestOptions</code> for more information.</td></tr>
<tr><td><code>RandState</code></td><td>State of the random number stream at the start of estimation. Empty, <code>[]</code>, if randomization was not used during estimation. For more information, see <code>rng</code> in the MATLAB documentation.</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation — Structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
<tr><td><code>Type</code></td><td>Data type — For <code>idnlgrey</code> models, this is set to <code>'Time domain data'</code>.</td></tr>
<tr><td><code>Length</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time. This is equivalent to <code>data.Ts</code>.</td></tr>
<tr><td><code>InterSam</code></td><td>Input intersample behavior. One of the following values:
• <code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
• <code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
• <code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.
The value of <code>Intersample</code> has no effect on estimation results for discrete-time models.</td></tr>
<tr><td><code>InputOff</code></td><td>Empty, <code>[]</code>, for nonlinear estimation methods.</td></tr>
<tr><td><code>OutputOf</code></td><td>Empty, <code>[]</code>, for nonlinear estimation methods.</td></tr>
</table>
</td></tr>
</table>

Report Field	Description
`Terminati`	Termination conditions for the iterative search used for prediction error minimization. Structure with the following fields: <table><tr><th>Field</th><th>Description</th></tr><tr><td><code>WhyStop</code></td><td>Reason for terminating the numerical search.</td></tr><tr><td><code>Iteratio</code></td><td>Number of search iterations performed by the estimation algorithm.</td></tr><tr><td><code>FirstOrd</code></td><td>∞ -norm of the gradient search vector when the search algorithm terminates.</td></tr><tr><td><code>FcnCount</code></td><td>Number of times the objective function was called.</td></tr><tr><td><code>UpdateNo</code></td><td>Norm of the gradient search vector in the last iteration. Omitted when the search method is <code>'lsqnonlin'</code>.</td></tr><tr><td><code>LastImpr</code></td><td>Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is <code>'lsqnonlin'</code>.</td></tr><tr><td><code>Algorith</code></td><td>Algorithm used by <code>'lsqnonlin'</code> search method. Omitted when other search methods are used.</td></tr></table> For estimation methods that do not require numerical search optimization, the `Termination` field is omitted.

For more information, see “Estimation Report”.

More About

- “Estimate Nonlinear Grey-Box Models”
- “Loss Function and Model Quality Metrics”
- “Regularized Estimates of Model Parameters”
- “Estimation Report”

See Also

`aic` | `fpe` | `goodnessofFit` | `idnlgrey` | `nlgreyestOptions` | `pem`

Introduced in R2015a

nlgreyestOptions

Option set for `nlgreyest`

Syntax

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

Description

`opt = nlgreyestOptions` creates the default option set for `nlgreyest`. Use dot notation to customize the option set, if needed.

`opt = nlgreyestOptions(Name,Value)` creates an option set with options specified by one or more `Name,Value` pair arguments. The options that you do not specify retain their default value.

Examples

Create Default Option Set for Nonlinear Grey-Box Model Estimation

```
opt = nlgreyestOptions;
```

Estimate a Nonlinear Grey-Box Model Using Specific Options

Create estimation option set for `nlgreyest` to view estimation progress, and to set the maximum iteration steps to 50.

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

Load data.

```
load(fullfile(matlabroot,'toolbox','ident','iddemos','data','dcmotordata'));
z = iddata(y,u,0.1,'Name','DC-motor');
```

The data is from a linear DC motor with one input (voltage), and two outputs (angular position and angular velocity). The structure of the model is specified by `dcmotor_m.m` file.

Create a nonlinear grey-box model.

```
file_name = 'dcmotor_m';
Order = [2 1 2];
Parameters = [1;0.28];
InitialStates = [0;0];

init_sys = idnlgrey(file_name,Order,Parameters,InitialStates,0, ...
    'Name','DC-motor');
```

Estimate the model parameters using the estimation options.

```
sys = nlgreyest(z,init_sys,opt);
```

Specify Options for Nonlinear Grey-Box Model Estimation

Create an option set for `nlgreyest` where:

- Parameter covariance data is not generated.
- Subspace Gauss-Newton least squares method is used for estimation.

```
opt = nlgreyestOptions('EstCovar',false,'SearchMethod','gn');
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example: `nlgreyestOptions('Display','on')`

`'GradientOptions'` — Options for computing Jacobians and gradients

structure

Options for computing Jacobians and gradients, specified as the comma-separated pair consisting of `'GradientOptions'` and a structure with fields:

Field Name	Description	Default
`DiffMaxChange`	Largest allowed parameter perturbation when computing numerical derivatives. Specified as a positive real value > `'DiffMinChange'`.	`Inf`
`DiffMinChange`	Smallest allowed parameter perturbation when computing numerical derivatives. Specified as a positive real value <`'DiffMaxChange'`.	`0.01*sqrt(eps)`
`DiffScheme`	Method for computing numerical derivatives with respect to the components of the parameters and/or the initial state(s) to form the Jacobian. Specified as one of the following: • `'Auto'` - Automatically chooses from the following methods. • `'Central approximation'` • `'Forward approximation'` • `'Backward approximation'`	`'Auto'`
`GradientType`	Method used when computing derivatives (Jacobian) of the parameters or the initial states to be estimated. Specified as one of the following: • `'Auto'` — Automatically chooses from the following methods. • `'Basic'` — Individually computes all numerical derivatives required to form each column of the Jacobian.	`'Auto'`

Field Name	Description	Default
	- `'Refined'` — Simultaneously computes all numerical derivatives required to form each column of the Jacobian.	

To specify field values in `GradientOptions`, create a default `nlgreyestOptions` set and modify the fields using dot notation. Any fields that you do not modify retain their default values.

```
opt = nlgreyestOptions;
opt.GradientOptions.GradientType = 'Basic';
```

`'EstCovar'` — Parameter covariance data generation setting

`1` or `true` (default) | `0` or `false`

Controls whether parameter covariance data is generated, specified as `true` (`1`) or `false` (`0`).

`'Display'` — Estimation progress display setting

`'off'` (default) | `'on'`

Estimation progress display setting, specified as the comma-separated pair consisting of `'Display'` and one of the following:

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

`'Regularization'` — Options for regularized estimation of model parameters

structure

Options for regularized estimation of model parameters, specified as the comma-separated pair consisting of `'Regularization'` and a structure with fields:

Field Name	Description	Default
`Lambda`	Bias versus variance trade-off constant, specified as a nonnegative scalar.	`0` — Indicates no regularization.
`R`	Weighting matrix, specified as a vector of nonnegative scalars or a square positive semi-definite matrix. The	`1` — Indicates a value of `eye(np)`.

Field Name	Description	Default
	length must be equal to the number of free parameters in the model, `np`. Use the `nparams` command to determine the number of model parameters.	
`Nomina`	The nominal value towards which the free parameters are pulled during estimation specified as one of the following: • `'zero'` — Pull parameters towards zero. • `'model'` — Pull parameters towards pre-existing values in the initial model.	`'zero'`

To specify field values in `Regularization`, create a default `nlgreyestOptions` set and modify the fields using dot notation. Any fields that you do not modify retain their default values.

```
opt = nlgreyestOptions;
opt.Regularization.Lambda = 1.2;
opt.Regularization.R = 0.5*eye(np);
```

Regularization is a technique for specifying model flexibility constraints, which reduce uncertainty in the estimated parameter values. For more information, see "Regularized Estimates of Model Parameters".

`'SearchMethod'` — Numerical search method used for iterative parameter estimation

`'auto'` (default) | `'gn'` | `'gna'` | `'lm'` | `'grad'` | `'lsqnonlin'`

Numerical search method used for iterative parameter estimation, specified as the comma-separated pair consisting of `'SearchMethod'` and one of the following:

- `'auto'` — If Optimization Toolbox is available, `'lsqnonlin'` is used. Otherwise, a combination of the line search algorithms, `'gn'`, `'lm'`, `'gna'`, and `'grad'` methods is tried at each iteration. The descent direction leading to the largest reduction in estimation cost is used.
- `'gn'` — Subspace Gauss-Newton least squares search. Singular values of the Jacobian matrix less than `GnPinvConst*eps*max(size(J))*norm(J)` are discarded when computing the search direction. J is the Jacobian matrix. The Hessian matrix is approximated by J^TJ. If there is no improvement in this direction, the function tries the gradient direction.

- `'gna'` — Adaptive subspace Gauss-Newton search. Eigenvalues less than `gamma*max(sv)` of the Hessian are ignored, 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 fewer than five bisections. This value is decreased by the factor `2*LMStep` each time a search is successful without any bisections.
- `'lm'` — Levenberg-Marquardt least squares search, where the next parameter value is `-pinv(H+d*I)*grad` from the previous one. *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.
- `'grad'` — Steepest descent least squares search.
- `'lsqnonlin'` — Trust region reflective algorithm provided by the Optimization Toolbox. This method cannot be used with the `'OutputWeight'` option`'noise'`. See `lsqnonlin` for more information.

`'SearchOption'` — Option set for the search algorithm

search option set

Option set for the search algorithm, specified as the comma-separated pair consisting of `'SearchOption'` and a search option set with fields that depend on the value of `SearchMethod`.

`SearchOption` Structure When `SearchMethod` Is Specified as `'lsqnonlin'` or `'auto'`, When Optimization Toolbox Is Available

Field Name	Description	Default
`TolFun`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar. The value of `TolFun` is the same as that of `opt.SearchOption.Advanced.TolFun`.	`1e-5`
`TolX`	Termination tolerance on the estimated parameter values, specified as a positive scalar. The value of `TolX` is the same as that of `opt.SearchOption.Advanced.TolX`.	`1e-6`

Field Name	Description	Default
`MaxIter`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIter` is reached or another stopping criterion is satisfied, such as `TolFun`. The value of `MaxIter` is the same as that of `opt.SearchOption.Advanced.MaxIter`.	`20`
`Advance`	Advanced search settings, specified as an option set for `lsqnonlin`. For more information, see the Optimization Options table in "Optimization Options".	Use `optimset('lsqnonlin` to create a default option set.

`SearchOption` Structure When `SearchMethod` Is Specified as `'gn'`, `'gna'`, `'lm'`, `'grad'`, or `'auto'`, When Optimization Toolbox Is Not Available

Field Name	Description	Default
`Toleranc`	Minimum percentage difference between the current value of the loss function and its expected improvement after the next iteration, specified as a positive scalar. When the percentage of expected improvement is less than `Tolerance`, the iterations stop. The estimate of the expected loss-function improvement at the next iteration is based on the Gauss-Newton vector computed for the current parameter value.	`0.01`
`MaxIter`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIter` is reached or another stopping criterion is satisfied, such as `Tolerance`. Setting `MaxIter = 0` returns the result of the start-up procedure. Use `sys.Report.Termination.Iterations` to get the actual number of iterations during an estimation, where *sys* is an `idtf` model.	`20`
`Advanced`	Advanced search settings, specified as a structure with the following fields:	

<table>
<tr><th>Field Name</th><th>Description</th><th>Default</th></tr>
<tr><td></td><td colspan="2">
<table>
<tr><th>Field Name</th><th>Description</th><th>Default</th></tr>
<tr><td><code>GnPinvCons</code></td><td>Jacobian matrix singular value threshold, specified as a positive scalar. Singular values of the Jacobian matrix that are smaller than <code>GnPinvConst*max(size(J)*norm(J)*eps)</code> are discarded when computing the search direction. Applicable when <code>SearchMethod</code> is <code>'gn'</code>.</td><td><code>10000</code></td></tr>
<tr><td><code>InitGnaTol</code></td><td>Initial value of <i>gamma</i>, specified as a positive scalar. Applicable when <code>SearchMethod</code> is <code>'gna'</code>.</td><td><code>0.0001</code></td></tr>
<tr><td><code>LMStartVal</code></td><td>Starting value of search-direction length <i>d</i> in the Levenberg-Marquardt method, specified as a positive scalar. Applicable when <code>SearchMethod</code> is <code>'lm'</code>.</td><td><code>0.001</code></td></tr>
<tr><td><code>LMStep</code></td><td>Size of the Levenberg-Marquardt step, specified as a positive integer. The next value of the search-direction length <i>d</i> in the Levenberg-Marquardt method is <code>LMStep</code> times the previous one. Applicable when <code>SearchMethod</code> is <code>'lm'</code>.</td><td><code>2</code></td></tr>
<tr><td><code>MaxBisecti</code></td><td>Maximum number of bisections used for line search along the search direction, specified as a positive integer.</td><td><code>25</code></td></tr>
<tr><td><code>MaxFunEval</code></td><td>Maximum number of calls to the model file, specified as a positive integer. Iterations stop if the number of calls to the model file exceeds this value.</td><td><code>Inf</code></td></tr>
<tr><td><code>MinParChan</code></td><td>Smallest parameter update allowed per iteration, specified as a nonnegative scalar.</td><td><code>0</code></td></tr>
<tr><td><code>RelImprove</code></td><td>Relative improvement threshold, specified as a nonnegative scalar. Iterations stop if the relative improvement of the criterion function is less than this value.</td><td><code>0</code></td></tr>
<tr><td><code>StepReduct</code></td><td>Step reduction factor, specified as a positive scalar that is greater than 1. The suggested parameter update is reduced by the factor <code>StepReduction</code></td><td><code>2</code></td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Field Name</th><th>Description</th><th>Default</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field Name</th><th>Description</th><th>Default</th></tr>
<tr><td></td><td>after each try. This reduction continues until either <code>MaxBisections</code> tries are completed or a lower value of the criterion function is obtained.
<code>StepReduction</code> is not applicable for <code>SearchMethod</code> <code>'lm'</code> (Levenberg-Marquardt method).</td><td></td></tr>
</table>
</td><td></td></tr>
</table>

To specify field values in `SearchOption`, create a default `nlgreyestOptions` set and modify the fields using dot notation. Any fields that you do not modify retain their default values.

```
opt = nlgreyestOptions('SearchMethod','gna');
opt.SearchOption.MaxIter = 50;
opt.SearchOption.Advanced.RelImprovement = 0.5;
```

`'OutputWeight'` — Weighting of prediction error in multi-output estimations

`[]` (default) | `'noise'` | matrix

Weighting of prediction error in multi-output model estimations, specified as the comma-separated pair consisting of `'OutputWeight'` and one of the following:

- `[]` — No weighting is used. Specifying as `[]` is the same as `eye(Ny)`, where `Ny` is the number of outputs.
- `'noise'` — Optimal weighting is automatically computed as the inverse of the estimated noise variance. This weighting minimizes `det(E'*E/N)`, where `E` is the matrix of prediction errors and `N` is the number of data samples. This option is not available when using `'lsqnonlin'` as a `'SearchMethod'`.
- A positive semidefinite matrix, `W`, of size equal to the number of outputs. This weighting minimizes `trace(E'*E*W/N)`, where `E` is the matrix of prediction errors and `N` is the number of data samples.

`'Advanced'` — Additional advanced options

structure

Additional advanced options, specified as the comma-separated pair consisting of `'Advanced'` and a structure with field:

Field Name	Description	Default
`ErrorThreshold`	Threshold for when to adjust the weight of large errors from quadratic to linear, specified as a nonnegative scalar. Errors larger than `ErrorThreshold` times the estimated standard deviation have a linear weight in the loss function. The standard deviation is estimated robustly as the median of the absolute deviations from the median of the prediction errors divided by 0.7. If your estimation data contains outliers, try setting `ErrorThreshold` to `1.6`.	`0` — Leads to a purely quadratic loss function.

To specify field values in `Advanced`, create a default `nlgreyestOptions` set and modify the fields using dot notation. Any fields that you do not modify retain their default values.

```
opt = nlgreyestOptions;
opt.Advanced.ErrorThreshold = 1.2;
```

Output Arguments

`opt` — Option set for `nlgreyest`

`nlgreyestOptions` option set

Option set for `nlgreyest`, returned as an `nlgreyestOptions` option set.

See Also

`nlgreyest`

Introduced in R2015a

nlhw

Estimate Hammerstein-Wiener model

Syntax

```
sys = nlhw(Data,Orders)
sys = nlhw(Data,Orders,InputNL,OutputNL)

sys = nlhw(Data,LinModel)
sys = nlhw(Data,LinModel,InputNL,OutputNL)

sys = nlhw(Data,sys0)

sys = nlhw( ___ ,Options)
```

Description

`sys = nlhw(Data,Orders)` creates and estimates a Hammerstein-Wiener model using the estimation data, model orders and delays, and default piecewise linear functions as input and output nonlinearity estimators.

`sys = nlhw(Data,Orders,InputNL,OutputNL)` specifies `InputNL` and `OutputNL` as the input and output nonlinearity estimators, respectively.

`sys = nlhw(Data,LinModel)` uses a linear model to specify the model orders and delays, and default piecewise linear functions for the input and output nonlinearity estimators.

`sys = nlhw(Data,LinModel,InputNL,OutputNL)` specifies `InputNL` and `OutputNL` as the input and output nonlinearity estimators, respectively.

`sys = nlhw(Data,sys0)` refines or estimates the parameters of a Hammerstein-Wiener model, `sys0`, using the estimation data.

Use this syntax to:

- Update the parameters of a previously estimated model to improve the fit to the estimation data. In this case, the estimation algorithm uses the parameters of `sys0` as initial guesses.
- Estimate the parameters of a model previously created using the `idnlhw` constructor. Prior to estimation, you can configure the model properties using dot notation.

`sys = nlhw(___,Options)` specifies additional model estimation options. Use `Options` with any of the previous syntaxes.

Examples

Estimate a Hammerstein-Wiener Model

```
load iddata3
m1 = nlhw(z3,[4 2 1]);
```

Estimate a Hammerstein Model with Saturation

Load data.

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

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 Hammerstein-Wiener Model with a Custom Network Nonlinearity

Generating a custom network nonlinearity requires the definition of a user-defined unit function.

Define the unit function and save it as `gaussunit.m`.

```
% Copyright 2015 The MathWorks, Inc.

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

Create a custom network nonlinearity using the `gaussunit` function.

```
H = @gaussunit;
CNet = customnet(H);
```

Load the estimation data.

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

Estimate a Hammerstein-Wiener model using the custom network.

```
m = nlhw(z1,[5 1 3],CNet,[]);
```

Estimate Default Hammerstein-Wiener Model Using an Input-Output Polynomial Model of 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.

```
sys = nlhw(ThrottleData,LinearModel,[],'saturation');
```

Estimate a Hammerstein-Wiener Model Using `idnlhw` to first Define the Model Properties

Load the estimation data.

```
load iddata1
```

Construct a Hammerstein-Wiener model using `idnlhw` to define the model properties `B` and `F`.

```
sys0 = idnlhw([2,2,0],[],'wavenet');
sys0.B{1} = [0.8,1];
sys0.F{1} = [1,-1.2,0.5];
```

Estimate the model.

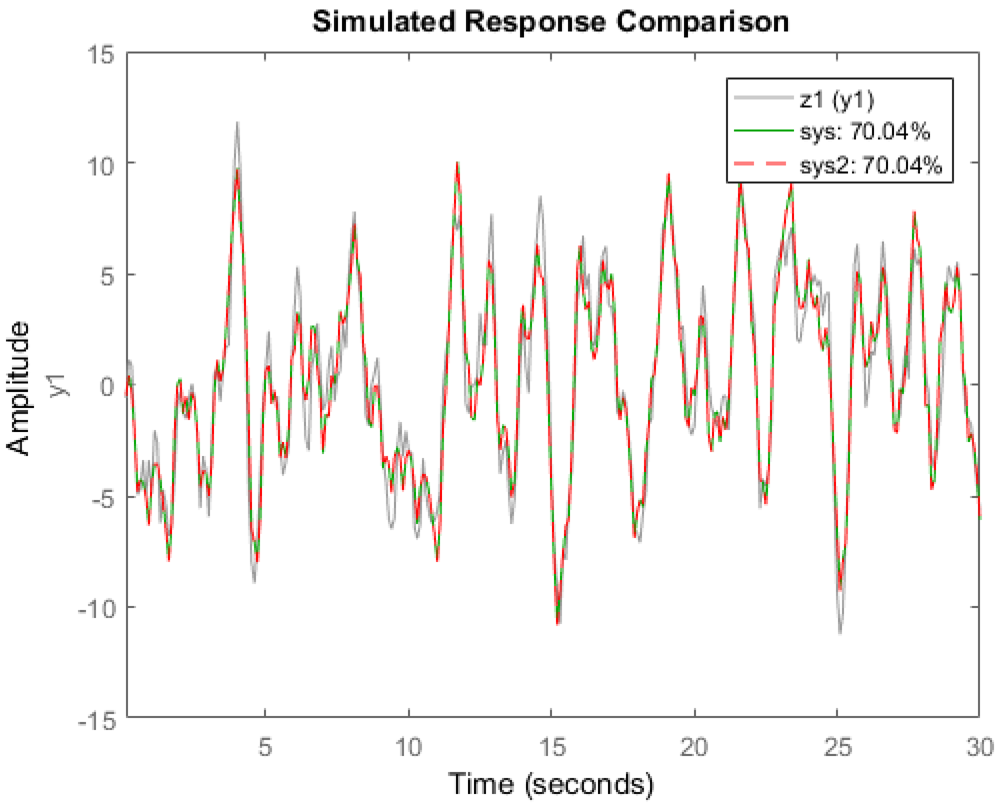
```
sys = nlhw(z1,sys0);
```

Estimate a Hammerstein-Wiener model using `nlhw` to define the model properties `B` and `F`.

```
sys2 = nlhw(z1,[2,2,0],[],'wavenet','B',{[0.8,1]},'F',{[1,-1.2,0.5]});
```

Compare the two estimated models to see that they are equivalent.

```
compare(z1,sys,'g',sys2,'r--');
```



Refine a Hammerstein-Wiener Model Using Successive Calls of `nlhw`

Estimate a Hammerstein-Wiener Model.

```
load iddata3
sys = nlhw(z3,[4 2 1],'sigmoidnet','wavenet');
```

Refine the model, `sys`.

```
sys = nlhw(z3,sys);
```

Estimate Hammerstein-Wiener Model Using an Estimation Option Set

Create estimation option set for `nlhw` to view estimation progress and to set the maximum iteration steps to 50.

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

Load data and estimate the model.

```
load iddata3
sys = nlhw(z3,[4 2 1],'sigmoidnet','deadzone',opt);
```

- “”
- “Estimate Hammerstein-Wiener Models Using Linear OE Models”

Input Arguments

`Data` — Time domain data

`iddata` object

Time-domain estimation data, specified as an `iddata`.

`Orders` — Order and delays of the linear subsystem transfer function

`[nb nf nk]` vector of positive integers | `[nb nf nk]` vector of matrices

Order and delays of the linear subsystem transfer function, specified as a `[nb nf nk]` vector.

Dimensions of `Orders`:

- For a SISO transfer function, `Orders` is a vector of positive integers.

  `nb` is the number of zeros plus 1, `nf` is the number of poles, and `nk` is the input delay.
- For a MIMO transfer function with n_u inputs and n_y outputs, `Orders` is a vector of matrices.

  `nb`, `nf`, and `nk` are n_y-by-n_u 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` — Input static nonlinearity

`'pwlinear'` (default) | `'sigmoidnet'` | `'wavenet'` | `'saturation'` | `'deadzone'` | `'poly1d'` | `'unitgain'` | nonlinearity estimator object | array of nonlinearity estimators

Input static nonlinearity estimator, specified as one of the following.

`'pwlinear'` or `pwlinear` 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 `[]` or `unitgain` object	Unit gain
`customnet` object	Custom network — Similar to `sigmoidnet`, but with a user-defined replacement for the sigmoid function.

Specifying a character vector, for example `'sigmoidnet'`, creates a nonlinearity estimator object with default settings. Use object representation instead to configure the properties of a nonlinearity estimator.

```
InputNL = wavenet;
InputNL.NumberOfUnits = 10;
```

Alternatively, use the associated input nonlinearity estimator function with Name-Value pair arguments.

```
InputNL = wavenet('NumberOfUnits',10);
```

For n_u input channels, you can specify nonlinear estimators individually for each input channel by setting `InputNL` to an n_u-by-1 array of nonlinearity estimators.

```
InputNL = [sigmoidnet('NumberofUnits',5); deadzone([-1,2])]
```

To specify the same nonlinearity for all inputs, specify a single input nonlinearity estimator.

`OutputNL` — Output static nonlinearity

`'pwlinear'` (default) | `'sigmoidnet'` | `'wavenet'` | `'saturation'` | `'deadzone'` | `'poly1d'` | `'unitgain'` | nonlinearity estimator object | array of nonlinearity estimators

Output static nonlinearity estimator, specified as one of the following:

`'pwlinear'` or `pwlinear` 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 `[]` or `unitgain` object	Unit gain
`customnet` object	Custom network — Similar to `sigmoidnet`, but with a user-defined replacement for the sigmoid function.

Specifying a character vector creates a nonlinearity estimator object with default settings. Use object representation instead to configure the properties of a nonlinearity estimator.

```
OutputNL = sigmoidnet;
OutputNL.NumberOfUnits = 10;
```

Alternatively, use the associated input nonlinearity estimator function with Name-Value pair arguments.

```
OutputNL = sigmoidnet('NumberOfUnits',10);
```

For n_y output channels, you can specify nonlinear estimators individually for each output channel by setting `OutputNL` to an n_y-by-1 array of nonlinearity estimators. To specify the same nonlinearity for all outputs, specify a single output nonlinearity estimator.

`LinModel` — Discrete time linear model

`idpoly` | `idss` with `K = 0` | `idtf`

Discrete-time linear model used to specify the linear subsystem, specified as one of the following:

- Input-output polynomial model of Output-Error (OE) structure (`idpoly`)
- State-space model with no disturbance component (`idss` with `K = 0`)
- Transfer function model (`idtf`)

Typically, you estimate the model using `oe`, `n4sid`, or `tfest`.

`sys0` — Hammerstein-Wiener model

`idnlhw` object

Hammerstein-Wiener model, specified as an `idnlhw` object. `sys0` can be:

- A model previously created using `idnlhw` to specify model properties.
- A model previously estimated using `nlhw`, that you want to update using a new estimation data set.

  You can also refine `sys0` using the original estimation data set. If the previous estimation stopped when the numerical search was stuck at a local minima of the cost function, use `init` to first randomize the parameters of `sys0`. See `sys0.Report.Termination` for search stopping conditions. Using `init` does not guarantee a better solution on further refinement.

`Options` — Estimation options

`nlhwOptions` option set

Estimation options for Hammerstein-Wiener model identification, specified as an `nlhwOptions` option set.

Output Arguments

`sys` — Estimated Hammerstein-Wiener model

`idnlhw` object

Estimated Hammerstein-Wiener model, returned as an `idnlhw` object. The model is estimated using the specified model orders and delays, input and output nonlinearity estimators, and estimation options.

Information about the estimation results and options used is stored in the `Report` property of the model. `Report` has the following fields:

Report Field	Description
`Status`	Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.
`Method`	Estimation command used.

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Fit</code></td><td>Quantitative assessment of the estimation, returned as a structure. See “Loss Function and Model Quality Metrics” for more information on these quality metrics. The structure has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>FitPerc</code></td><td>Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as a percentage.</td></tr>
<tr><td><code>LossFcn</code></td><td>Value of the loss function when the estimation completes.</td></tr>
<tr><td><code>MSE</code></td><td>Mean squared error (MSE) measure of how well the response of the model fits the estimation data.</td></tr>
<tr><td><code>FPE</code></td><td>Final prediction error for the model.</td></tr>
<tr><td><code>AIC</code></td><td>Raw Akaike Information Criteria (AIC) measure of model quality.</td></tr>
<tr><td><code>AICc</code></td><td>Small sample-size corrected AIC.</td></tr>
<tr><td><code>nAIC</code></td><td>Normalized AIC.</td></tr>
<tr><td><code>BIC</code></td><td>Bayesian Information Criteria (BIC).</td></tr>
</table></td></tr>
<tr><td><code>Parameter</code></td><td>Estimated values of model parameters.</td></tr>
<tr><td><code>OptionsUs</code></td><td>Option set used for estimation. If no custom options were configured, this is a set of default options. See <code>nlhwOptions</code> for more information.</td></tr>
<tr><td><code>RandState</code></td><td>State of the random number stream at the start of estimation. Empty, <code>[]</code>, if randomization was not used during estimation. For more information, see <code>rng</code> in the MATLAB documentation.</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation, returned as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
<tr><td><code>Type</code></td><td>Data type.</td></tr>
<tr><td><code>Length</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time.</td></tr>
<tr><td><code>InterSam</code></td><td>Input intersample behavior, returned as one of the following values:
• <code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
• <code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
• <code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.</td></tr>
<tr><td><code>InputOff</code></td><td>Offset removed from time-domain input data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
<tr><td><code>OutputOf</code></td><td>Offset removed from time-domain output data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td>`Terminati`</td><td>Termination conditions for the iterative search used for prediction error minimization. Structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>`WhyStop`</td><td>Reason for terminating the numerical search.</td></tr>
<tr><td>`Iteratio`</td><td>Number of search iterations performed by the estimation algorithm.</td></tr>
<tr><td>`FirstOrd`</td><td>∞ -norm of the gradient search vector when the search algorithm terminates.</td></tr>
<tr><td>`FcnCount`</td><td>Number of times the objective function was called.</td></tr>
<tr><td>`UpdateNo`</td><td>Norm of the gradient search vector in the last iteration. Omitted when the search method is `'lsqnonlin'`.</td></tr>
<tr><td>`LastImpr`</td><td>Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is `'lsqnonlin'`.</td></tr>
<tr><td>`Algorith`</td><td>Algorithm used by `'lsqnonlin'` search method. Omitted when other search methods are used.</td></tr>
</table>
For estimation methods that do not require numerical search optimization, the `Termination` field is omitted.</td></tr>
</table>

For more information, see “Estimation Report”.

More About

- “Identifying Hammerstein-Wiener Models”
- “Using Linear Model for Hammerstein-Wiener Estimation”
- “Loss Function and Model Quality Metrics”
- “Regularized Estimates of Model Parameters”
- “Estimation Report”

See Also

`aic` | `customnet` | `deadzone` | `fpe` | `goodnessofFit` | `idnlhw` | `idnlhw/findop` | `init` | `linapp` | `linearize` | `n4sid` | `nlhwOptions` | `oe` | `pem` | `poly1d` | `pwlinear` | `saturation` | `sigmoidnet` | `tfest` | `unitgain` | `wavenet`

Introduced in R2007a

nlhwOptions

Option set for `nlhw`

Syntax

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

Description

`opt = nlhwOptions` creates the default option set for `nlhw`. Use dot notation to customize the option set, if needed.

`opt = nlhwOptions(Name,Value)` creates an option set with options specified by one or more `Name,Value` pair arguments. The options that you do not specify retain their default value.

Examples

Estimate Hammerstein-Wiener Model Using an Estimation Option Set

Create estimation option set for `nlhw` to view estimation progress and to set the maximum iteration steps to 50.

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

Load data and estimate the model.

```
load iddata3
sys = nlhw(z3,[4 2 1],'sigmoidnet','deadzone',opt);
```

Specify an Option Set for Hammerstein-Wiener Model Estimation

Create an options set for `nlhw` where:

- Initial conditions are estimated from the estimation data.

- Subspace Gauss-Newton least squares method is used for estimation.

```
opt = nlhwOptions('InitialCondition','estimate','SearchMethod','gn');
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example: `nlhwOptions('InitialCondition','estimate')`

`'InitialCondition'` — Handling of initial conditions

`'zero'` (default) | `'estimate'`

Handling of initial conditions during estimation using `nlhw`, specified as the comma-separated pair consisting of `InitialCondition` and one of the following:

- `'zero'` — The initial conditions are set to zero.
- `'estimate'` — The initial conditions are treated as independent estimation parameters.

`'Display'` — Estimation progress display setting

`'off'` (default) | `'on'`

Estimation progress display setting, specified as the comma-separated pair consisting of `'Display'` and one of the following:

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

`'OutputWeight'` — Weighting of prediction error in multi-output estimations

`'noise'` (default) | positive semidefinite matrix

Weighting of prediction error in multi-output model estimations, specified as the comma-separated pair consisting of `'OutputWeight'` and one of the following:

- `'noise'` — Optimal weighting is automatically computed as the inverse of the estimated noise variance. This weighting minimizes `det(E'*E)`, where `E` is the matrix of prediction errors. This option is not available when using `'lsqnonlin'` as a `'SearchMethod'`.
- A positive semidefinite matrix, `W`, of size equal to the number of outputs. This weighting minimizes `trace(E'*E*W/N)`, where `E` is the matrix of prediction errors and `N` is the number of data samples.

`'Regularization'` — Options for regularized estimation of model parameters

structure

Options for regularized estimation of model parameters, specified as the comma-separated pair consisting of `'Regularization'` and a structure with fields:

Field Name	Description	Default
`Lambda`	Bias versus variance trade-off constant, specified as a nonnegative scalar.	`0` — Indicates no regularization.
`R`	Weighting matrix, specified as a vector of nonnegative scalars or a square positive semi-definite matrix. The length must be equal to the number of free parameters in the model, `np`. Use the `nparams` command to determine the number of model parameters.	`1` — Indicates a value of `eye(np)`.
`Nominal`	The nominal value towards which the free parameters are pulled during estimation, specified as one of the following: • `'zero'` — Pull parameters towards zero. • `'model'` — Pull parameters towards pre-existing values in the initial model. Use this option only when you have a well-initialized `idnlhw` model with finite parameter values.	`'zero'`

To specify field values in `Regularization`, create a default `nlhwOptions` set and modify the fields using dot notation. Any fields that you do not modify retain their default values.

```
opt = nlhwOptions;
opt.Regularization.Lambda = 1.2;
```

```
opt.Regularization.R = 0.5*eye(np);
```

Regularization is a technique for specifying model flexibility constraints, which reduce uncertainty in the estimated parameter values. For more information, see “Regularized Estimates of Model Parameters”.

`'SearchMethod'` — Numerical search method used for iterative parameter estimation

`'auto'` (default) | `'gn'` | `'gna'` | `'lm'` | `'grad'` | `'lsqnonlin'`

Numerical search method used for iterative parameter estimation, specified as the comma-separated pair consisting of `'SearchMethod'` and one of the following:

- `'auto'` — A combination of the line search algorithms, `'gn'`, `'lm'`, `'gna'`, and `'grad'` methods is tried at each iteration. The descent direction leading to the largest reduction in estimation cost is used.
- `'gn'` — Subspace Gauss-Newton least squares search. Singular values of the Jacobian matrix less than `GnPinvConst*eps*max(size(J))*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'` — Adaptive subspace Gauss-Newton search. Eigenvalues less than `gamma*max(sv)` of the Hessian are ignored, where *sv* contains the singular values of the Hessian. The Gauss-Newton direction is computed in the remaining subspace. *gamma* has the initial value `InitGnaTol` (see `Advanced` in `'SearchOption'` 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 five bisections. This value is decreased by the factor `2*LMStep` each time a search is successful without any bisections.
- `'lm'` — Levenberg-Marquardt least squares search, where the next parameter value is `-pinv(H+d*I)*grad` from the previous one. 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.
- `'grad'` — Steepest descent least squares search.
- `'lsqnonlin'` — Trust region reflective algorithm provided by Optimization Toolbox. This method cannot be used with the `'OutputWeight'` option`'noise'`. See `lsqnonlin` for more information.

`'SearchOption'` — Option set for the search algorithm

search option set

Option set for the search algorithm, specified as the comma-separated pair consisting of `'SearchOption'` and a search option set with fields that depend on the value of `SearchMethod`.

`SearchOption` Structure When `SearchMethod` Is Specified as `'gn'`, `'gna'`, `'lm'`, `'grad'`, or `'auto'`

<table>
<tr><th>Field Name</th><th>Description</th><th>Default</th></tr>
<tr><td><code>Toleran</code></td><td>Minimum percentage difference between the current value of the loss function and its expected improvement after the next iteration, specified as a positive scalar. When the percentage of expected improvement is less than <code>Tolerance</code>, the iterations stop. The estimate of the expected loss-function improvement at the next iteration is based on the Gauss-Newton vector computed for the current parameter value.</td><td><code>0.01</code></td></tr>
<tr><td><code>MaxIter</code></td><td>Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when <code>MaxIter</code> is reached or another stopping criterion is satisfied, such as <code>Tolerance</code>.
Setting <code>MaxIter = 0</code> returns the result of the start-up procedure.
Use <code>sys.Report.Termination.Iterations</code> to get the actual number of iterations during an estimation, where <i>sys</i> is an <code>idtf</code> model.</td><td><code>20</code></td></tr>
<tr><td><code>Advance</code></td><td colspan="2">Advanced search settings, specified as a structure with the following fields:
<table>
<tr><th>Field Name</th><th>Description</th><th>Default</th></tr>
<tr><td><code>GnPinvCons</code></td><td>Jacobian matrix singular value threshold, specified as a positive scalar. Singular values of the Jacobian matrix that are smaller than <code>GnPinvConst*max(size(J)*norm(J)*eps)</code> are discarded when computing the search direction. Applicable when <code>SearchMethod</code> is <code>'gn'</code>.</td><td><code>10000</code></td></tr>
<tr><td><code>InitGnaTol</code></td><td>Initial value of <i>gamma</i>, specified as a positive scalar. Applicable when <code>SearchMethod</code> is <code>'gna'</code>.</td><td><code>0.0001</code></td></tr>
</table></td></tr>
</table>

<table>
<tr><th>Field Name</th><th>Description</th><th>Default</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field Name</th><th>Description</th><th>Default</th></tr>
<tr><td><code>LMStartVal</code></td><td>Starting value of search-direction length <i>d</i> in the Levenberg-Marquardt method, specified as a positive scalar. Applicable when <code>SearchMethod</code> is <code>'lm'</code>.</td><td><code>0.001</code></td></tr>
<tr><td><code>LMStep</code></td><td>Size of the Levenberg-Marquardt step, specified as a positive integer. The next value of the search-direction length <i>d</i> in the Levenberg-Marquardt method is <code>LMStep</code> times the previous one. Applicable when <code>SearchMethod</code> is <code>'lm'</code>.</td><td><code>2</code></td></tr>
<tr><td><code>MaxBisecti</code></td><td>Maximum number of bisections used for line search along the search direction, specified as a positive integer.</td><td><code>25</code></td></tr>
<tr><td><code>MaxFunEval</code></td><td>Maximum number of calls to the model file, specified as a positive integer. Iterations stop if the number of calls to the model file exceeds this value.</td><td><code>Inf</code></td></tr>
<tr><td><code>MinParChan</code></td><td>Smallest parameter update allowed per iteration, specified as a nonnegative scalar.</td><td><code>0</code></td></tr>
<tr><td><code>RelImprove</code></td><td>Relative improvement threshold, specified as a nonnegative scalar. Iterations stop if the relative improvement of the criterion function is less than this value.</td><td><code>0</code></td></tr>
<tr><td><code>StepReduct</code></td><td>Step reduction factor, specified as a positive scalar that is greater than 1. The suggested parameter update is reduced by the factor <code>StepReduction</code> after each try. This reduction continues until <code>MaxBisections</code> tries are completed or a lower value of the criterion function is obtained.
<code>StepReduction</code> is not applicable for <code>SearchMethod</code> <code>'lm'</code> (Levenberg-Marquardt method).</td><td><code>2</code></td></tr>
</table>
</td><td></td></tr>
</table>

`SearchOption` Structure When `SearchMethod` Is Specified as `'lsqnonlin'`

Field Name	Description	Default
`TolFun`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values, specified as a positive scalar. The value of `TolFun` is the same as that of `opt.SearchOption.Advanced.TolFun`.	`1e-5`
`TolX`	Termination tolerance on the estimated parameter values, specified as a positive scalar. The value of `TolX` is the same as that of `opt.SearchOption.Advanced.TolX`.	`1e-6`
`MaxIter`	Maximum number of iterations during loss-function minimization, specified as a positive integer. The iterations stop when `MaxIter` is reached or another stopping criterion is satisfied, such as `TolFun`. The value of `MaxIter` is the same as that of `opt.SearchOption.Advanced.MaxIter`.	`20`
`Advance`	Advanced search settings, specified as an option set for `lsqnonlin`. For more information, see the Optimization Options table in “Optimization Options”.	Use `optimset('lsqnonlin')` to create a default option set.

To specify field values in `SearchOption`, create a default `nlhwOptions` set and modify the fields using dot notation. Any fields that you do not modify retain their default values.

```
opt = nlhwOptions;
opt.SearchOption.MaxIter = 50;
opt.SearchOption.Advanced.RelImprovement = 0.5;
```

`'Advanced'` — Additional advanced options

structure

Additional advanced options, specified as the comma-separated pair consisting of `'Advanced'` and a structure with fields:

Field Name	Description	Default
`ErrorThre`	Threshold for when to adjust the weight of large errors from quadratic to linear, specified as a nonnegative scalar. Errors larger than `ErrorThreshold` times the estimated standard deviation have a linear weight in the loss function. The standard deviation is estimated robustly as the median of the absolute deviations from the median of the prediction errors, divided by 0.7. If your estimation data contains outliers, try setting `ErrorThreshold` to `1.6`.	`0` — Leads to a purely quadratic loss function.
`MaxSize`	Maximum number of elements in a segment when input-output data is split into segments, specified as a positive integer.	`250000`

To specify field values in `Advanced`, create a default `nlhwOptions` set and modify the fields using dot notation. Any fields that you do not modify retain their default values.

```
opt = nlhwOptions;
opt.Advanced.ErrorThreshold = 1.2;
```

Output Arguments

`opt` — Option set for `nlhw`

`nlhwOptions` option set

Option set for `nlhw`, returned as an `nlhwOptions` option set.

See Also

`nlhw`

Introduced in R2015a

noise2meas

Noise component of model

Syntax

```
noise_model = noise2meas(sys)
noise_model = noise2meas(sys,noise)
```

Description

`noise_model = noise2meas(sys)` returns the noise component, `noise_model`, of a linear identified model, `sys`. Use `noise2meas` to convert a time-series model (no inputs) to an input/output model. The converted model can be used for linear analysis, including viewing pole/zero maps, and plotting the step response.

`noise_model = noise2meas(sys,noise)` specifies the noise variance normalization method.

Input Arguments

`sys`

Identified linear model.

`noise`

Noise variance normalization method, specified as one of the following values:

- `'innovations'` — Noise sources are not normalized and remain as the innovations process.
- `'normalize'` — Noise sources are normalized to be independent and of unit variance.

Default: `'innovations'`

Output Arguments

`noise_model`

Noise component of `sys`.

`sys` represents the system

$$y(t) = Gu(t) + He(t)$$

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

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) + Ke(t) \\ y(t) &= Cx(t) + Du(t) + e(t) \\ e(t) &= Lv(t) \end{aligned}$$

$v(t)$ 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'`, then `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'`, then `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 `tfdata`.

Examples

Convert Noise Component of Linear Identified Model into Input/Output Model

Convert a time-series model 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 to an input/output model, and normalize its noise variance.

Identify a linear model using data.

```
load twotankdata;
z = 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.

View the disturbance matrix.

```
sys.K
```

```
ans =

    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.

View the B matrix for `noise_model_normalize`

```
noise_model_normalize.B
```

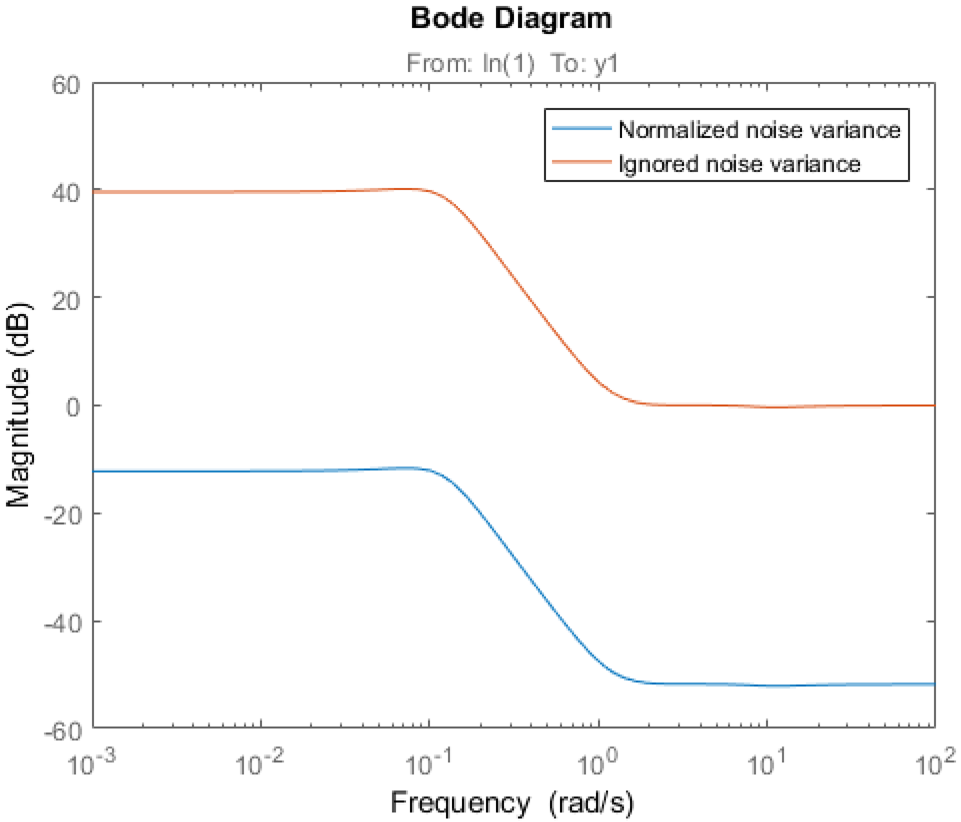
```
ans =
```

```
    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 51 dB. As expected, the magnitude difference is approximately equal to `20*log10(L)`.

See Also

`idssdata` | `noisecnv` | `spectrum` | `tfdata` | `zpkdata`

Introduced in R2012a

noisecnv

Transform identified linear model with noise channels to model with measured channels only

Syntax

```
mod1 = noisecnv(mod)
mod2 = noisecnv(mod,'normalize')
```

Description

`mod1 = noisecnv(mod)` and `mod2 = noisecnv(mod,'normalize')` transform an identified linear model with noise channels to a model with measured channels only.

`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 Λ:

$$y = Gu + He$$
$$\text{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `mod.NoiseVariance` = Λ. The model can also be described with unit variance, 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(:,[])` 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 identified linear models 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

`idssdata` | `noise2meas` | `tfdata` | `zpkdata`

Introduced before R2006a

norm

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_∞ 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_∞ norm to `tol`.

This command requires Control System Toolbox license.

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_∞ norm.

Default: 0.01

Output Arguments

`n`

H_2 norm or H_∞ 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.

Examples

This example uses `norm` to compute the H_2 and H_∞ 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_∞ infinity norm, enter:

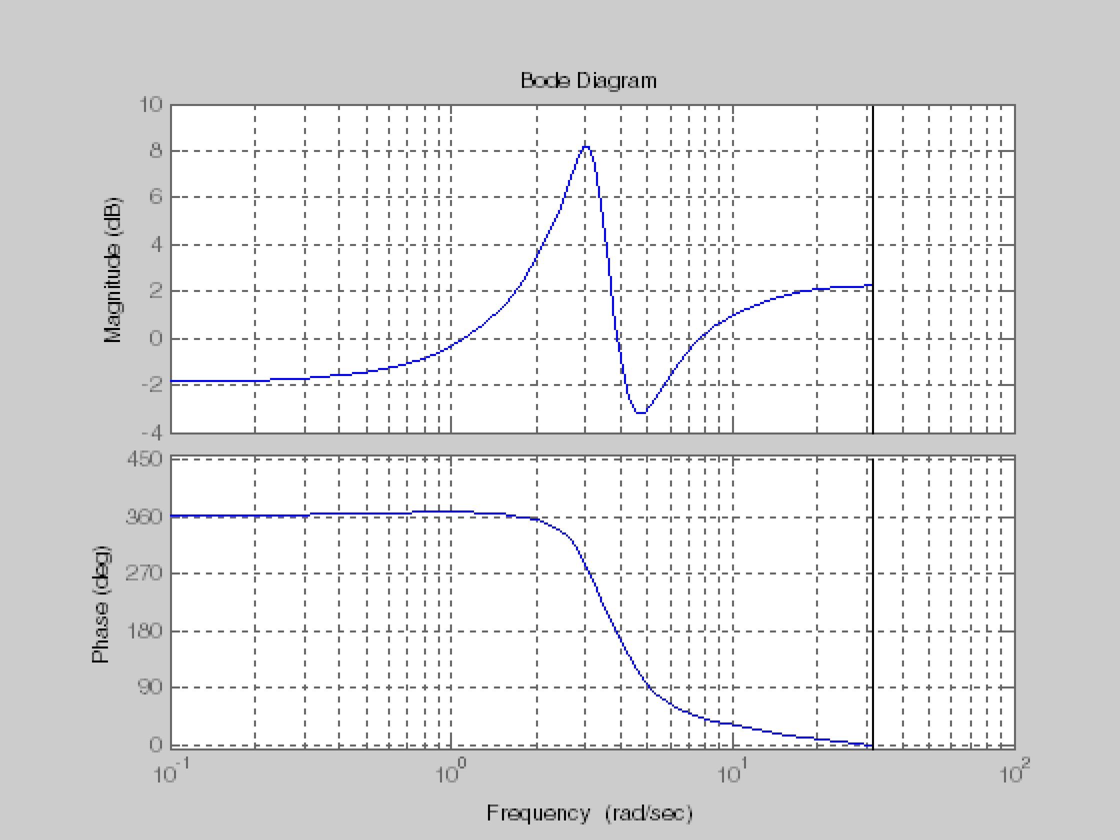
```
[ninf,fpeak] = norm(H,inf)
```

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
```

More About

H2 norm

The H_2 norm of a stable continuous-time system with transfer function $H(s)$, is given by:

$$\|H\|_2 = \sqrt{\frac{1}{2\pi}\int_{-\infty}^{\infty}\text{Trace}\left[H(j\omega)^H H(j\omega)\right]d\omega}.$$

For a discrete-time system with transfer function $H(z)$, the H_2 norm is given by:

$$\|H\|_2 = \sqrt{\frac{1}{2\pi}\int_{-\pi}^{\pi}\text{Trace}\left[H(e^{j\omega})^H H(e^{j\omega})\right]d\omega}.$$

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:

$$\|H\|_2^2 = \lim_{t\to\infty} E\left\{y(t)^T y(t)\right\}, \qquad E\left(w(t)w(\tau)^T\right) = \delta(t-\tau)I.$$

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_∞ norm (also called the L_∞ norm) of a SISO linear system is the peak gain of the frequency response. For a MIMO system, the H_∞ norm is the peak gain across all input/output channels. Thus, for a continuous-time system $H(s)$, the H_∞ norm is given by:

$$\left\|H(s)\right\|_\infty = \max_{\omega} \left|H(j\omega)\right| \qquad \text{(SISO)}$$

$$\left\|H(s)\right\|_\infty = \max_{\omega} \sigma_{\max}\left(H(j\omega)\right) \qquad \text{(MIMO)}$$

where $\sigma_{\max}(\cdot)$ denotes the largest singular value of a matrix.

For a discrete-time system $H(z)$:

$$\left\|H(z)\right\|_\infty = \max_{\theta\in[0,\pi]} \left|H\left(e^{j\theta}\right)\right| \qquad \text{(SISO)}$$

$$\left\|H(z)\right\|_\infty = \max_{\theta\in[0,\pi]} \sigma_{\max}\left(H\left(e^{j\theta}\right)\right) \qquad \text{(MIMO)}$$

The H_∞ norm is infinite if `sys` has poles on the imaginary axis (in continuous time), or on the unit circle (in discrete time).

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_∞ norm, `norm` uses the algorithm of [1]. `norm` computes the H_∞ 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_∞-Norm of a Transfer Function Matrix," *System Control Letters*, 14 (1990), pp. 287-293.

See Also

`freqresp` | `sigma`

Introduced before R2006a

nparams

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.Denominator.Free(3) = false;
np = nparams(sys,'free');
```

See Also

`idfrd` | `idgrey` | `idpoly` | `idproc` | `idss` | `idtf` | `size`

Introduced in R2012a

nuderst

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.

Introduced before R2006a

nyquist

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')`.

`[re,im,w] = nyquist(sys)` and `[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`.

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 ω_k = w(k).

$$\begin{aligned} \mathrm{re}(1,1,k) &= \mathrm{Re}\big(h(j\omega_k)\big) \\ \mathrm{im}(1,1,k) &= \mathrm{Im}\big(h(jw_k)\big) \end{aligned}$$

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,

$$\begin{aligned} \mathrm{re}(\mathrm{i},\mathrm{j},\mathrm{k}) &= \mathrm{Re}\big(h_{ij}(j\omega_k)\big) \\ \mathrm{im}(\mathrm{i},\mathrm{j},\mathrm{k}) &= \mathrm{Im}\big(h_{ij}(j\omega_k)\big) \end{aligned}$$

where h_{ij} is the transfer function from input j to output i.

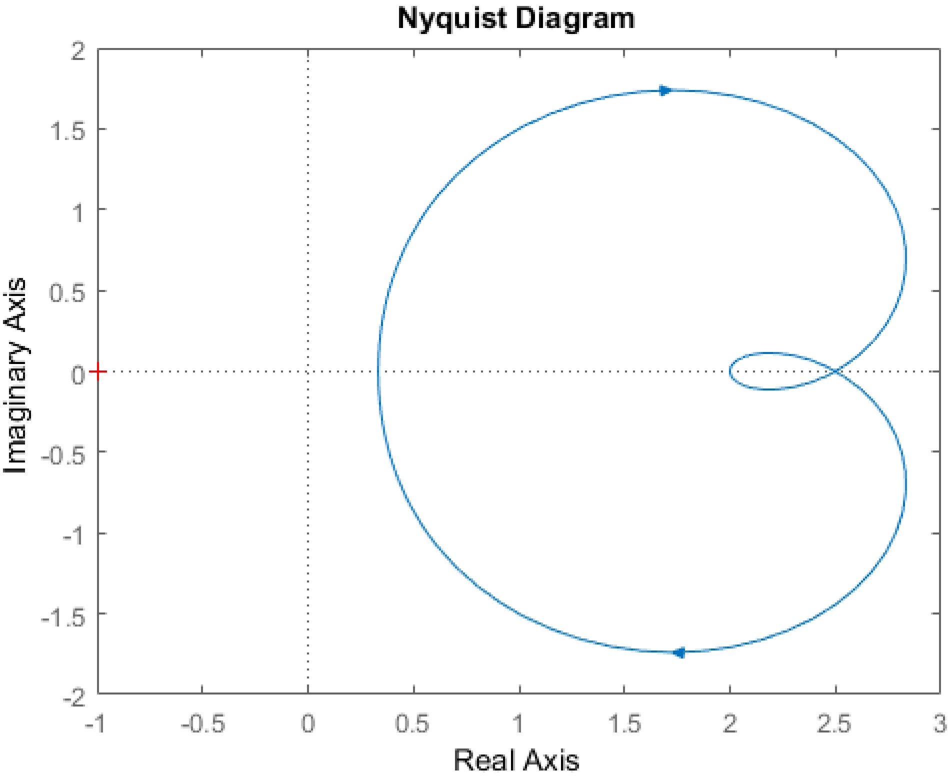
Examples

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)
```



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

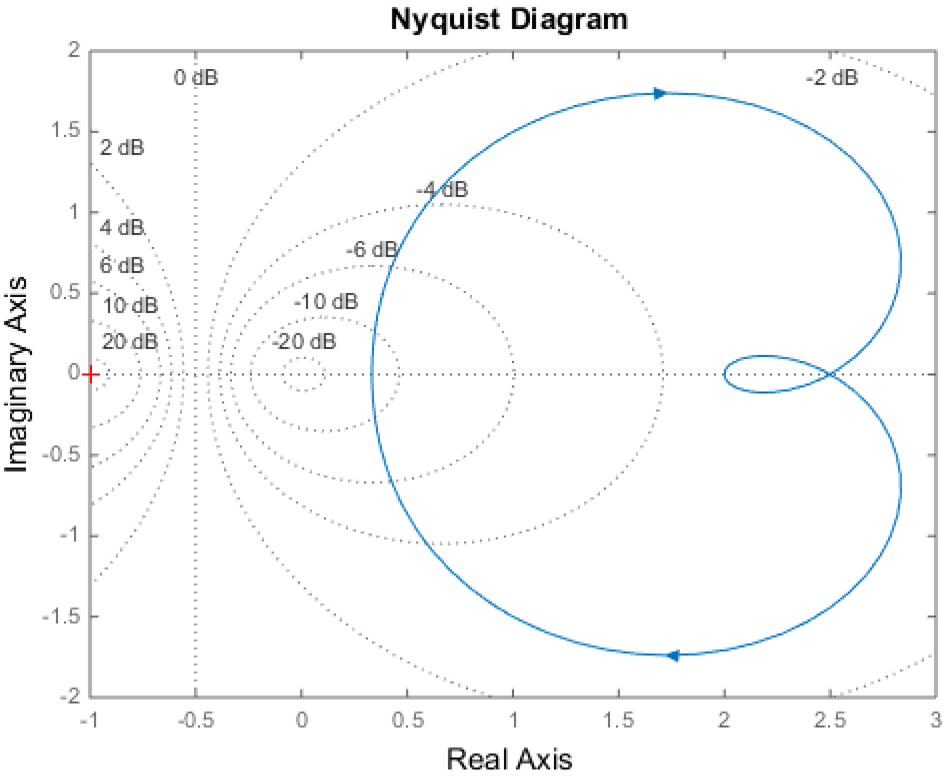
$$T(j\omega) = \left| \frac{G(j\omega)}{1 + G(j\omega)} \right|$$

is a constant value. In this equation, ω 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.

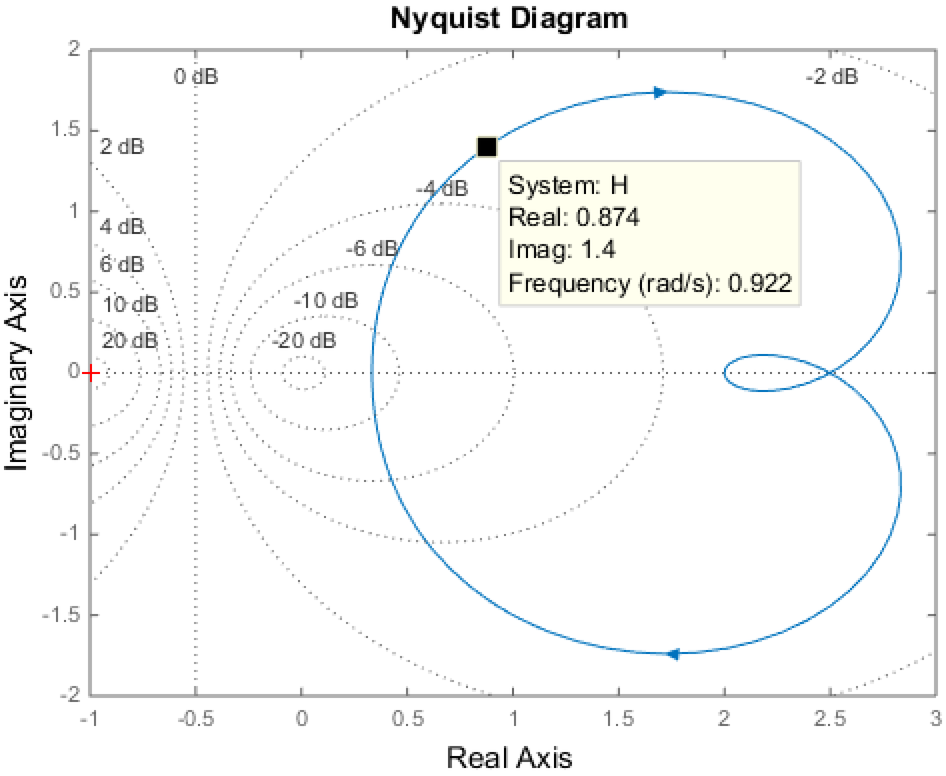


You have two zoom options available from the right-click menu that apply specifically to Nyquist plots:

- **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.



Nyquist Plot of Identified Model with Response Uncertainty

Compute the standard deviation of the real and imaginary parts of frequency response of an identified model. Use this data to create a 3σ plot of the response uncertainty. (Identified models require System Identification Toolbox.)

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 3σ 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:')
```

More About

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”.

Algorithms

See `bode`.

See Also

`bode` | `evalfr` | `freqresp` | Linear System Analyzer | `nichols` | `sigma`

Introduced before R2006a

nyquistoptions

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 and System Identification 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 values: `'off'` | `'on'` **Default**: `'off'`
GridColor	Color of the grid lines Specified as one of the following: Vector of RGB values in the range `[0,1]` | character vector of color name | `'none'`. For example, for yellow color, specify as one of the following: `[1 1 0]`, `'yellow'`, or `'y'`. **Default**: `[0.15,0.15,0.15]`
XlimMode, YlimMode	Limit modes
Xlim, Ylim	Axes limits
IOGrouping	Grouping of input-output pairs

Option	Description
	Specified as one of the following values: `'none'` `|'inputs'|'outputs'|'all'` **Default**: `'none'`
InputLabels, OutputLabels	Input and output label styles
InputVisible, OutputVisible	Visibility of input and output channels

Option	Description
FreqUnits	Frequency units, specified as one of the following values: • `'Hz'` • `'rad/second'` • `'rpm'` • `'kHz'` • `'MHz'` • `'GHz'` • `'rad/nanosecond'` • `'rad/microsecond'` • `'rad/millisecond'` • `'rad/minute'` • `'rad/hour'` • `'rad/day'` • `'rad/week'` • `'rad/month'` • `'rad/year'` • `'cycles/nanosecond'` • `'cycles/microsecond'` • `'cycles/millisecond'` • `'cycles/hour'` • `'cycles/day'` • `'cycles/week'` • `'cycles/month'` • `'cycles/year'` **Default**: `'rad/s'` You can also specify `'auto'` which uses frequency units `rad/TimeUnit` relative to system time units specified in the `TimeUnit` property. For

Option	Description
	multiple systems with different time units, the units of the first system are used.
MagUnits	Magnitude units Specified as one of the following values: `'dB'` | `'abs'` **Default**: `'dB'`
PhaseUnits	Phase units Specified as one of the following values: `'deg'` | `'rad'` **Default**: `'deg'`
ShowFullContour	Show response for negative frequencies Specified as one of the following values: `'on'` | `'off'` **Default**: `'on'`
ConfidenceRegionNumbe	Number of standard deviations to use to plotting the response confidence region (identified models only). **Default**: `1`.
ConfidenceRegionDisplay	The frequency spacing of confidence ellipses. For identified models only. **Default**: `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` | `showConfidence` | `setoptions`

Introduced in R2012a

nyquistplot

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.

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','off');
```

Right-click to turn on the confidence region characteristic by using the **Characteristics-> Confidence Region**.

More About

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”.

See Also

`showConfidence` | `getoptions` | `nyquist` | `setoptions`

Introduced in R2012a

oe

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:

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

$y(t)$ 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 linear system `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.

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 Ny-by-Nu matrix. Ny is the number of outputs and Nu is the number of inputs.
- `nf` — Order of the F polynomial. `nf` is an Ny-by-Nu matrix. Ny is the number of outputs and Nu is the number of inputs.
- `nk` — Input delay, expressed as the number of samples. `nk` is an Ny-by-Nu matrix. Ny is the number of outputs and Nu 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]` and omit `nk`.

`init_sys`

Linear system that configures the initial parameterization of `sys`.

You obtain `init_sys` by either performing an estimation using measured data or by direct construction.

If `init_sys` is an `idpoly` model of Output-Error structure, `oe` uses the parameter values of `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*) and *F*(*q*). For example:

- To specify an initial guess for the *F*(*q*) term of `init_sys`, set `init_sys.Structure.F.Value` as the initial guess.
- To specify constraints for 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 `init_sys` is not a polynomial model of Output-Error structure, the software first converts `init_sys` to an Output-Error structure model. `oe` uses the parameters of the resulting model as the initial guess for estimating `sys`.

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 option set, created using `oeOptions`, that specifies estimation options including:

- Estimation objective
- Handling of initial conditions
- Numerical search method and the associated options

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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 delay for each input channel, specified as a scalar value or numeric vector. 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 sample time `Ts`. For example, `InputDelay = 3` means a delay of three sample times.

For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector. Each entry of this vector is a numerical value that represents 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

`'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 sample time `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. Each entry of this array is a numerical value that represents 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`

Output-Error polynomial model that fits the estimation data, returned as a `idpoly` model object. This model is created using the specified model orders, delays, and estimation options.

Information about the estimation results and options used is stored in the `Report` property of the model. `Report` has the following fields:

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Status</code></td><td>Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.</td></tr>
<tr><td><code>Method</code></td><td>Estimation command used.</td></tr>
<tr><td><code>InitialCo</code></td><td>Handling of initial conditions during model estimation, returned as one of the following values:

• <code>'zero'</code> — The initial conditions were set to zero.
• <code>'estimate'</code> — The initial conditions were treated as independent estimation parameters.
• <code>'backcast'</code> — The initial conditions were estimated using the best least squares fit.

This field is especially useful to view how the initial conditions were handled when the <code>InitialCondition</code> option in the estimation option set is <code>'auto'</code>.</td></tr>
<tr><td><code>Fit</code></td><td>Quantitative assessment of the estimation, returned as a structure. See “Loss Function and Model Quality Metrics” for more information on these quality metrics. The structure has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>FitPerc</code></td><td>Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as a percentage.</td></tr>
<tr><td><code>LossFcn</code></td><td>Value of the loss function when the estimation completes.</td></tr>
<tr><td><code>MSE</code></td><td>Mean squared error (MSE) measure of how well the response of the model fits the estimation data.</td></tr>
<tr><td><code>FPE</code></td><td>Final prediction error for the model.</td></tr>
<tr><td><code>AIC</code></td><td>Raw Akaike Information Criteria (AIC) measure of model quality.</td></tr>
<tr><td><code>AICc</code></td><td>Small sample-size corrected AIC.</td></tr>
<tr><td><code>nAIC</code></td><td>Normalized AIC.</td></tr>
<tr><td><code>BIC</code></td><td>Bayesian Information Criteria (BIC).</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Parameter</code></td><td>Estimated values of model parameters.</td></tr>
<tr><td><code>OptionsUs</code></td><td>Option set used for estimation. If no custom options were configured, this is a set of default options. See <code>oeOptions</code> for more information.</td></tr>
<tr><td><code>RandState</code></td><td>State of the random number stream at the start of estimation. Empty, <code>[]</code>, if randomization was not used during estimation. For more information, see <code>rng</code> in the MATLAB documentation.</td></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation, returned as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
<tr><td><code>Type</code></td><td>Data type.</td></tr>
<tr><td><code>Length</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time.</td></tr>
<tr><td><code>InterSam</code></td><td>Input intersample behavior, returned as one of the following values:
• <code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
• <code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
• <code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.</td></tr>
<tr><td><code>InputOff</code></td><td>Offset removed from time-domain input data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
<tr><td><code>OutputOf</code></td><td>Offset removed from time-domain output data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Terminati</code></td><td>Termination conditions for the iterative search used for prediction error minimization. Structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>WhyStop</code></td><td>Reason for terminating the numerical search.</td></tr>
<tr><td><code>Iteratio</code></td><td>Number of search iterations performed by the estimation algorithm.</td></tr>
<tr><td><code>FirstOrd</code></td><td>∞ -norm of the gradient search vector when the search algorithm terminates.</td></tr>
<tr><td><code>FcnCount</code></td><td>Number of times the objective function was called.</td></tr>
<tr><td><code>UpdateNo</code></td><td>Norm of the gradient search vector in the last iteration. Omitted when the search method is <code>'lsqnonlin'</code>.</td></tr>
<tr><td><code>LastImpr</code></td><td>Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is <code>'lsqnonlin'</code>.</td></tr>
<tr><td><code>Algorith</code></td><td>Algorithm used by <code>'lsqnonlin'</code> search method. Omitted when other search methods are used.</td></tr>
</table>
For estimation methods that do not require numerical search optimization, the <code>Termination</code> field is omitted.</td></tr>
</table>

For more information on using `Report`, see “Estimation Report”.

Examples

Estimate Continuous-Time Model Using Frequency Response

Obtain the estimation data.

```
filename = fullfile(matlabroot,'examples','ident','oe_data1.mat');
load(filename);
```

`data`, an `idfrd` object, contains the continuous-time frequency response for the following model:

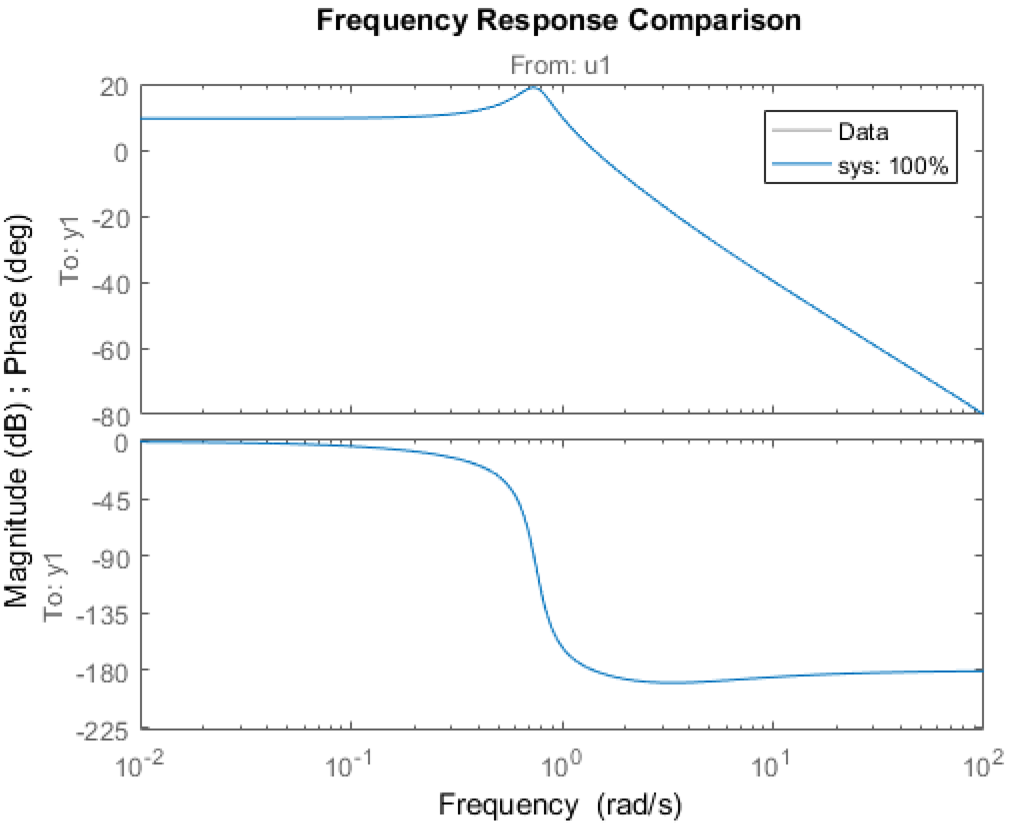
$$G(s) = \frac{s+3}{s^3 + 2s^2 + s + 1}$$

Estimate the model.

```
nb = 2;
nk = 3;
sys = oe(data,[nb nk]);
```

Evaluate the goodness of the fit.

```
compare(data,sys);
```



Estimate Output-Error Model Using Regularization

Estimate a high-order OE model from data collected by simulating a high-order system. Determine the regularization constants by trial and error and use the values for model estimation.

Load data.

```
load regularizationExampleData.mat m0simdata
```

Estimate an unregularized OE model of order 30.

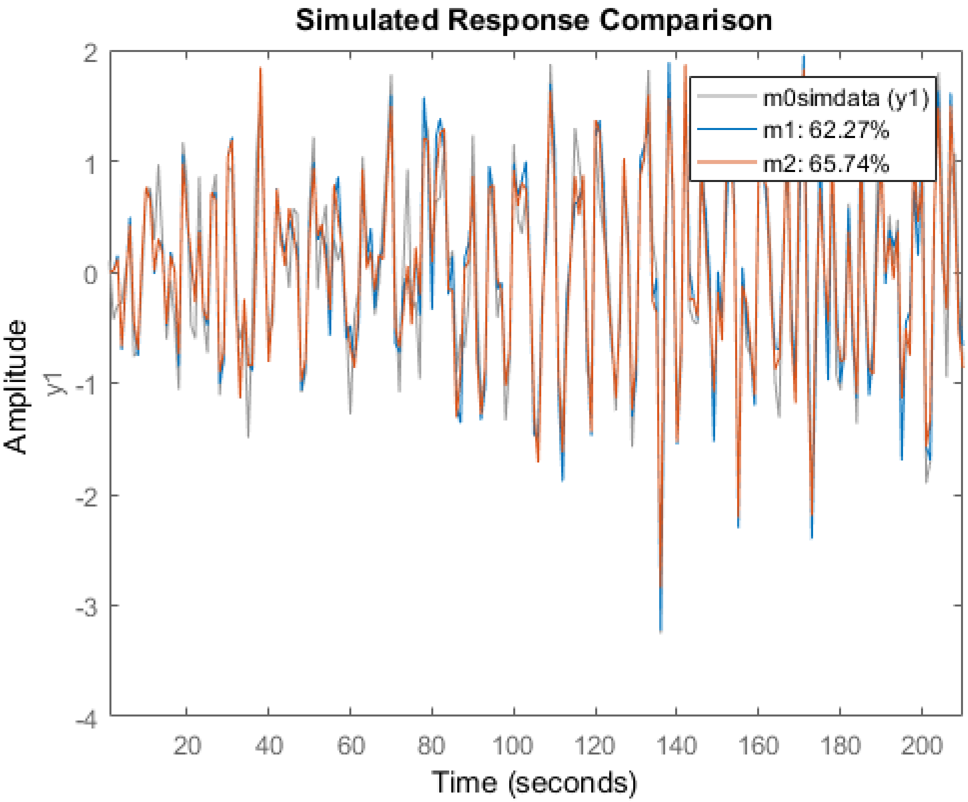
```
m1 = oe(m0simdata,[30 30 1]);
```

Obtain a regularized OE model by determining Lambda value using trial and error.

```
opt = oeOptions;
opt.Regularization.Lambda = 1;
m2 = oe(m0simdata,[30 30 1],opt);
```

Compare the model outputs with the estimation data.

```
opt = compareOptions('InitialCondition','z');
compare(m0simdata,m1,m2,opt);
```

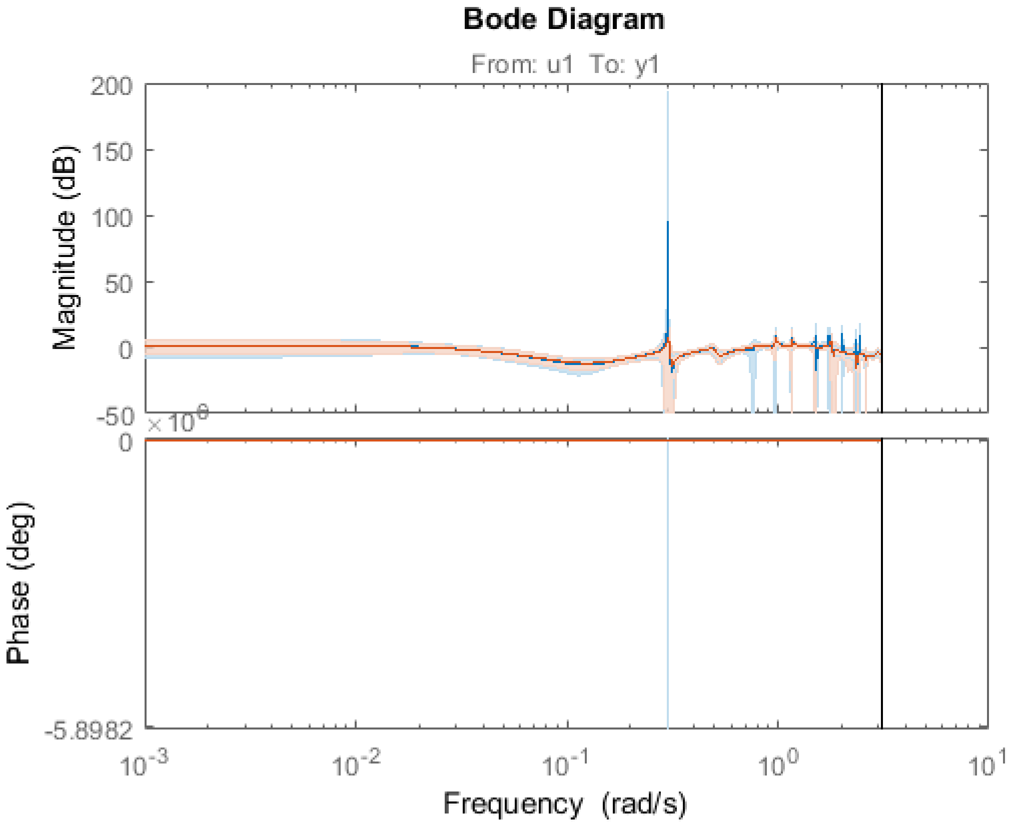


The regularized model, `m2`, produces a better fit than the unregularized model, `m1`.

Compare the variance in the model responses.

```
h = bodeplot(m1,m2);
opt = getoptions(h);
```

```
opt.PhaseMatching = 'on';
opt.ConfidenceRegionNumberSD = 3;
opt.PhaseMatching = 'on';
setoptions(h,opt);
showConfidence(h);
```



The variance of the regularized model `m2` is reduced compared to the unregularized model `m1`.

Estimate Model Using Band-Limited Discrete-Time Frequency-Domain Data

Obtain the estimation data.

```
filename = fullfile(matlabroot,'examples','ident','oe_data2.mat');
```

```
load(filename,'data','Ts');
```

`data`, an `iddata` object, contains the discrete-time frequency response for the following model:

$$G(s) = \frac{1000}{s + 500}$$

The sample time for `data`, `Ts`, is 0.001 seconds.

Treat `data` as continuous-time data. When you plot `data`, the input/output signals are band-limited, which allows you to treat `data` as continuous-time data. You can now obtain a continuous-time model.

```
data.Ts = 0;
```

Specify the estimation options.

```
opt = oeOptions('WeightingFilter',[0 0.5*pi/Ts]);
```

This prefilter choice directs the software to ignore the response values for frequencies higher than `0.5*pi/Ts` rad/s.

Estimate the model.

```
nb = 1;
nf = 3;
sys = oe(data,[nb nf],opt);
```

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.

More About

Output-Error (OE) Model

The general Output-Error model structure is:

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

The orders of the Output-Error model are:

$$\begin{aligned} nb&: \; B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1} \\ nf&: \; F(q) = 1 + f_1 q^{-1} + \ldots + f_{nf} q^{-nf} \end{aligned}$$

Continuous-Time, Output-Error Model

If `data` is continuous-time frequency-domain data, `oe` estimates a continuous-time model with transfer function:

$$G(s) = \frac{B(s)}{F(s)} = \frac{b_{nb} s^{(nb-1)} + b_{nb-1} s^{(nb-2)} + \ldots + b_1}{s^{nf} + f_{nf} s^{(nf-1)} + \ldots + f_1}$$

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. Use `model = oe(data,[nb nf])`. Use the `IODelay` model property to specify any input-output delays. For example, use `model = oe(data, [nb nf], 'IODelay',iod)` instead.

Tips

- To estimate a continuous-time model when `data` represents continuous-time frequency response data, omit `nk`.

  For example, use `sys = oe(data,[nb nf])`.

Algorithms

The estimation algorithm minimizes prediction errors.

- "What Are Polynomial Models?"
- "Regularized Estimates of Model Parameters"

See Also

`armax` | `arx` | `bj` | `compare` | `iddata` | `idfrd` | `idpoly` | `iv4` | `n4sid` | `oeOptions` | `polyest` | `sim` | `tfest`

Introduced before R2006a

oeOptions

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. `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'` — Handling of initial conditions
`'auto'` (default) | `'zero'` | `'estimate'` | `'backcast'`

Handling of initial conditions during estimation, specified as 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.

`'WeightingFilter'` — Weighting prefilter

[ ] (default) | vector | matrix | cell array | linear system

Weighting prefilter applied to the loss function to be minimized during estimation. To understand the effect of `WeightingFilter` on the loss function, see "Loss Function and Model Quality Metrics".

Specify `WeightingFilter` as one of the following values:

- `[]` — No weighting prefilter is used.
- Passbands — Specify a row vector or matrix containing frequency values that define desired passbands. You select a frequency band where the fit between estimated model and estimation data is optimized. For example, `[wl,wh]` where `wl` and `wh` represent lower and upper limits of a passband. For a matrix with several rows defining frequency passbands, `[w1l,w1h;w2l,w2h;w3l,w3h;...]`, the estimation algorithm uses the union of the frequency ranges to define the estimation passband.

  Passbands are expressed in `rad/TimeUnit` for time-domain data and in `FrequencyUnit` for frequency-domain data, where `TimeUnit` and `FrequencyUnit` are the time and frequency units of the estimation data.
- SISO filter — Specify a single-input-single-output (SISO) linear filter in one of the following ways:
  - A SISO LTI model
  - `{A,B,C,D}` format, which specifies the state-space matrices of a filter with the same sample time as estimation data.
  - `{numerator,denominator}` format, which specifies the numerator and denominator of the filter as a transfer function with same sample time as estimation data.

    This option calculates the weighting function as a product of the filter and the input spectrum to estimate the transfer function.
- Weighting vector — Applicable for frequency-domain data only. Specify a column vector of weights. This vector must have the same length as 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.

`'EnforceStability'` — Control whether to enforce stability of model

`false` (default) | `true`

Control whether to enforce stability of estimated model, specified as the comma-separated pair consisting of `'EnforceStability'` and either `true` or `false`.

Use this option when estimating models using frequency-domain data. Models estimated using time-domain data are always stable.

Data Types: `logical`

`'EstCovar'` — Control whether to generate parameter covariance data

`true` (default) | `false`

Controls whether parameter covariance data is generated, specified as `true` or `false`.

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

`'Display'` — Specify whether to display the estimation progress

`'off'` (default) | `'on'`

Specify whether to display the estimation progress, specified as one of the following values:

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

`'InputOffset'` — Removal of offset from time-domain input data during estimation

[ ] (default) | vector of positive integers | matrix

Removal of offset from time-domain input data during estimation, specified as the comma-separated pair consisting of `'InputOffset'` and one of the following:

- A column vector of positive integers of length *Nu*, where *Nu* is the number of inputs.
- `[]` — Indicates no offset.
- *Nu*-by-*Ne* matrix — For multi-experiment data, specify `InputOffset` as an *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.

`'OutputOffset'` — Removal of offset from time-domain output data during estimation

[ ] (default) | vector | matrix

Removal of offset from time-domain output data during estimation, specified as the comma-separated pair consisting of `'OutputOffset'` and one of the following:

- A column vector of length *Ny*, where *Ny* is the number of outputs.
- `[]` — Indicates no offset.
- *Ny*-by-*Ne* matrix — For multi-experiment 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.

`'Regularization'` — Options for regularized estimation of model parameters

structure

Options for regularized estimation of model parameters. For more information on regularization, see "Regularized Estimates of Model Parameters".

`Regularization` is a structure with the following fields:

- `Lambda` — Constant that determines the bias versus variance tradeoff.

  Specify a positive scalar to add the regularization term to the estimation cost.

  The default value of zero implies no regularization.

  Default: 0
- `R` — Weighting matrix.

  Specify a vector of nonnegative numbers or a square positive semi-definite matrix. The length must be equal to the number of free parameters of the model.

  For black-box models, using the default value is recommended. For structured and grey-box models, you can also specify a vector of `np` positive numbers such that each entry denotes the confidence in the value of the associated parameter.

  The default value of 1 implies a value of `eye(npfree)`, where `npfree` is the number of free parameters.

  Default: 1
- `Nominal` — The nominal value towards which the free parameters are pulled during estimation.

  The default value of zero implies that the parameter values are pulled towards zero. If you are refining a model, you can set the value to `'model'` to pull the parameters towards the parameter values of the initial model. The initial parameter values must be finite for this setting to work.

Default: 0

`'SearchMethod'` — Search method used for iterative parameter estimation

`'auto'` (default) | `'gn'` | `'gna'` | `'lm'` | `'lsqnonlin'` | `'grad'`

Search method used for iterative parameter estimation, specified as one of the following values:

- `'gn'` — The subspace Gauss-Newton direction. Singular values of the Jacobian matrix less than `GnPinvConst*eps*max(size(J))*norm(J)` are discarded when computing the search direction. *J* is the Jacobian matrix. The Hessian matrix is approximated by J^TJ. 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 `gamma*max(sv)` of the Hessian are ignored, 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*LMStep` each time a search is successful without any bisections.
- `'lm'` — Uses the Levenberg-Marquardt method so that the next parameter value is `-pinv(H+d*I)*grad` from the previous one. *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 handle only the Trace criterion.
- `'grad'` — The steepest descent gradient search method.
- `'auto'` — The algorithm chooses one of the preceding options. The descent direction is calculated using `'gn'`, `'gna'`, `'lm'`, and `'grad'` successively at each iteration. The iterations continue until a sufficient reduction in error is achieved.

`'SearchOption'` — Option set for the search algorithm

search option set

Option set for the search algorithm with fields that depend on the value of `SearchMethod`.

SearchOption structure when SearchMethod is specified as 'gn', 'gna', 'lm', 'grad', or 'auto'

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td><code>Toleran</code></td><td>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 <code>Tolerance</code>, the iterations stop. The estimate of the expected loss-function improvement at the next iteration is based on the Gauss-Newton vector computed for the current parameter value.

Default: <code>0.01</code></td></tr>
<tr><td><code>MaxIter</code></td><td>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>Tolerance</code>.

Setting <code>MaxIter = 0</code> returns the result of the start-up procedure.

Use <code>sys.Report.Termination.Iterations</code> to get the actual number of iterations during an estimation, where <i>sys</i> is an <code>idtf</code> model.

Default: <code>20</code></td></tr>
<tr><td><code>Advance</code></td><td>Advanced search settings.

Specified as a structure with the following fields:
<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td><code>GnPinvCon</code></td><td>Singular values of the Jacobian matrix that are smaller than <code>GnPinvConst*max(size(J)*norm(J)*eps)</code> are discarded when computing the search direction. Applicable when <code>SearchMethod</code> is <code>'gn'</code>.

<code>GnPinvConst</code> must be a positive, real value.

Default: <code>10000</code></td></tr>
<tr><td><code>InitGnaTo</code></td><td>Initial value of <i>gamma</i>. Applicable when <code>SearchMethod</code> is <code>'gna'</code>.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>Default: <code>0.0001</code></td></tr>
<tr><td><code>LMStartVa</code></td><td>Starting value of search-direction length <i>d</i> in the Levenberg-Marquardt method. Applicable when <code>SearchMethod</code> is <code>'lm'</code>.
Default: <code>0.001</code></td></tr>
<tr><td><code>LMStep</code></td><td>Size of the Levenberg-Marquardt step. The next value of the search-direction length <i>d</i> in the Levenberg-Marquardt method is <code>LMStep</code> times the previous one. Applicable when <code>SearchMethod</code> is <code>'lm'</code>.
Default: <code>2</code></td></tr>
<tr><td><code>MaxBisect</code></td><td>Maximum number of bisections used by the line search along the search direction.
Default: <code>25</code></td></tr>
<tr><td><code>MaxFunEva</code></td><td>Iterations stop if the number of calls to the model file exceeds this value.
<code>MaxFunEvals</code> must be a positive, integer value.
Default: <code>Inf</code></td></tr>
<tr><td><code>MinParCha</code></td><td>Smallest parameter update allowed per iteration.
<code>MinParChange</code> must be a positive, real value.
Default: <code>0</code></td></tr>
<tr><td><code>RelImprov</code></td><td>Iterations stop if the relative improvement of the criterion function is less than <code>RelImprovement</code>.
<code>RelImprovement</code> must be a positive, integer value.
Default: <code>0</code></td></tr>
<tr><td><code>StepReduc</code></td><td>Suggested parameter update is reduced by the factor <code>StepReduction</code> after each try. This reduction continues until</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>either <code>MaxBisections</code> tries are completed or a lower value of the criterion function is obtained.

<code>StepReduction</code> must be a positive, real value that is greater than 1.

Default: 2</td></tr>
</table>
</td></tr>
</table>

SearchOption structure when SearchMethod is specified as 'lsqnonlin'

Field Name	Description
`TolFun`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values. The value of `TolFun` is the same as that of `opt.SearchOption.Advanced.TolFun`. **Default:** 1e-5
`TolX`	Termination tolerance on the estimated parameter values. The value of `TolX` is the same as that of `opt.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 `opt.SearchOption.Advanced.MaxIter`. **Default:** 20
`Advance`	Options set for `lsqnonlin`.

Field Name	Description
	For more information, see the Optimization Options table in “Optimization Options”. Use `optimset('lsqnonlin')` to create an options set for `lsqnonlin`, and then modify it to specify its various options.

`'Advanced'` — Additional advanced options

structure

Additional advanced options, specified as 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 loss function. The standard deviation is estimated robustly as the median of the absolute deviations from the median of the prediction errors, divided by `0.7`. For more information on robust norm choices, see section 15.2 of [2].

  `ErrorThreshold = 0` disables robustification and leads to a purely quadratic loss function. When estimating with frequency-domain data, the software sets `ErrorThreshold` to zero. For time-domain data that contains outliers, try setting `ErrorThreshold` to `1.6`.

  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 condition.

  The initial condition is estimated when

  $$\frac{\left\| y_{p,z} - y_{meas} \right\|}{\left\| y_{p,e} - y_{meas} \right\|} > \text{AutoInitThreshold}$$

  - y_{meas} is the measured output.
  - $y_{p,z}$ is the predicted output of a model estimated using zero initial conditions.
  - $y_{p,e}$ is the predicted output of a model estimated using estimated initial conditions.

  Applicable when `InitialCondition` is `'auto'`.

  Default: `1.05`

Output Arguments

`opt` — Options set for `oe`

`oeOptions` option set

Option set for `oe`, returned as an `oeOptions` option set.

Examples

Create Default Options Set for Output-Error Estimation

```
opt = oeOptions;
```

Specify Options for Output-Error Estimation

Create an options set for `oe` using the `'backcast'` algorithm to initialize the condition and set the `Display` to `'on'`.

```
opt = oeOptions('InitialCondition','backcast','Display','on');
```

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

```
opt = oeOptions;
opt.InitialCondition = 'backcast';
opt.Display = 'on';
```

More About

- "Loss Function and Model Quality Metrics"

References

[1] 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.

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

See Also

`idfilt` | `oe`

Introduced in R2012a

idnlarx/operspec

Construct operating point specification object for `idnlarx` model

Syntax

```
spec = operspec(nlsys)
```

Description

`spec = operspec(nlsys)` creates a default operating point specification object for the `idnlarx` model `nlsys`. This object is used with `findop` and specifies constraints on the model input and output signal values. Modify the default specifications using dot notation.

Input Arguments

`nlsys` — Nonlinear ARX model

`idnlarx` object

Nonlinear ARX model, specified as an `idnlarx` object.

Output Arguments

`spec` — Operating point specification

operating point specification object

Operating point specification, used to determine an operating point of the `idnlarx` model using `findop`, returned as an object containing the following:

- `Input` — Structure with fields:

Field	Description	Default for Each Input
`Value`	Initial guesses or fixed levels for the values of the model inputs, specified as a vector with length equal to the number of input signals.	`0`
`Min`	Minimum value constraints on the model inputs, specified as a vector with length equal to the number of input signals.	`-Inf`
`Max`	Maximum value constraints on the model inputs, specified as a vector with length equal to the number of input signals.	`Inf`
`Known`	Known value indicator, specified as a logical vector with length equal to the number of input signals and with the following values: • `true` — `findop` will set the corresponding input signal to `Value`. • `false` — `findop` will estimate the corresponding input signal using `Value` as an initial guess.	`true`

- `Output` — Structure with fields:

Field	Description	Default for Each Output
`Value`	Initial guesses for the values of the model outputs, specified as a vector with length equal to the number of output signals.	`0`
`Min`	Minimum value constraints on the model outputs, specified as a vector with length equal to the number of output signals.	`-Inf`
`Max`	Maximum value constraints on the model outputs, specified as a vector with length equal to the number of output signals.	`Inf`

See Also

`idnlarx/findop`

Introduced in R2008a

idnlhw/operspec

Construct operating point specification object for `idnlhw` model

Syntax

```
spec = operspec(nlsys)
```

Description

`spec = operspec(nlsys)` creates a default operating point specification object for the `idnlhw` model `nlsys`. This object is used with `findop` and specifies constraints on the model input and output signal values. Modify the default specifications using dot notation.

Input Arguments

`nlsys` — Nonlinear Hammerstein-Wiener model

`idnlhw` object

Nonlinear Hammerstein-Wiener model, specified as an `idnlhw` object.

Output Arguments

`spec` — Operating point specification

operating point specification object

Operating point specification, used to determine an operating point of the `idnlhw` model using `findop`, returned as an object containing the following:

- `Input` — Structure with fields:

Field	Description	Default for Each Input
`Value`	Initial guesses or fixed levels for the values of the model inputs, specified as a vector with length equal to the number of input signals.	`0`
`Min`	Minimum value constraints on the model inputs, specified as a vector with length equal to the number of input signals.	`-Inf`
`Max`	Maximum value constraints on the model inputs, specified as a vector with length equal to the number of input signals.	`Inf`
`Known`	Known value indicator, specified as a logical vector with length equal to the number of input signals and with the following values: • `true` — `findop` will set the corresponding input signal to `Value`. • `false` — `findop` will estimate the corresponding input signal using `Value` as an initial guess.	`true`

- `Output` — Structure with fields:

Field	Description	Default for Each Input
`Value`	Target values the model outputs, specified as a vector with length equal to the number of output signals.	`0`
`Min`	Minimum value constraints on the model outputs, specified as a vector with length equal to the number of output signals.	`-Inf`
`Max`	Maximum value constraints on the model outputs, specified as a vector with length equal to the number of output signals.	`Inf`
`Known`	Known value indicator, specified as a logical vector with length equal to the number of output signals and with the following values: • `true` — `findop` will use `Value` as an estimation target for the corresponding output. • `false` — `findop` will keep the corresponding output within the constraints specified by `Min` and `Max`.	`false`

Note:

1. If `Input.Known` is `true` for all model inputs, then the initial state values are determined using the input specifications only. In this case, `findop` ignores the specifications in the `Output` structure.

2. Otherwise, `findop` uses the output specifications to meet the objectives indicated by `Output.Known`.

See Also

`idnlhw/findop`

Introduced in R2008a

order

Query model order

Syntax

```
NS = order(sys)
```

Description

`NS = order(sys)` returns the model order `NS`. 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).

`order(sys)` is an overloaded method that accepts SS, TF, and ZPK models. For LTI arrays, `NS` is an array of the same size listing the orders of each model in `sys`.

Caveat

`order` does not attempt to find minimal realizations of MIMO systems. For example, consider this 2-by-2 MIMO system:

```
s=tf('s');
h = [1, 1/(s*(s+1)); 1/(s+2), 1/(s*(s+1)*(s+2))];
order(h)
ans =

     6
```

Although `h` has a 3rd order realization, `order` returns 6. Use

```
order(ss(h,'min'))
```

to find the minimal realization order.

See Also

`pole` | `balred`

Introduced in R2012a

pe

Prediction error for identified model

Syntax

```
err = pe(sys,data,K)
err = pe(sys,data,K,opt)
[err,x0e,sys_pred] = pe( ___ )
pe(sys,data,K, ___ )
pe(sys,Linespec,data,K, ___ )
pe(sys1,...,sysN,data,K, ___ )
pe(sys1,Linespec1,...,sysN,LinespecN,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 on the 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( ___ )` also returns the estimated initial state, `x0e`, and a predictor system, `sys_pred`.

`pe(sys,data,K, ___ )` plots the prediction error. Use with any of the previous input argument combinations. To change display options in the plot, right-click the plot to access the context menu. For more details about the menu, see "Tips" on page 1-949.

`pe(sys,Linespec,data,K, ___ )` uses `Linespec` to specify the line type, marker symbol, and color.

`pe(sys1,...,sysN,data,K, ___ )` plots the prediction errors for multiple identified models. `pe` automatically chooses colors and line styles.

`pe(sys1,Linespec1,...,sysN,LinespecN,data,K, ___ )` uses the line type, marker symbol, and color specified for each model.

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 can 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, created using `peOptions`, that configures the computation of the predicted response. Options that you can specify include:

- Handling of initial conditions
- Data offsets

`Linespec`

Line style, marker, and color

Line style, marker, and color, specified as a character vector. For example, `'b'` or `'b+:'`.

For more information about configuring `Linespec`, see “Specify Line Style, Color, and Markers” in the MATLAB documentation.

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, the output, *yp*, is such that `err.OutputData = data.OutputData - yp`. For state-space models, the software uses `x0e` 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 one-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.

Plot Prediction Error for Multiple Models

Load the estimation data.

```
load iddata1;
```

```
data = z1;
```

Estimate an ARX model of order [2 2 1].

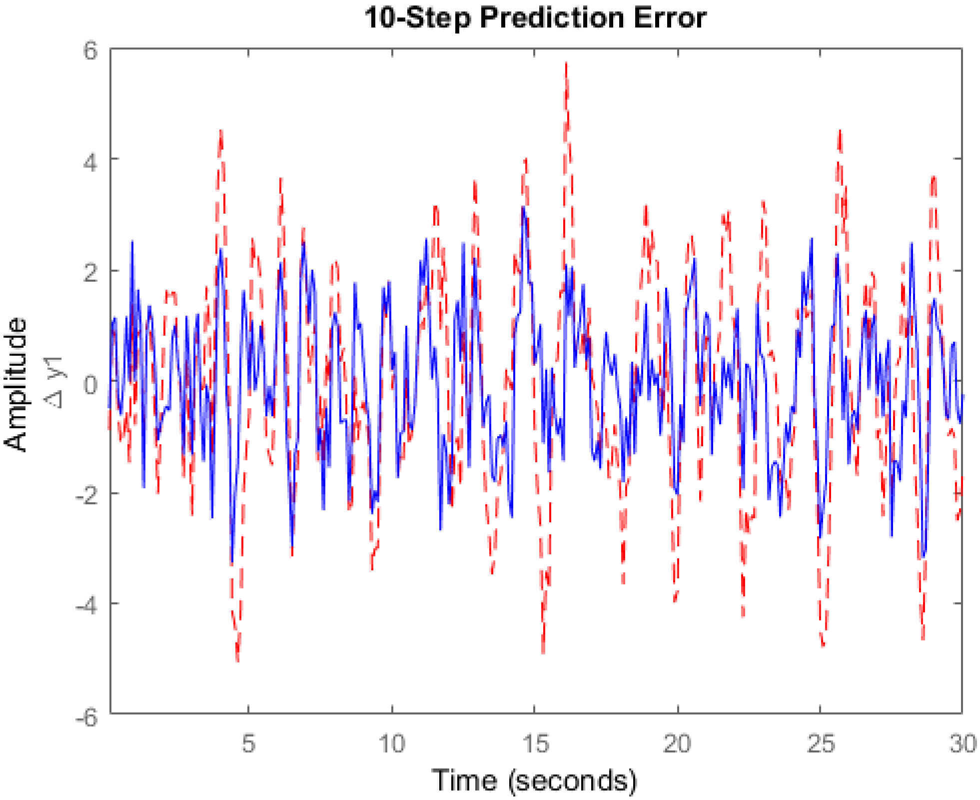
```
sys1 = arx(data,[2 2 1]);
```

Estimate a transfer function with 2 poles.

```
 sys2 = tfest(data,2);
```

Plot the prediction error for the estimated models. Specify prediction horizon as 10, and specify the line styles for plotting the prediction error of each system.

```
pe(sys1,'r--',sys2,'b',data,10);
```



To change the display options, right-click the plot to access the context menu. For example, to view the estimation data, select **Show Validation Data** from the context menu. To view the predicted outputs, select **Predicted Response Plot**.

More About

Tips

- Right-clicking the plot of the prediction error opens the context menu, where you can access the following options:
    - **Systems** — Select systems to view prediction error. By default, the prediction error of all systems is plotted.
    - **Data Experiment** — For multi-experiment data only. Toggle between data from different experiments.
    - **Characteristics** — View the following data characteristics:
        - **Peak Value** — View the absolute peak value of the data. Applicable for time–domain data only.
        - **Peak Response** — View peak response of the data. Applicable for frequency–response data only.
        - **Mean Value** — View mean value of the data. Applicable for time–domain data only.
    - **Show** — For frequency–domain and frequency–response data only.
        - **Magnitude** — View magnitude of frequency response of the system.
        - **Phase** — View phase of frequency response of the system.
    - **Show Validation Data** — Plot data used to compute the prediction error.
    - **I/O Grouping** — For datasets containing more than one input or output channel. Select grouping of input and output channels on the plot.
        - **None** — Plot input-output channels in their own separate axes.
        - **All** — Group all input channels together and all output channels together.
    - **I/O Selector** — For datasets containing more than one input or output channel. Select a subset of the input and output channels to plot. By default, all output channels are plotted.

- **Grid** — Add grids to the plot.
- **Normalize** — Normalize the y-scale of all data in the plot.
- **Full View** — Return to full view. By default, the plot is scaled to full view.
- **Prediction Horizon** — Set the prediction horizon, or choose simulation.
- **Initial Condition** — Specify handling of initial conditions. Not applicable for frequency-response data.

  Specify as one of the following:

  - **Estimate** — Treat the initial conditions as estimation parameters.
  - **Zero** — Set all initial conditions to zero.
  - **Absorb delays and estimate** — Absorb nonzero delays into the model coefficients and treat the initial conditions as estimation parameters. Use this option for discrete-time models only.
- **Predicted Response Plot** — Plot the predicted model response.
- **Prediction Error Plot** — Plot the error between the model response and prediction data. By default, the error plot is shown.
- **Properties** — Open the Property Editor dialog box to customize plot attributes.

See Also

`ar` | `arx` | `compare` | `iddata` | `n4sid` | `peOptions` | `predict` | `resid` | `sim`

Introduced before R2006a

peOptions

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. `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`. If 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 *Ne* elements, where *Ne* 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, `sys`, `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: `'e'`

`'InputOffset'`

Removes offset from time domain input data during prediction-error calculation.

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

For multi-experiment data, specify `InputOffset` as an *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.

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 *Ny*, where *Ny* is the number of outputs.

In case of multi-experiment 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.

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 value is the same as using `eye(Ny)` for the output weight, where *Ny* is the number of outputs.
- `'noise'` — Inverse of the noise variance stored with the model.
- matrix — A positive, semidefinite matrix of dimension *Ny*-by-*Ny*, where *Ny* 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

```
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

`idpar` | `pe`

Introduced in R2012a

pem

Prediction error estimate for linear and nonlinear model

Syntax

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

Description

`sys = pem(data,init_sys)` updates the parameters of an initial model to fit the estimation data. The function uses prediction-error minimization algorithm to update the parameters of the initial model. Use this command to refine the parameters of a previously estimated model.

`sys = pem(data,init_sys,opt)` specifies estimation options using an option set.

Examples

Refine Estimated State-Space Model

Estimate a discrete-time state-space model using the subspace method. Then, 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` provides a 73.85% fit to the estimation data.

```
init_sys.Report.Fit.FitPercent
```

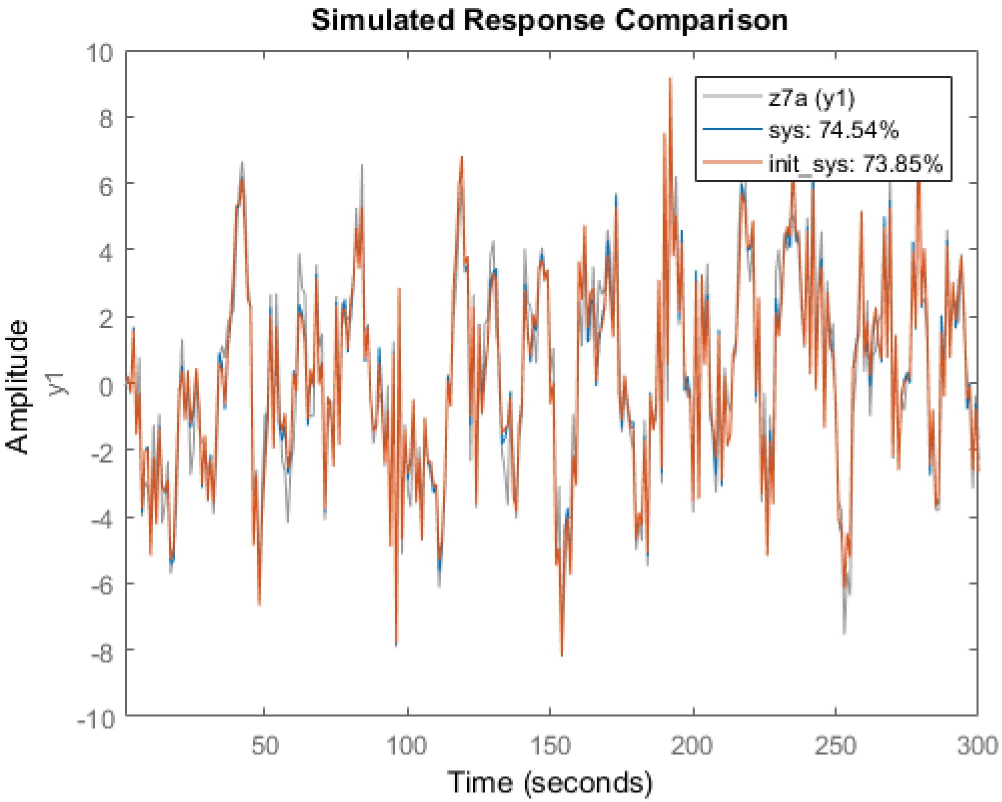
```
ans =

   73.8490
```

Use `pem` to improve the closeness of the fit.

```
sys = pem(z7a,init_sys);
```

Analyze the results.

```
compare(z7a,sys,init_sys);
```



`sys` provides a 74.54% fit to the estimation data.

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','dcmotordata'));
data = iddata(y, u, 0.1);
data.Tstart = 0;
data.TimeUnit = 's';
```

Configure the nonlinear grey-box model (`idnlgrey`) model.

For this example, use `dcmotor_m.m` file. To view this file, type `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);
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.

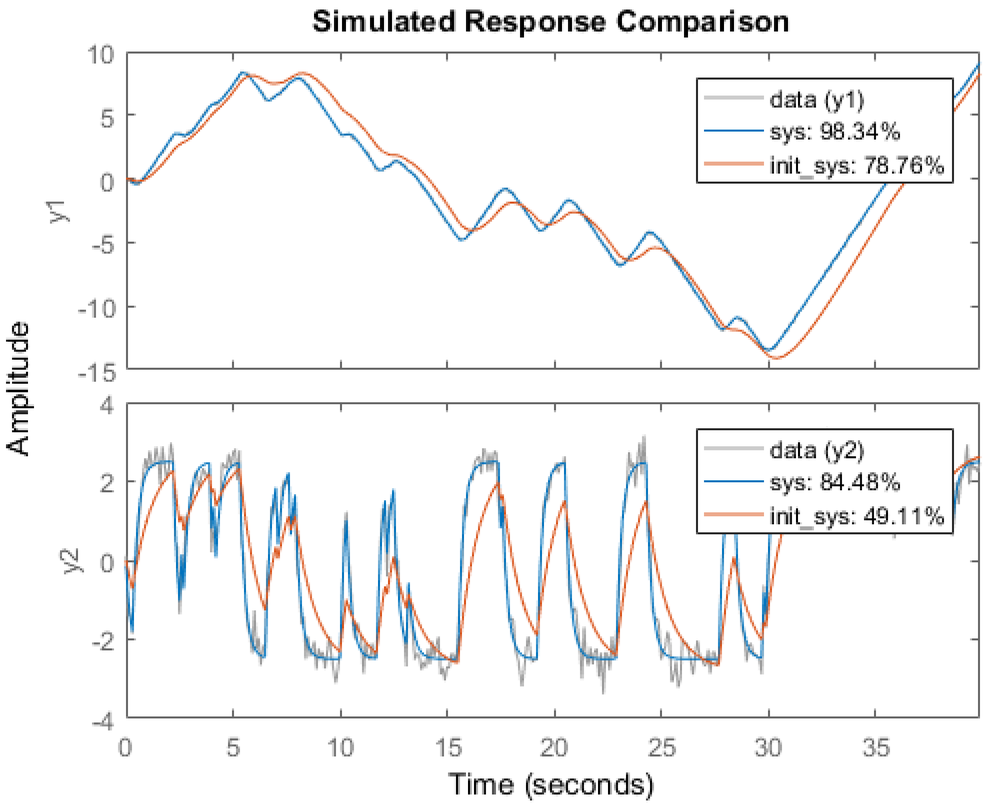
Estimate the model parameters and initial states.

```
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 estimation data.

Configure Estimation Using Process Model

Create a process model structure and update its parameter values to minimize prediction error.

Initialize the coefficients of a process model.

```
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;
```

The `Kp`, `Tw`, `Zeta`, `Td`, and `Tz` coefficients of `init_sys` are configured with their initial guesses.

Use `init_sys` to configure the estimation of a prediction error minimizing model using measured data. Because `init_sys` is an `idproc` model, use `procestOptions` to create the option set.

```
load iddata1 z1;
opt = procestOptions('Display','on','SearchMethod','lm');
sys = pem(z1,init_sys,opt);
```

Examine the model fit.

```
sys.Report.Fit.FitPercent
```

```
ans =

   70.6330
```

`sys` provides a 70.63% fit to the measured data.

- “Refine Linear Parametric Models”

Input Arguments

`data` — Estimation data

`iddata` | `idfrd`

Estimation data that contains measured input-output data, specified as an `iddata` or `idfrd` object. You can use frequency-domain data only when `init_sys` is a linear model.

The input-output dimensions of `data` and `init_sys` must match.

`init_sys` — Identified model that configures the initial parameterization of `sys`

linear model | nonlinear model

Identified model that configures the initial parameterization of `sys`, specified as a linear, or nonlinear model. You can obtain `init_sys` by performing an estimation using measured data or by direct construction.

`init_sys` must have finite parameter values. You can configure initial guesses, specify minimum/maximum bounds, and fix or free for estimating any parameter of `init_sys`:

- For linear models, use the `Structure` property. For more information, see “Imposing Constraints on Model Parameter Values”.
- For nonlinear grey-box models, use the `InitialStates` and `Parameters` properties. Parameter constraints cannot be specified for nonlinear ARX and Hammerstein-Wiener models.

`opt` — Estimation options

option set

Estimation options that configure the algorithm settings, handling of estimation focus, initial conditions, and data offsets, specified as an option set. The command used to create the option set depends on the initial model type:

Model Type	Use
`idss`	`ssestOptions`
`idtf`	`tfestOptions`
`idproc`	`procestOptions`
`idpoly`	`polyestOptions`
`idgrey`	`greyestOptions`
`idnlarx`	`nlarxOptions`
`idnlhw`	`nlhwOptions`
`idnlgrey`	`nlgreyestOptions`

Output Arguments

`sys` — Identified model

linear model | nonlinear model

Identified model, returned as the same model type as `init_sys`. The model is obtained by estimating the free parameters of `init_sys` using the prediction error minimization algorithm.

Alternative Functionality

You can achieve the same results as `pem` by using dedicated estimation commands for the various model structures. For example, use `ssest(data,init_sys)` for estimating state-space models.

More About

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, the error is defined as:

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

where *e(t)* is a vector and the cost function $V_N(G,H)$ is a scalar value. The subscript N indicates that the cost function is a function of the number of data samples and becomes more accurate for larger values of N. For multiple-output models, the previous equation is more complex. For more information, see chapter 7 in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

See Also

`armax` | `bj` | `greyest` | `n4sid` | `nlarx` | `nlgreyest` | `nlhw` | `oe` | `polyest` | `procest` | `ssest` | `tfest`

Introduced before R2006a

pexcit

Level of excitation of input signals

Syntax

```
Ped = pexcit(Data)
[Ped.Maxnr] = pexcit(Data,Maxnr,Threshold)
```

Description

`Ped = pexcit(Data)` tests the degree of persistence of excitation for the input. `Data` is an `iddata` object with time- or frequency-domain signals. `Ped` is the degree or order of excitation of the inputs in `Data` and is 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.

`[Ped.Maxnr] = pexcit(Data,Maxnr,Threshold)` specifies the maximum order tested and threshold level used to measure which singular values are significant. Default value of `Maxnr` is `min(N/3,50)`, where `N` is the number of input data. Default value of `Threshold` is `1e-9`.

References

Section 13.2 in Ljung (1999).

See Also

`advice` | `iddata` | `feedback` | `idnlarx`

Introduced before R2006a

iddata/plot

Plot input-output data

Syntax

```
plot(data)
plot(data,LineSpec)
plot(data1,...,dataN)
plot(data1,LineSpec1...,dataN,LineSpecN)

plot(axes_handle,___)

plot(___,plotoptions)

h = plot(___)
```

Description

`plot(data)` plots the input and output channels of an `iddata` object. The outputs are plotted on the top axes and the inputs on the bottom axes.

- For time-domain data, the input and output signals are plotted as a function of time. Depending on the `InterSample` property of the `iddata` object, the input signals are plotted as linearly interpolated curves or as staircase plots. For example, if `data.InterSample = 'zoh'`, the input is piecewise constant between sampling points, and it is then plotted accordingly.
- For frequency-domain data, the magnitude and phase of each input and output signal is plotted over the available frequency span.

To plot a subselection of the data, use subreferencing:

- `plot(data(201:300))` plots the samples 201 to 300 in the dataset `data`.
- `plot(data(201:300,'Altitude',{'Angle_of_attack','Speed'}))` plots the chosen samples of output named `Altitude` and inputs named `Angle_of_attack` and `Speed`.

- `plot(data(:,[3 4],[3:7]))` plots all samples of output channel numbers 3 and 4 and input numbers 3 through 7.

`plot(data,LineSpec)` specifies the color, line style and marker symbol for the dataset.

`plot(data1,...,dataN)` plots multiple datasets. The number of plot axes are determined by the number of unique input and output names among all the datasets.

`plot(data1,LineSpec1...,dataN,LineSpecN)` specifies the line style, marker type, and color for each dataset. You can mix `data,LineSpec` pairs with `data`. For example, `plot(data1,data2,LineSpec2,data3)`.

`plot(axes_handle, ___ )` plots into the axes with handle `axes_handle` instead of into the current axes (`gca`). This syntax can include any of the input argument combinations in the previous syntaxes.

`plot( ___ ,plotoptions)` specifies the plot options. This syntax can include any of the input argument combinations in the previous syntaxes.

`h = plot( ___ )` returns the handle to the plot. You can use this handle to customize the plot with `getoptions` and `setoptions`.

Input Arguments

`data` — Input-output data

`iddata` object

Input-output data, specified as an `iddata` object. The data can be time-domain or frequency-domain. It can be a single- or multi-channel data, or single- or multi-experiment data.

`LineSpec` — Line style, marker symbol, and color

character vector

Line style, marker symbol, and color, specified as a character vector. `LineSpec` takes values such as `'b'`, `'b+:'`. For more information, see the `plot` reference page in the MATLAB documentation.

Data Types: `char`

`axes_handle` — Axes handle

handle

Axes handle, which is the reference to an axes object. Use the `gca` function to get the handle to the current axes, for example, `axes_handle= gca`.

`plotoptions` — Plot options

structure

Plot options, specified as an option set created using `iddataPlotOptions`.

Output Arguments

h — Lineseries handle

scalar | vector

Lineseries handle, returned as a scalar or vector. These are unique identifiers, which you can use to query and modify properties of a specific plot.

Examples

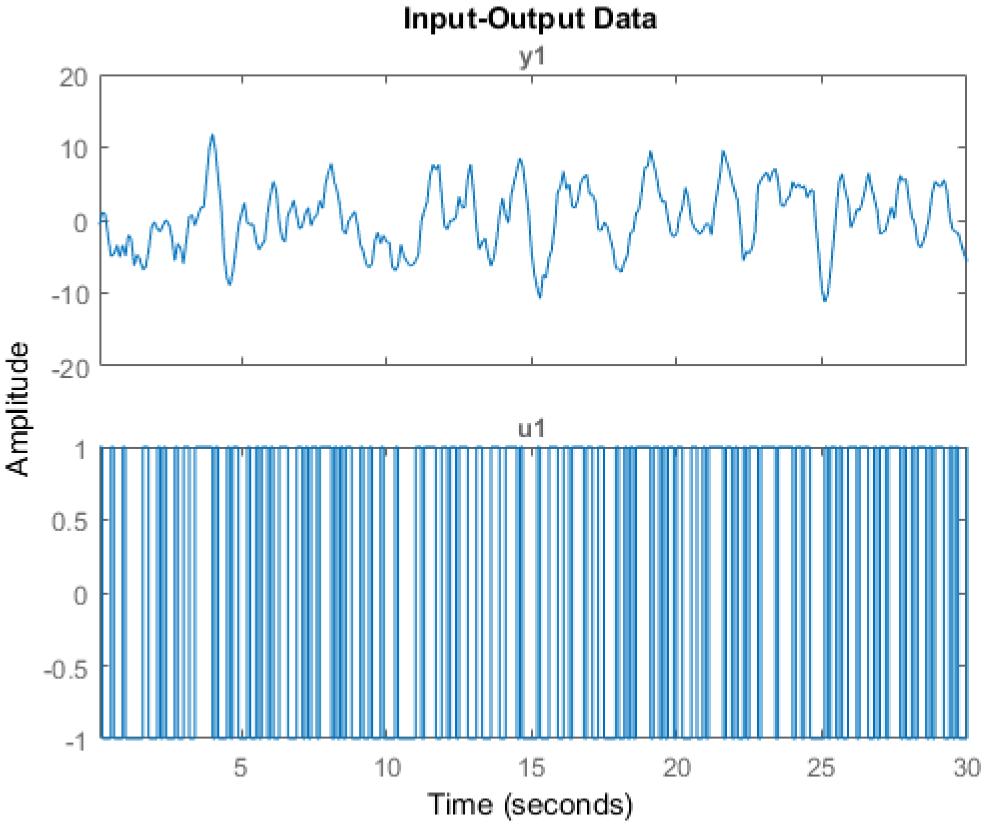
Plot Time-Domain Input-Output Data

Load the data.

```
load iddata1 z1;
```

Plot the data.

```
plot(z1)
```



The output is plotted on the top axes and the input on the bottom axes.

You can right-click the plot to explore its characteristics such as peak and mean values.

Plot Frequency-Domain Input-Output Data

Load the data.

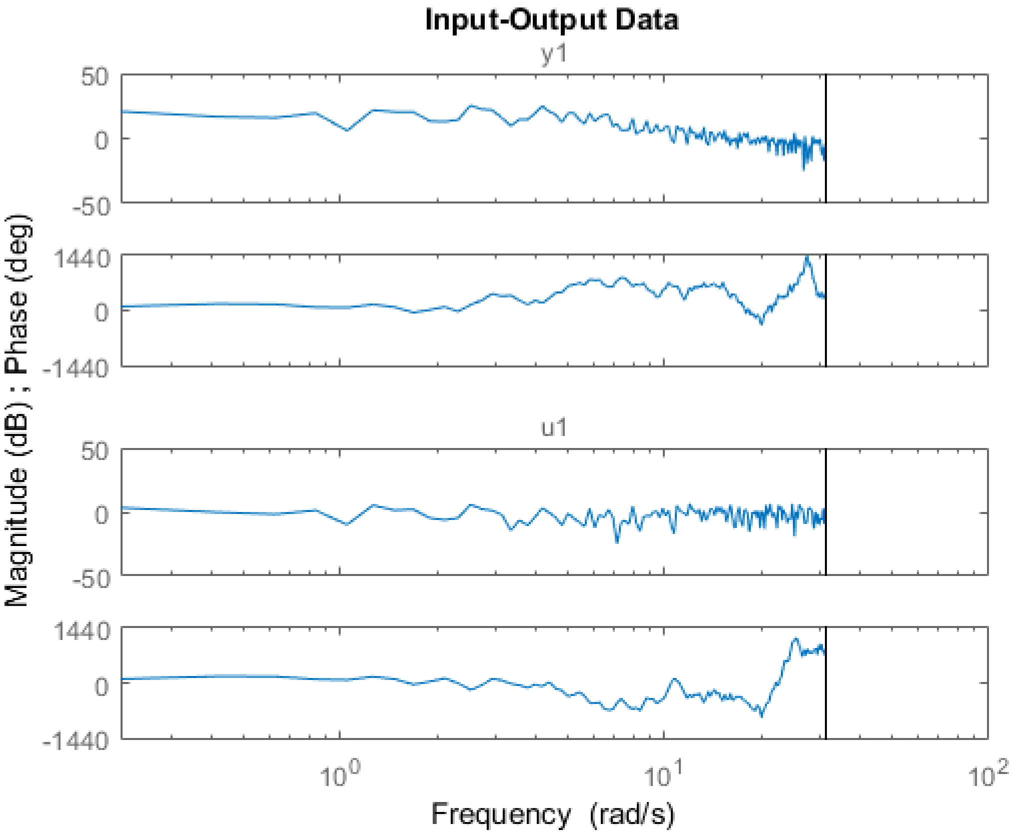
```
load iddata1 z1
```

Convert the data to frequency domain.

```
zf = fft(z1);
```

Plot the data.

```
plot(zf);
```



Plot Input Data, Output Data and Input-Output Data

Generate input data.

```
u = idinput([100 1 20],'sine',[],[],[5 10 1]);
u = iddata([],u,1,'per',100);
```

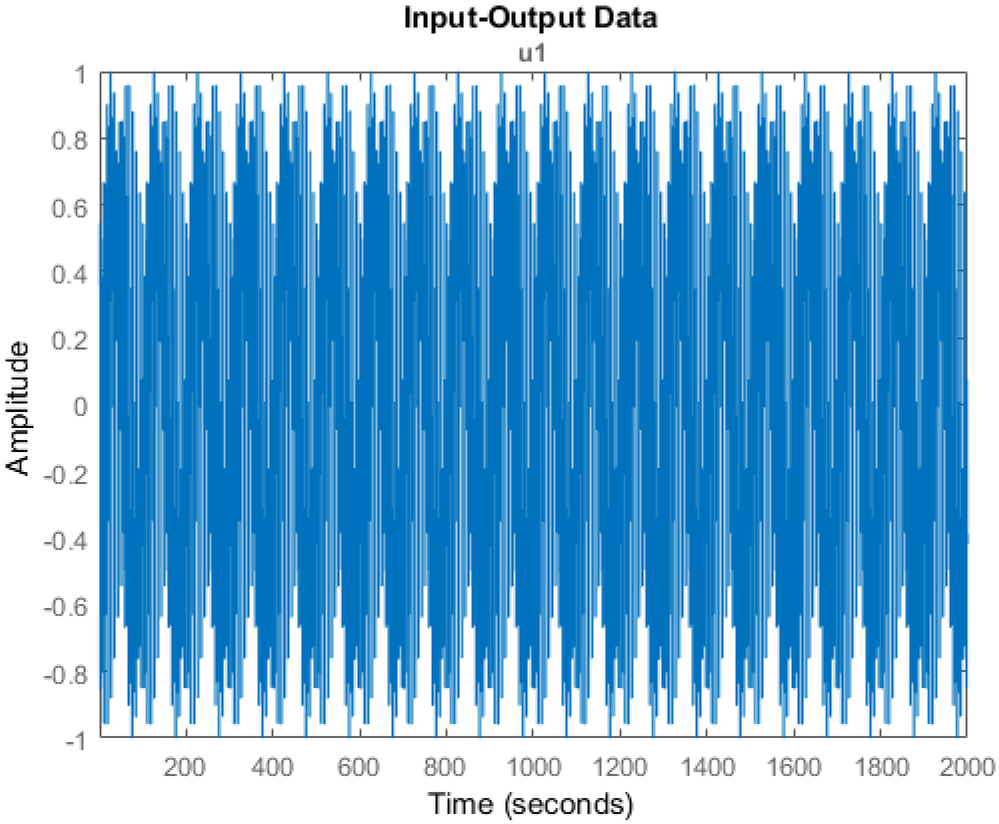
Generate output data.

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

```
y = sim(sys,u);
```

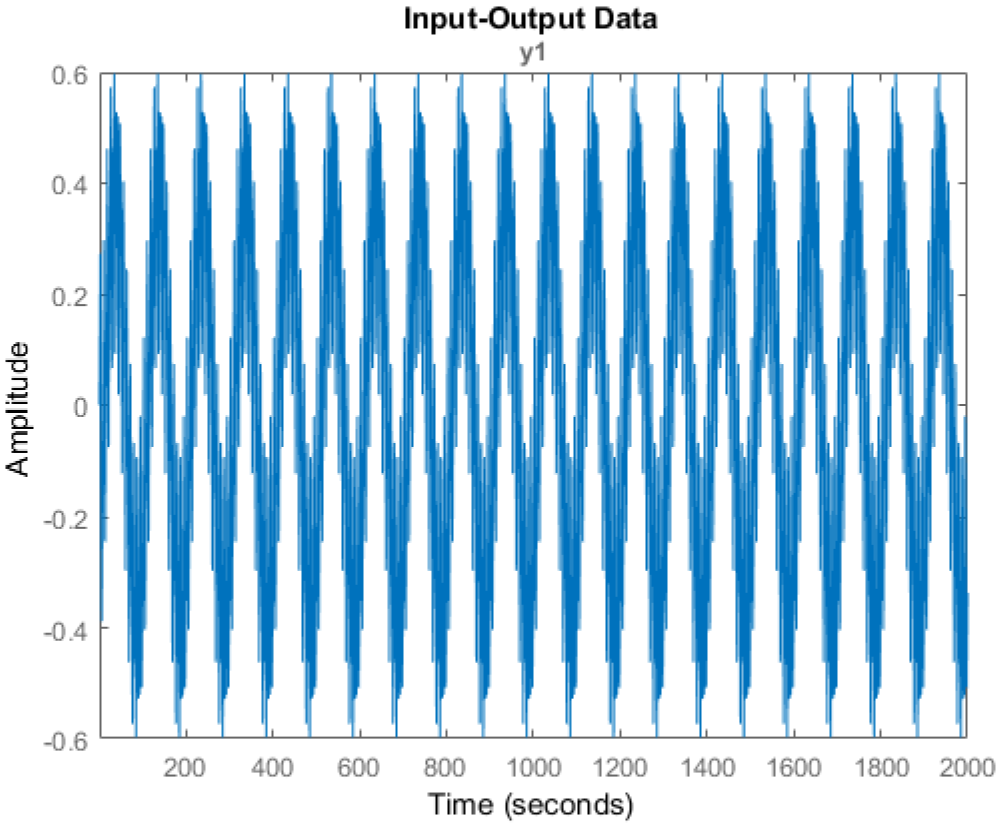
Plot only the input.

```
plot(u)
```



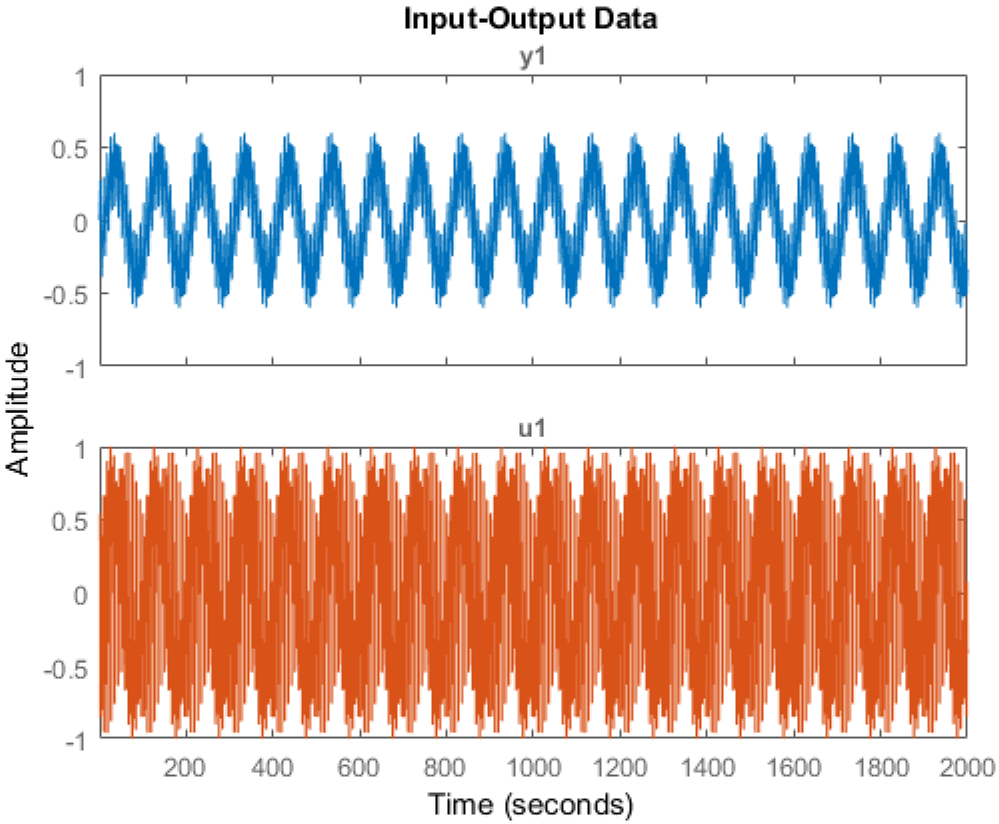
Plot only the output.

```
plot(y)
```



Plot the input and output together.

```
plot(y,u)
```



Alternatively, you can use `plot(iddata(y,u))` .

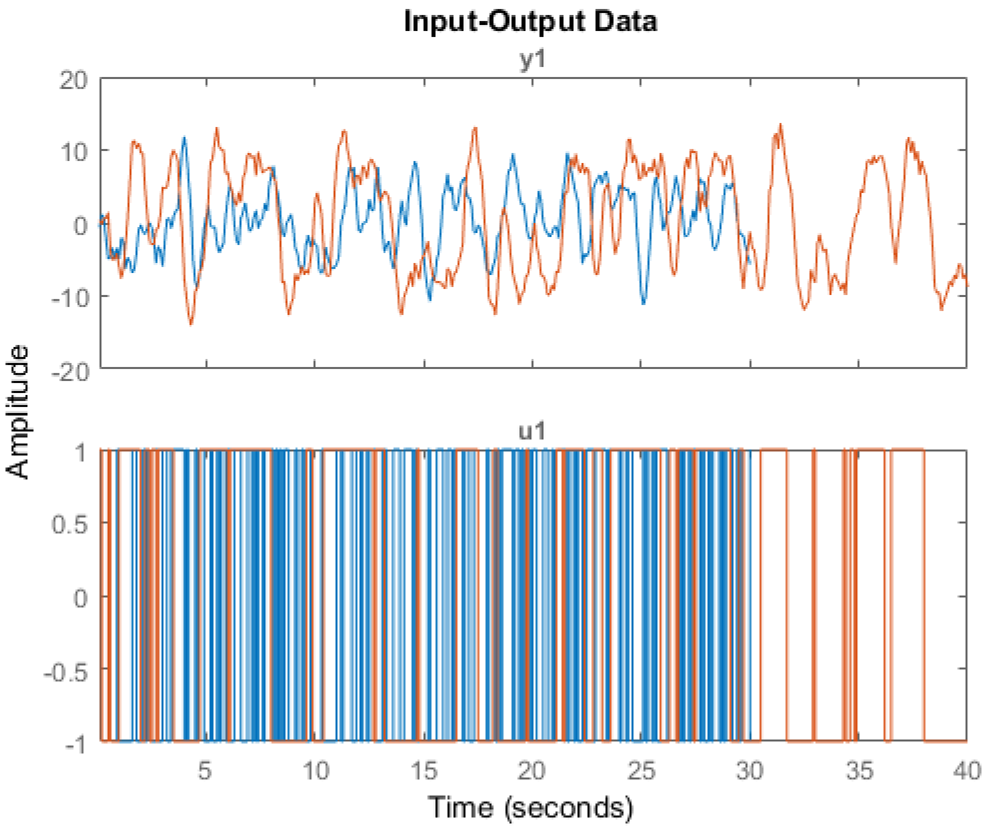
Plot Multi-Experiment Data

Create a multi-experiment data set.

```
load iddata1 z1
load iddata2 z2
zm = merge(z1,z2);
```

Plot the data.

```
plot(zm);
```



For multi-experiment data, each experiment is treated as a separate data set. You can right-click the plots to view their characteristics.

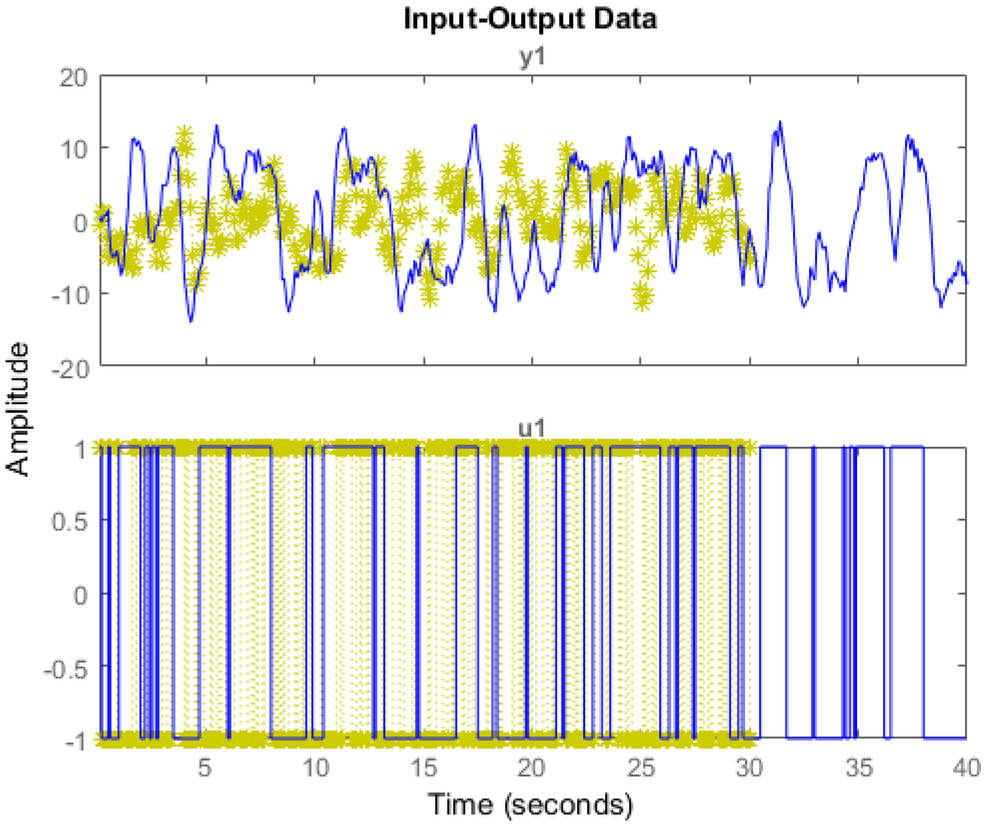
Specify Line Style, Marker Symbol and Color

Load multiple datasets.

```
load iddata1 z1;
load iddata2 z2;
```

Specify the linestyle properties.

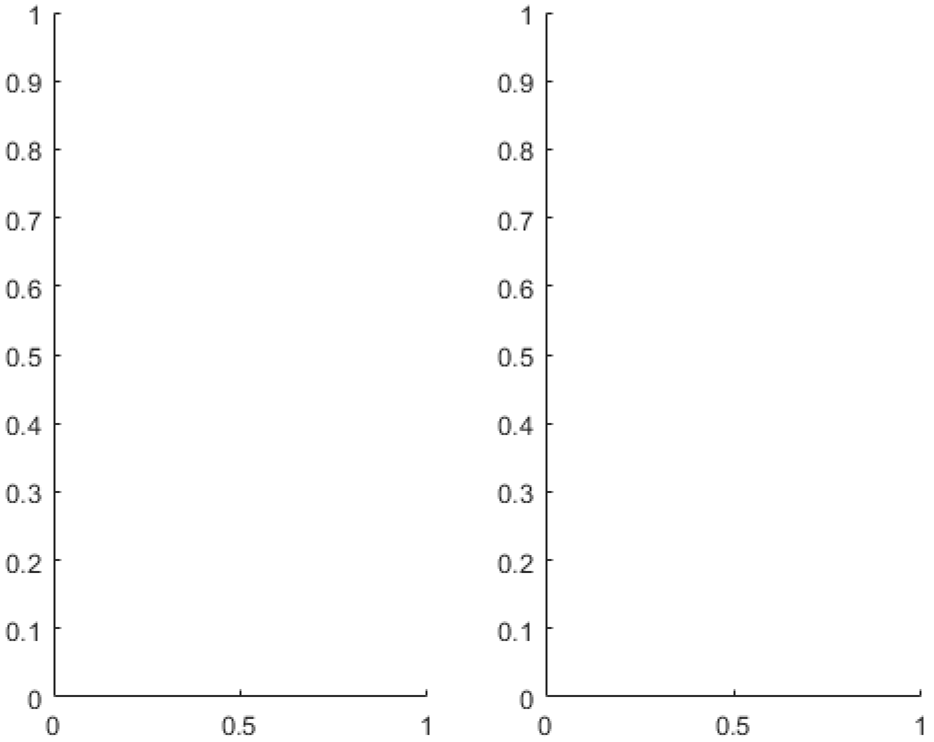
```
plot(z1,'y:*',z2,'b')
```



Specify Axes Handle

Create a figure with two subplots and return the handle to each subplot axes, s(1) and s(2).

```
figure; % new figure
s(1) = subplot(1,2,1); % left subplot
s(2) = subplot(1,2,2); % right subplot
```

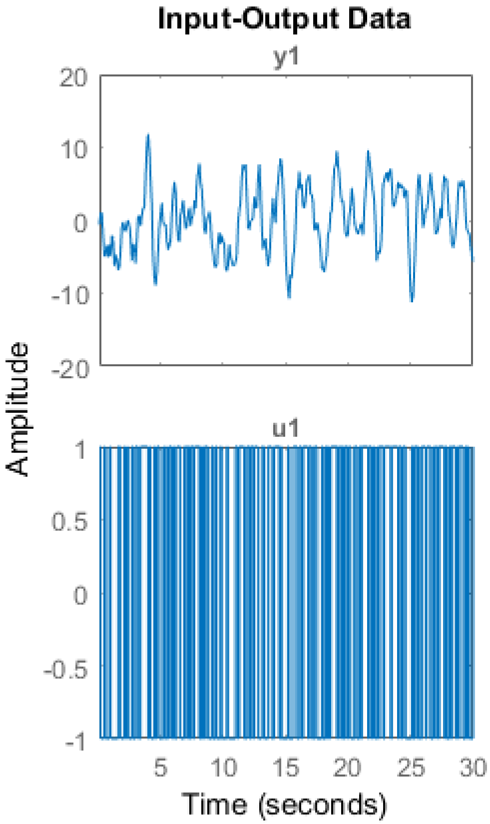


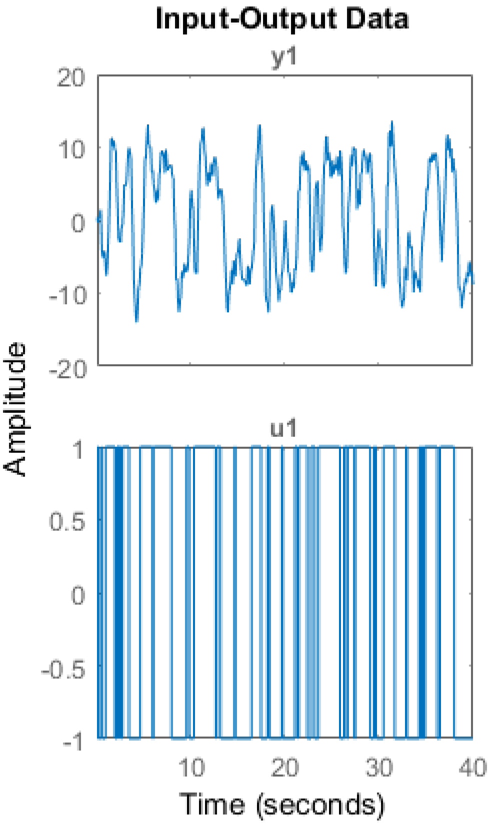
Load the data sets.

```
load iddata1;
load iddata2;
```

Create a data plot in each axes referring to the axes handles.

```
plot(s(1),z1)
plot(s(2),z2)
```





Specify Plot Options

Configure a time plot.

```
opt = iddataPlotOptions('time');
```

View the plot in minutes time units.

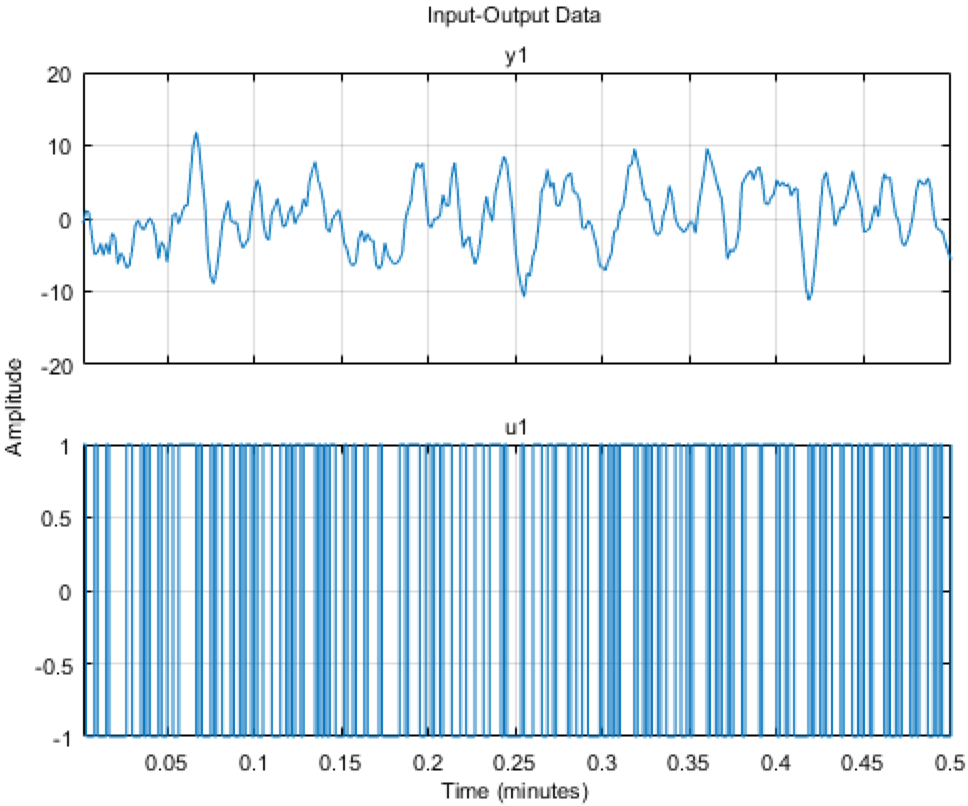
```
opt.TimeUnits = 'minutes';
```

Turn grid on.

```
opt.Grid = 'on';
```

Create plot with the options specified by opt.

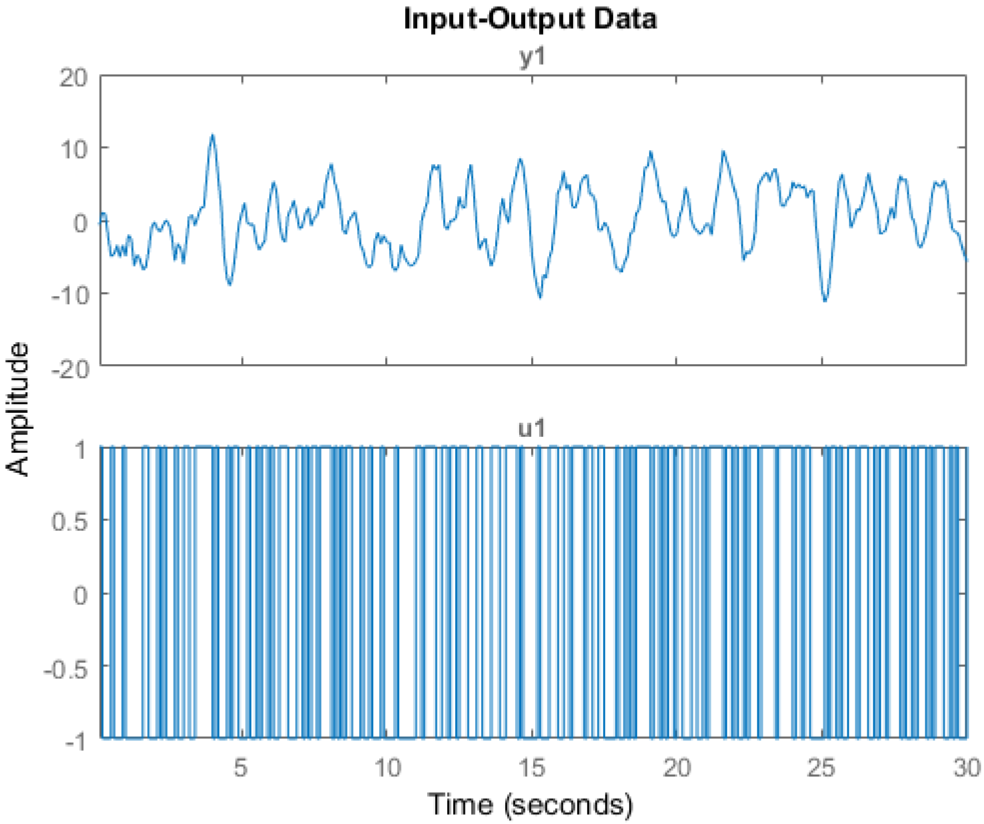
```
load iddata1 z1
plot(z1, opt);
```



Change Plot Properties Using Handles

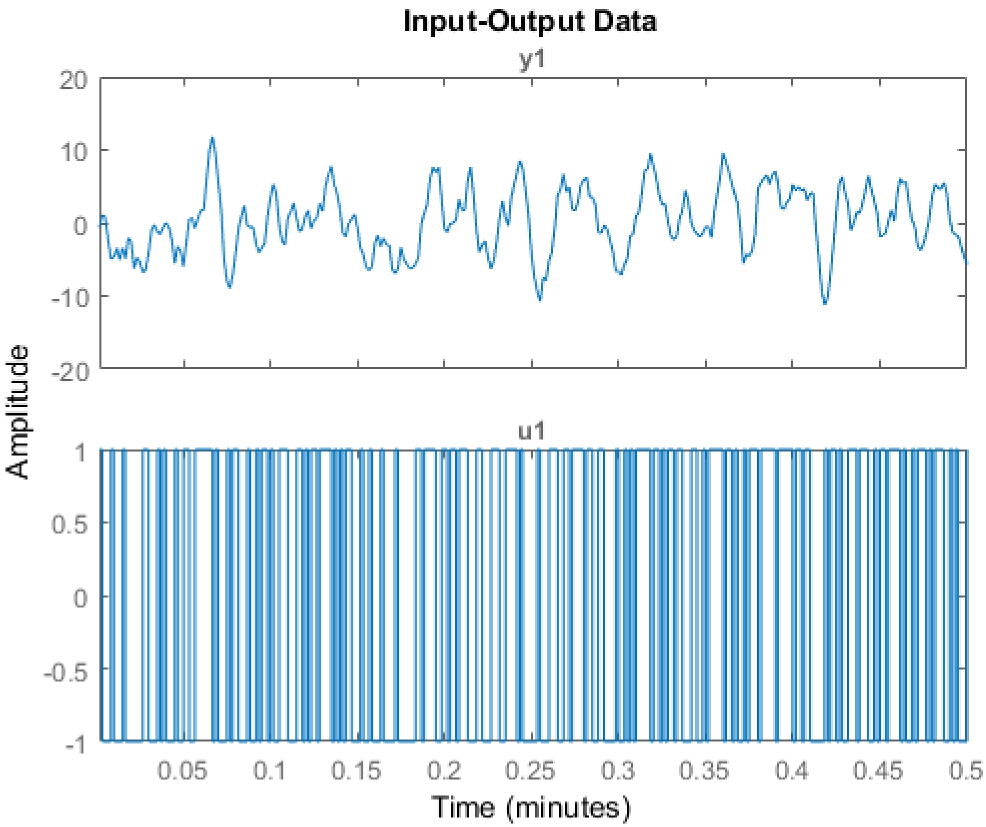
Create a data plot and return the handle.

```
load iddata1;
h = plot(z1);
```



Set the time unit on the plot.

```
setoptions(h,'TimeUnits','minutes');
```



Change Orientation of Input-Output Data Axes

Generate data with two inputs and one output.

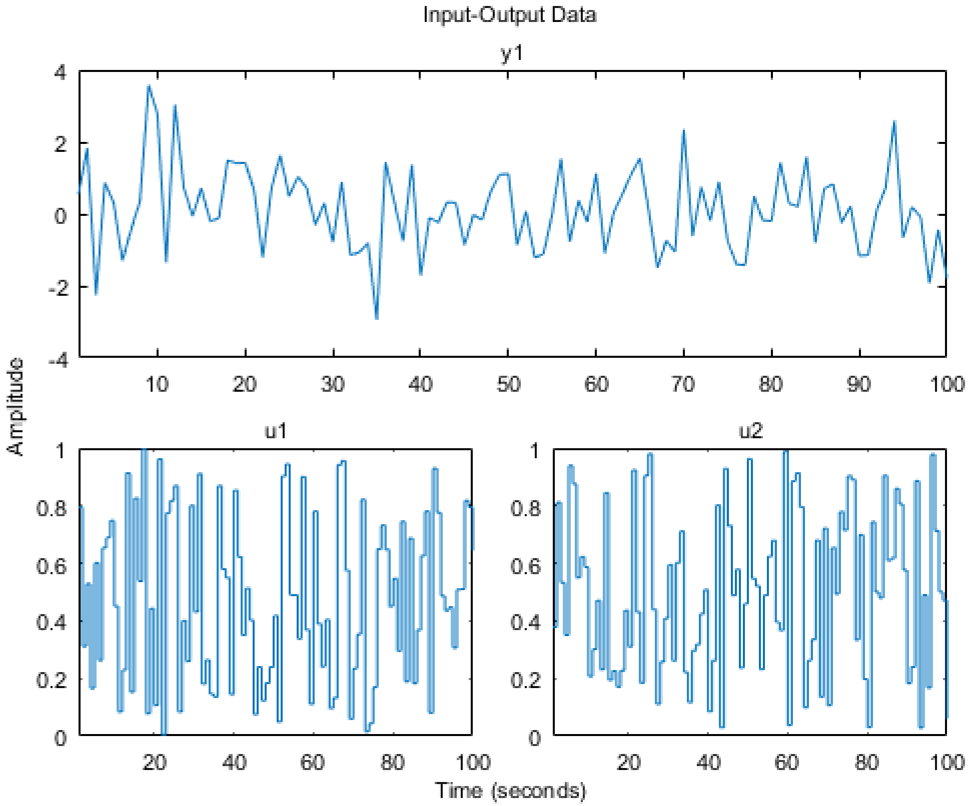
```
z = iddata(randn(100,1),rand(100,2));
```

Configure a time plot.

```
opt = iddataPlotOptions('time');
```

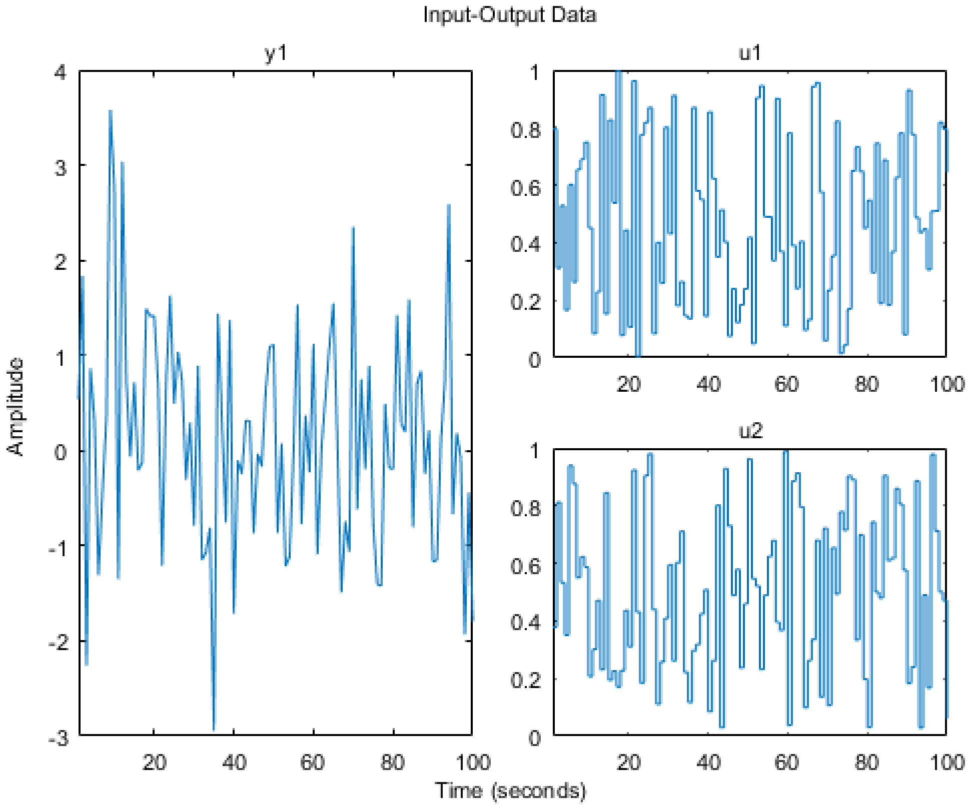
Plot the data.

```
h = plot(z,opt);
```



Change the orientation of the plots such that all inputs are plotted in one column, and all outputs are in a second column.

```
opt.Orientation = 'two-column';
h = plot(z,opt);
```



Alternatively, use `setoptions`.

```
setoptions(h,'Orientation','two-column')
```

You can also change the orientation by right-clicking the plot and choosing `Orientation` in the context menu.

More About

Tips

- Right-clicking the plot opens the context menu where you can access the following options and plot controls:

- **Datasets** — View the datasets used in the plot.
- **Characteristics** — View data characteristics.
    - **Peak Value** — Peak value of the data. Useful for transient data.
    - **Mean Value** — Mean value of the data. Useful for steady-state data.
- **Orientation** — Choose orientation of the input and output plots.
    - **Output row and input row** — (For datasets containing more than one input or output channel) Plot all outputs in one row and all inputs in a second row.
    - **Output column and input column** — (For datasets containing more than one input or output channel) Plot all outputs in one column and all inputs in a second column.
    - **Single row** — Plot all inputs and outputs in one row.
    - **Single column**— Plot all inputs and outputs in one column.
- **I/O Grouping** — (For datasets containing more than one input or output channel) Group input and output channels on the plot.
- **I/O Selector** — (For datasets containing more than one input or output channel) Select a subset of the input and output channels to plot. By default, all input and output channels are plotted.
- **Grid** — Add grids to your plot.
- **Normalize** — Normalize the y-scale of all data in the plot.
- **Properties** — Open the Property Editor dialog box, where you can customize plot attributes.

See Also

`iddata` | `iddataPlotOptions` | `identpref`

Introduced in R2014a

idnlarx/plot

Plot nonlinearity of nonlinear ARX model

Syntax

```
plot(model)
plot(model,color)
plot(model1,...,modelN)
plot(model1,color1...,modelN,colorN)
plot(___,'NumberofSamples',N)
```

Description

`plot(model)` plots the nonlinearity of a nonlinear ARX model on a nonlinear ARX plot. The plot shows the nonlinearity for all outputs of the model as a function of its input regressors.

`plot(model,color)` specifies the color to use.

`plot(model1,...,modelN)` generates the plot for multiple models.

`plot(model1,color1...,modelN,colorN)` specifies the color for each model. You do not need to specify the color for all models.

`plot(___,'NumberofSamples',N)` specifies the number of samples to use to grid the regressor space on each axis. This syntax can include any of the input argument combinations in the previous syntaxes.

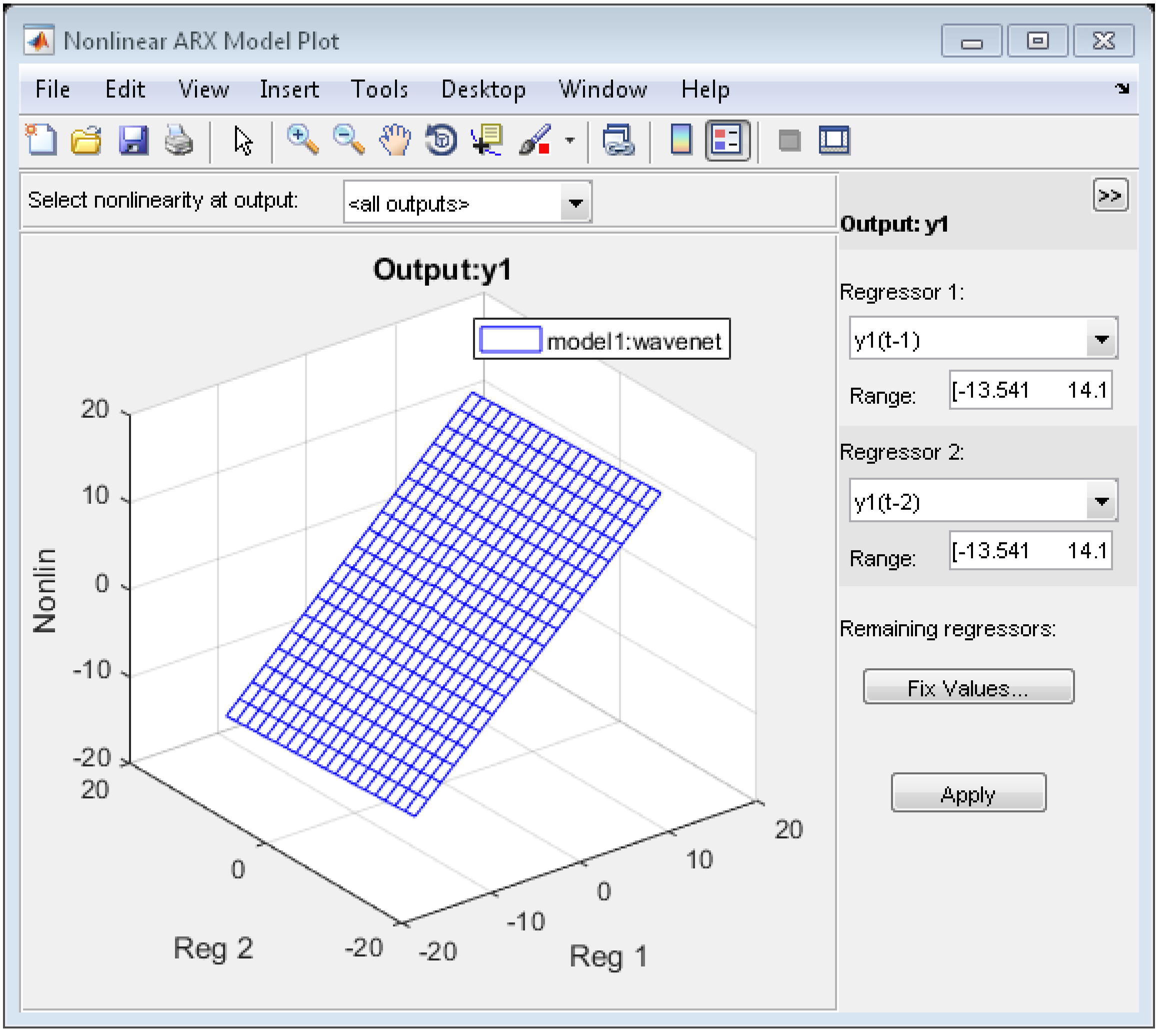
Examples

Plot Nonlinearity of a Nonlinear ARX Model

Estimate a nonlinear ARX model and plot its nonlinearity.

```
load iddata1
model1 = nlarx(z1,[4 2 1],'wave','nlr',[1:3]);
```

```
plot(model1)
```



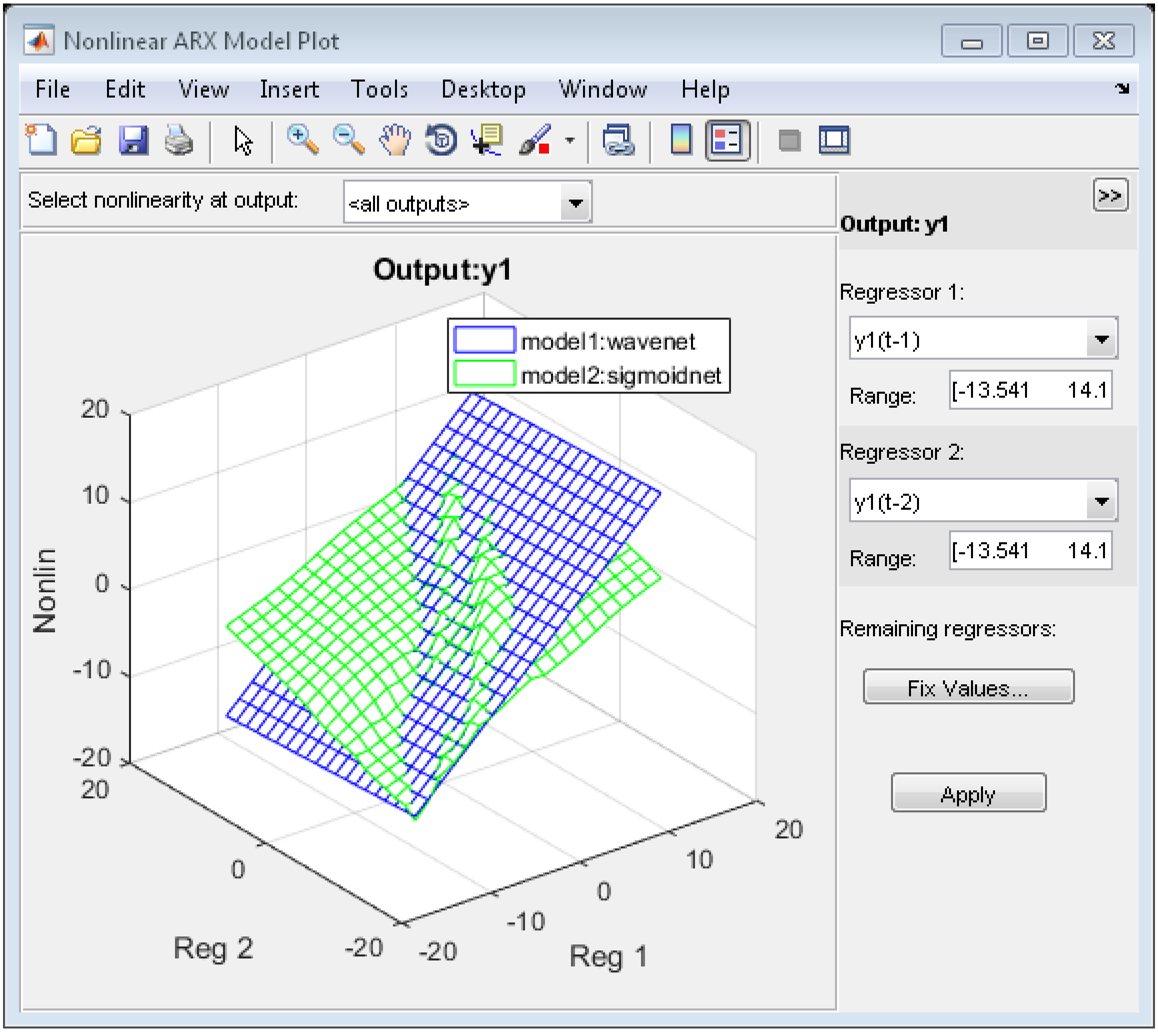
In the plot window, you can choose:

- The regressors to use on the plot axes, and specify the center points for the other regressors in the configuration panel. For multi-output models, each output is plotted separately.

- The output to view from the drop-down list located at the top of the plot.

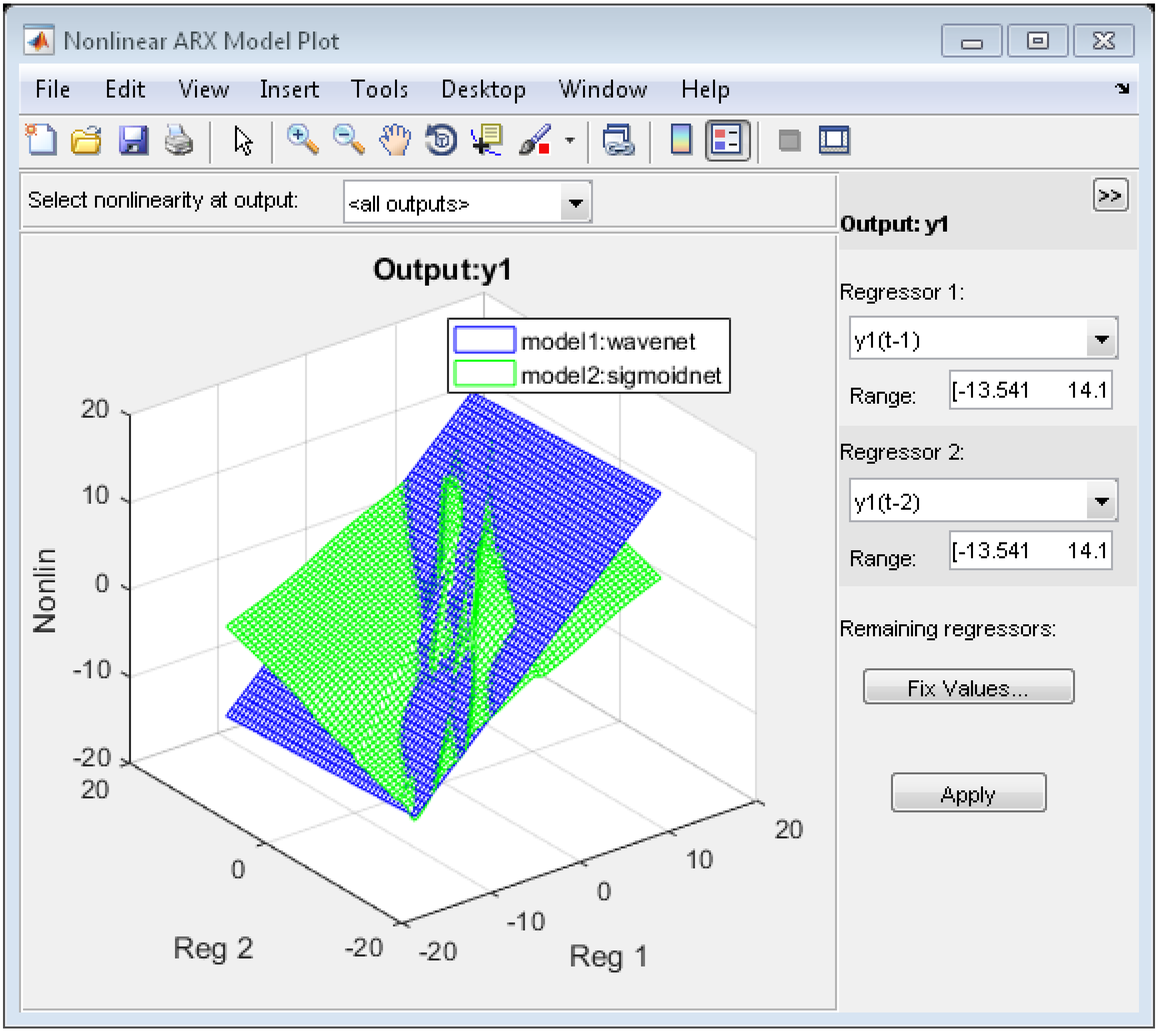
Specify Line Style for Multiple Models

```
load iddata1
model1 = nlarx(z1,[4 2 1],'wave','nlr',[1:3]);
model2 = nlarx(z1,[4 2 1],'sigmoidnet','nlr',[1:3]);
plot(model1,'b', model2, 'g')
```



Specify Number of Samples

```
load iddata1
model1 = nlarx(z1,[4 2 1],'wave','nlr',[1:3]);
model2 = nlarx(z1,[4 2 1],'sigmoidnet','nlr',[1:3]);
plot(model1,'b', model2, 'g','NumberofSamples',50)
```



Input Arguments

`model` — Estimated nonlinear ARX model

`idnlarx` model

Estimated nonlinear ARX model, specified as an `idnlarx` model object. Use `nlarx` to estimate the model.

`color` — Color to use
character vector of color name | vector of doubles

Color to use to plot the regressors, specified as one of the following:

- Character vector of color name, specified as one of the following:
  - `'b'`
  - `'y'`
  - `'m'`
  - `'c'`
  - `'r'`
  - `'g'`
  - `'w'`
- 3-element double vector of RGB values

By default, the colors are automatically chosen.

Data Types: `double` | `char`

`N` — Number of points
20 (default) | positive integer

Number of points used on the regressor axis to display the regressor samples, specified as a positive integer.

Data Types: `double`

More About

What is a Nonlinear ARX Plot?

The Nonlinear ARX plot displays the characteristics of model nonlinearities as a function of one or two regressors. The model nonlinearity (`model.Nonlinearity`) is a nonlinearity estimator function, such as `wavenet`, `sigmoidnet`, `treepartition`, and uses model regressors as its inputs. The value of the nonlinearity is plotted by projecting its response in 2 or 3-dimensional space. The plot uses one or two regressors as the plot axes for 2- or 3-D plots, respectively and a center point (cross-section location) for the other regressors.

Examining a nonlinear ARX plot can help you gain insight into which regressors have the strongest effect on the model output. Understanding the relative importance of the regressors on the output can help you decide which regressors should be included in the nonlinear function.

Furthermore, you can create several nonlinear models for the same data set using different nonlinearity estimators, such a `wavenet` network and `treepartition`, and then compare the nonlinear surfaces of these models. Agreement between nonlinear surfaces increases the confidence that these nonlinear models capture the true dynamics of the system.

In the plot window, you can choose:

- The regressors to use on the plot axes, and specify the center points for the other regressors in the configuration panel. For multi-output models, each output is plotted separately.
- The output to view from the drop-down list located at the top of the plot.

To learn more about configuring the plot, see "Tips" on page 1-987.

Tips

- To configure the nonlinear ARX plot:

  1. If your model contains multiple outputs, select the output channel in the **Select nonlinearity at output** drop-down list. Selecting the output channel displays the nonlinearity values that correspond to this output channel.
  2. If the regressor selection options are not visible, click << to expand the Nonlinear ARX Model Plot window.
  3. Select **Regressor 1** from the list of available regressors. In the **Range** field, enter the range of values to include on the plot for this regressor. The regressor values are plotted on the **Reg1** axis.
  4. Specify a second regressor for a 3-D plot by selecting one of the following types of options:

    - Select **Regressor 2** to display three axes. In the **Range** field, enter the range of values to include on the plot for this regressor. The regressor values are plotted on the **Reg2** axis.
    - Select `<none>` in the **Regressor 2** list to display only two axes.

5. To fix the values of the regressor that are not displayed, click **Fix Values**. In the Fix Regressor Values dialog box, double-click the **Value** cell to edit the constant value of the corresponding regressor. The default values are determined during model estimation. Click **OK**.
6. Click **Apply** to update the plot.

- "Structure of Nonlinear ARX Models"
- "Validating Nonlinear ARX Models"

See Also

`evaluate` | `getreg` | `idnlarx` | `nlarx`

Introduced in R2014a

idnlhw/plot

Plot input and output nonlinearity, and linear responses of Hammerstein-Wiener model

Syntax

```
plot(model)
plot(model,LineSpec)
plot(model1,...,modelN)
plot(model1,LineSpec1...,modelN,LineSpecN)

plot(___,Name,Value)
```

Description

`plot(model)` plots the input and output nonlinearity, and linear responses of a Hammerstein-Wiener model on a Hammerstein-Wiener plot. The plot shows the responses of the input and output nonlinearity, and linear blocks that represent the model.

`plot(model,LineSpec)` specifies the line style.

`plot(model1,...,modelN)` generates the plot for multiple models.

`plot(model1,LineSpec1...,modelN,LineSpecN)` specifies the line style for each model. You do not need to specify the line style for all models.

`plot(___,Name,Value)` specifies plot properties using additional options specified by one or more `Name,Value` pair arguments. This syntax can include any of the input argument combinations in the previous syntaxes.

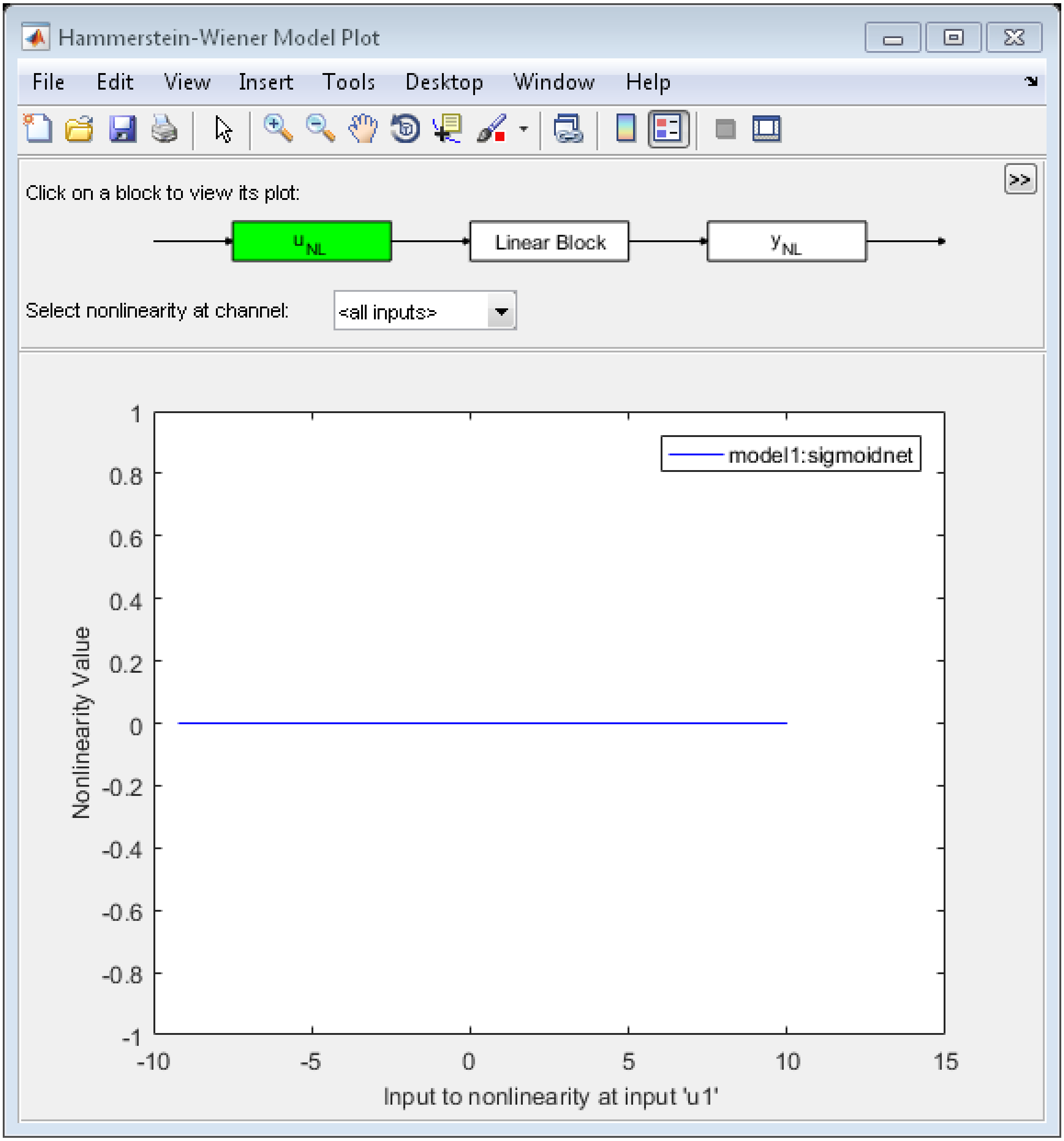
Examples

Plot Input and Output Nonlinearity and Linear Response of a Hammerstein-Wiener Model

Estimate a Hammerstein-Wiener Model and plot responses of its input and output nonlinearity and linear blocks.

```
load iddata3
```

```
model1 = nlhw(z3,[4 2 1],'sigmoidnet','deadzone');
plot(model1)
```

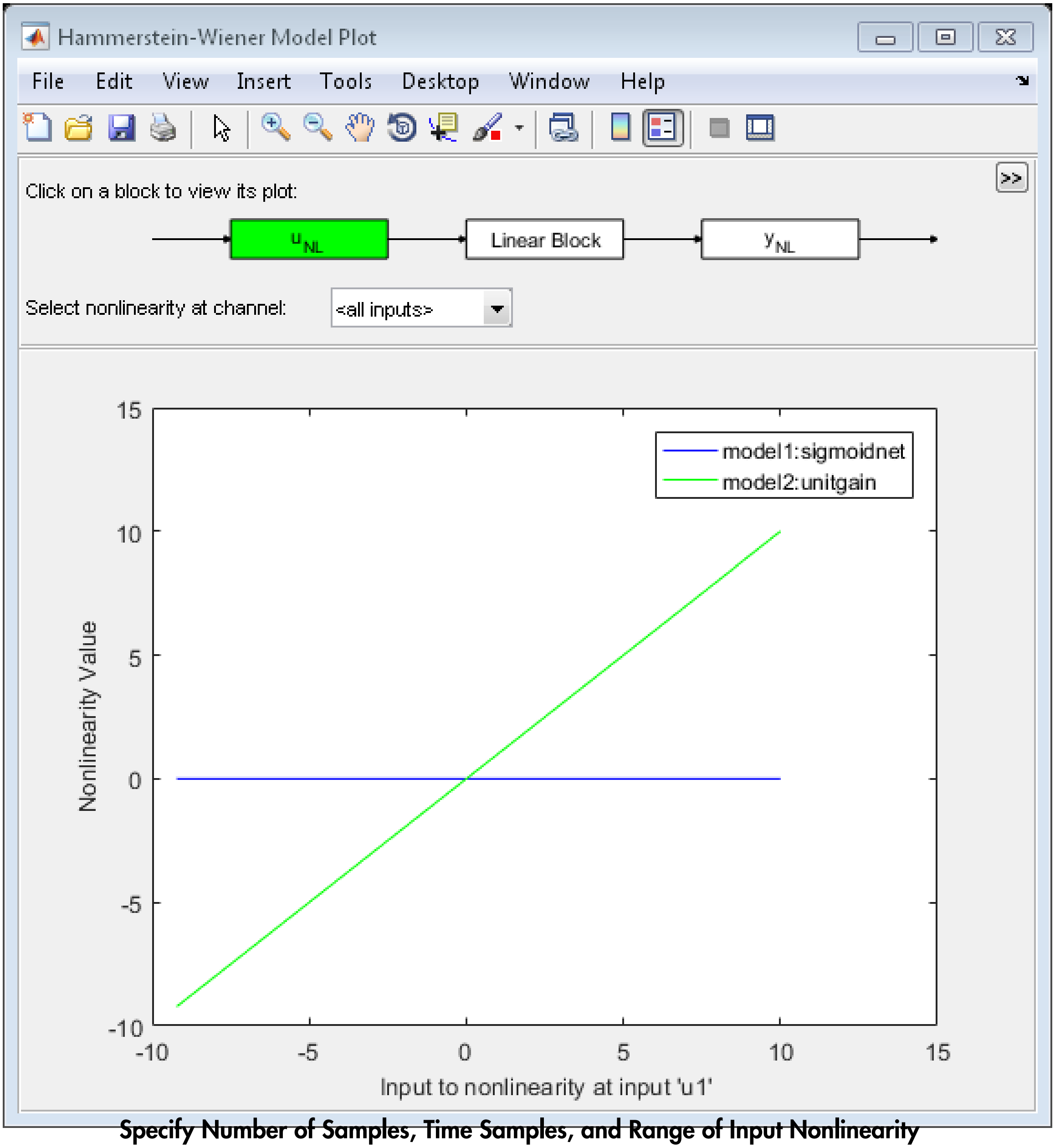

Hammerstein-Wiener Model Plot
File Edit View Insert Tools Desktop Window Help
>>
Click on a block to view its plot:
u_{NL}
Linear Block
y_{NL}
Select nonlinearity at channel:
<all inputs>
model1:sigmoidnet
Nonlinearity Value
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-10
-5
0
5
10
15
Input to nonlinearity at input 'u1'

Explore the various plots in the plot window by clicking one of the three blocks that represent the model:

- uNL - Input nonlinearity, representing the static nonlinearity at the input (`model.InputNonlinearity`) to the LinearBlock.
- Linear Block - Step, impulse,Bode and pole-zero plots of the embedded linear model (`model.LinearModel`). By default, a step plot is displayed.
- yNL - Output nonlinearity, representing the static nonlinearity at the output (`model.OutputNonlinearity`) of the Linear Block.

Specify Line Style for Multiple Hammerstein-Weiner Models

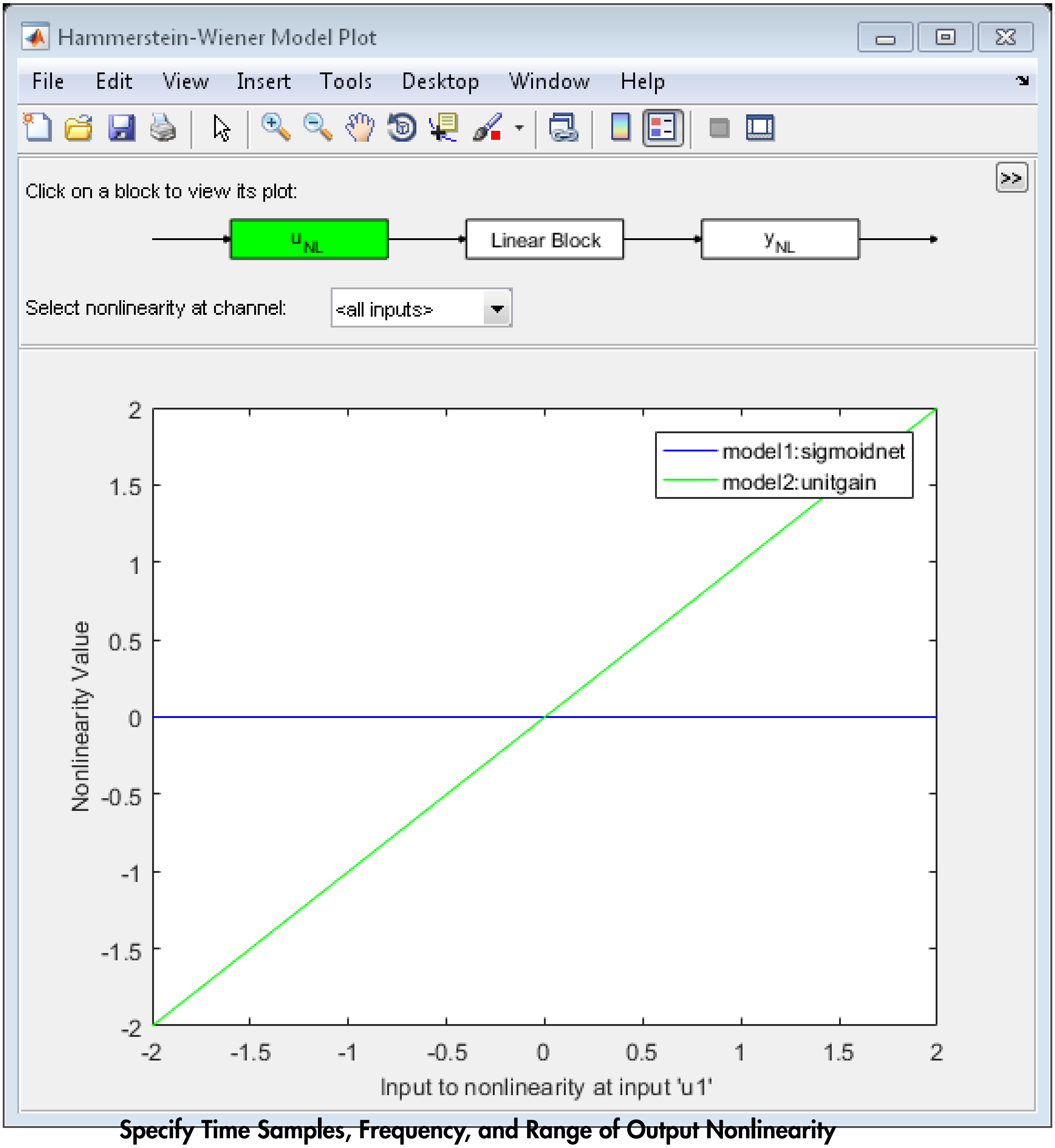
```
load iddata3
model1 = nlhw(z3,[4 2 1],'sigmoidnet','deadzone');
model2 = nlhw(z3, [4 2 1],[],'sigmoidnet');
plot(model1,'b-',model2,'g')
```



Specify Number of Samples, Time Samples, and Range of Input Nonlinearity

```
load iddata3
```

```
model1 = nlhw(z3,[4 2 1],'sigmoidnet','deadzone');
model2 = nlhw(z3, [4 2 1],[],'sigmoidnet');
plot(model1,'b-',model2,'g','NumberOfSamples',50,'time',10,'InputRange',[-2 2]);
```



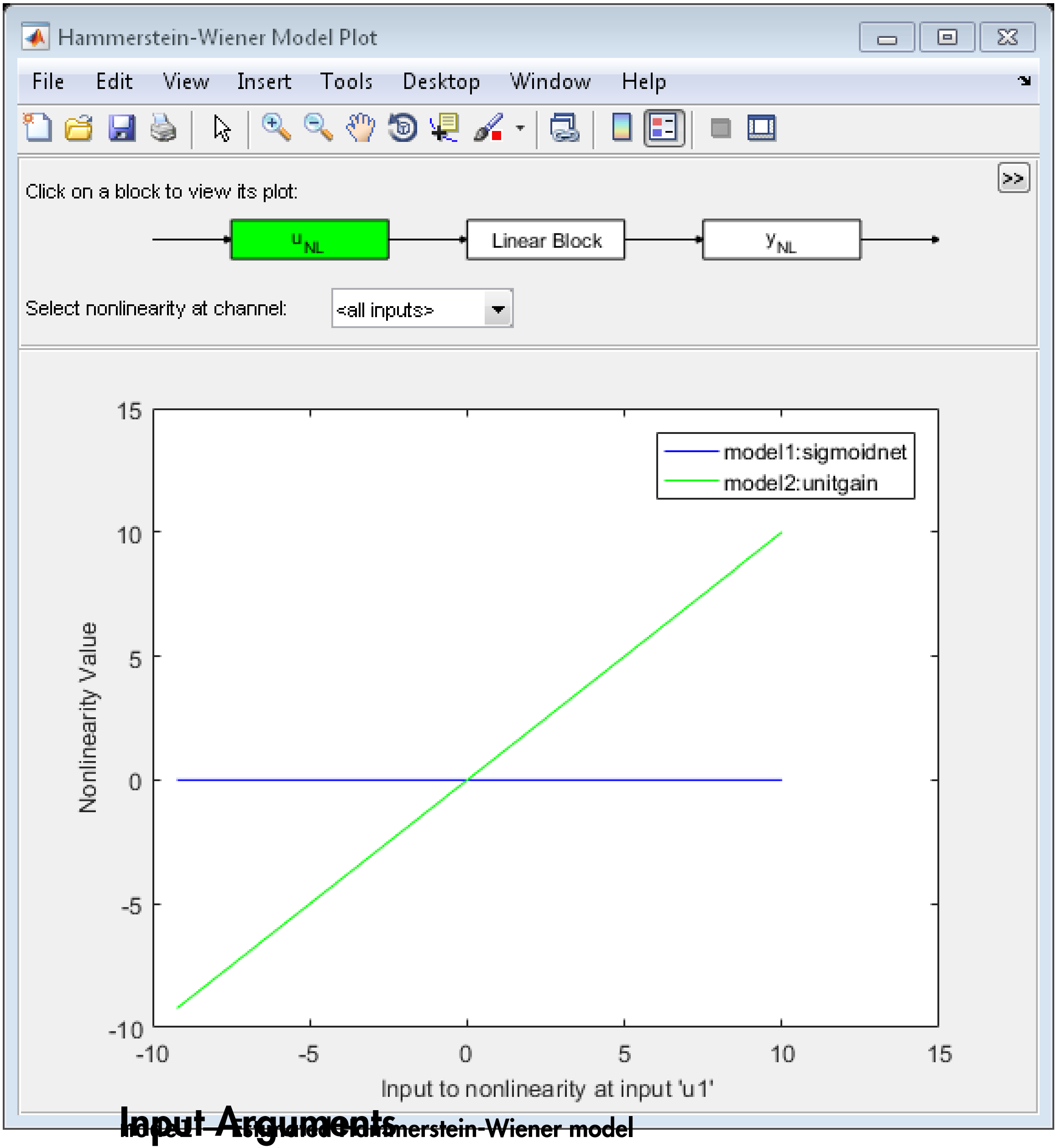
Click on a block to view its plot:

Select nonlinearity at channel: <all inputs>

Specify Time Samples, Frequency, and Range of Output Nonlinearity

```
load iddata3
```

```
model1 = nlhw(z3,[4 2 1],'sigmoidnet','deadzone');
model2 = nlhw(z3, [4 2 1],[],'sigmoidnet');
plot(model1,model2,'time',1:500,'freq',{0.01,100},'OutputRange',[0 1000]);
```



Click on a block to view its plot:

Select nonlinearity at channel: <all inputs>

Input Arguments

model — Estimated Hammerstein-Wiener model

`idnlhw` model

Estimated Hammerstein-Wiener model, specified as an `idnlhw` model object. Use `nlhw` to estimate the model.

`LineSpec` — Line style, marker symbol, and color
character vector

Line style, marker symbol, and color, specified as a character vector. `LineSpec` takes values such as `'b'`, `'b+:'`. For more information, see the `plot` reference page in the MATLAB documentation.

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example: `plot(model,'NumberofSamples',10)` specifies to use 10 data points for the input regressors.

`'NumberOfSamples'` — Number of data points to use for input regressors
100 (default) | positive integer

Number of data points to use for the input regressors when evaluating the nonlinearities at individual input or output channels, specified as a positive integer. This property does not affect the plots of the linear block.

Data Types: `double`

`'InputRange'` — Minimum and maximum regressor values for evaluating input nonlinearities
range of regressor values used during each model's estimation. (default) | positive integer | vector

Minimum and maximum regressor values to use when evaluating the nonlinearities at each input channel, specified as positive integers or `[min max]` vector, where minimum value is less than the maximum value.

You can use `'uRange'` as a shortcut name for this property.

Data Types: `double`

`'OutputRange'` — Minimum and maximum regressor values for evaluating output nonlinearities

range of regressor values used during each model's estimation (default) | positive integer | vector

Minimum and maximum regressor values to use when evaluating the nonlinearities at each output channel, specified as positive integers or `[min max]` vector, where minimum value is less than the maximum value.

You can use `'yRange'` as a shortcut name for this property.

Data Types: `double`

`'Time'` — Time samples to compute transient responses of the linear block

each model's dynamics determine the time samples used (default) | positive scalar | vector

The time samples at which the transient responses (step and impulse) of the linear block of the `idnlhw` model must be computed, specified as one of the following values:

- Positive scalar — Denotes end time for transient responses of all models. For example, 10.
- Vector of time instants — A double vector of equi-sampled values denotes the time samples at which the transient response must be computed. For example, [0:0.1:10].

This property takes the same values as the `step` command on the model.

`'Frequency'` — Frequencies at which to compute the Bode response

automatically chosen inside the Nyquist frequency range (default) | [min max] range of positive scalars | vector of positive integers

Frequencies at which to compute the Bode response, specified as one of the following values:

- `[Wmin Wmax]` range — Frequency interval between `Wmin` and `Wmax` (in units `rad/(model.TimeUnit)`) covered using logarithmically placed points.
- Vector of non-negative frequency values — Allows computation of bode response at those frequencies.

By default, the response is computed at some automatically chosen frequencies inside the Nyquist frequency range. Frequencies above Nyquist frequency (`pi/model.Ts`) are ignored.

This property takes the same values as the `bode` command on the model.

More About

What is a Hammerstein-Wiener Plot?

A Hammerstein-Wiener plot displays the characteristics of the linear block and the static nonlinearities of a Hammerstein-Wiener model.

Examining a Hammerstein-Wiener plot can help you determine whether you chose an unnecessarily complicated nonlinearity for modeling your system. For example, if you chose a piecewise-linear nonlinearity (which is very general), but the plot indicates saturation behavior, then you can estimate a new model using the simpler saturation nonlinearity instead.

For multivariable systems, you can use the Hammerstein-Wiener plot to determine whether to exclude nonlinearities for specific channels. If the nonlinearity for a specific input or output channel does not exhibit strong nonlinear behavior, you can estimate a new model after setting the nonlinearity at that channel to unit gain.

Explore the various plots in the plot window by clicking one of the three blocks that represent the model:

- u_{NL} — *Input nonlinearity*, representing the static nonlinearity at the input (`model.InputNonlinearity`) to the `Linear Block`.
- `Linear Block` — Step, impulse, Bode and pole-zero plots of the embedded linear model (`model.LinearModel`). By default, a step plot is displayed.
- y_{NL} — *Output nonlinearity*, representing the static nonlinearity at the output (`model.OutputNonlinearity`) of the `Linear Block`.

To learn more about how to configure the linear and nonlinear blocks plots, see “Tips” on page 1-1000.

Tips

- The Hammerstein-Wiener model can contain up to two nonlinear blocks. The nonlinearity at the input to the Linear Block is labeled u_{NL} and is called the *input nonlinearity*. The nonlinearity at the output of the Linear Block is labeled y_{NL} and is called the *output nonlinearity*.

  To configure the nonlinear block characteristics plot:

1. If the top pane is not visible, click << to expand the Hammerstein-Wiener Model Plot window.
2. Select the nonlinear block you want to plot:

   - To plot the response of the input nonlinearity function, click the u_{NL} block.
   - To plot the response of the output nonlinearity function, click the y_{NL} block.

   The selected block is highlighted in green.

   Note: The input to the output nonlinearity block y_{NL} is the output from the Linear Block and not the measured input data.

   ---
3. If your model contains multiple inputs or outputs, select the channel in the **Select nonlinearity at channel** list. Selecting the channel updates the plot and displays the nonlinearity values versus the corresponding input to this nonlinear block.
4. Click **Apply** to update the plot.

- The Hammerstein-Wiener model contains one Linear Block that represents the embedded linear model.

  To configure the linear block plot:

  1. If the top pane is not visible, click << to expand the Hammerstein-Wiener Model Plot window.
  2. Click the Linear Block to select it. The Linear Block is highlighted in green.
  3. In the **Select I/O pair** list, select the input and output data pair for which to view the response.
  4. In the **Choose plot type** list, select the linear plot from the following options:

     - `Step`
     - `Impulse`
     - `Bode`
     - `Pole-Zero Map`

- “Structure of Hammerstein-Wiener Models”

- "Validating Hammerstein-Wiener Models"

See Also

`idnlhw` | `nlhw`

Introduced in R2014a

pole

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.

Limitations

Multiple poles are numerically sensitive and cannot be computed to high accuracy. A pole λ with multiplicity m typically gives rise to a cluster of computed poles distributed on a circle with center λ and radius of order

$$\rho \approx \varepsilon^{1/m}$$

where ε is the relative machine precision (`eps`).

More About

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.

See Also

`damp` | `esort` | `dsort` | `pzmap` | `zero`

Introduced in R2012a

polydata

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`:

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

  $u(t)$ 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

Extract Polynomial Coefficients and Uncertainties from Identified 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 sample time of 0.1 s. Therefore, `sys` is a two-input, two-output discrete-time `idpoly` model of the form:

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

The inputs to `polyest` set the order of each polynomial in `sys`.

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. The dimensions of the cell arrays are determined by the input and output dimensions of `sys`. For example, `A` is a 2-by-2 cell array because `sys` has two inputs and 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   -0.8825   -0.2030    0.4364
```

For discrete-time `sys`, the coefficients are arranged in order of increasing powers of q^{-1}. Therefore, `A{2,2}` corresponds to the polynomial $1 - 0.8682q^{-1} - 0.2244q^{-2} + 0.4467q^{-3}$.

The dimensions of `dA` match those of `A`. Each entry in `dA` gives the standard deviation of the corresponding estimated polynomial coefficient in `A`. For example, examine the uncertainties of the second diagonal entry in `A`.

```
dA{2,2}
```

```
ans =

         0    0.2849    0.4269    0.2056
```

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

`iddata` | `idpoly` | `idssdata` | `polyest` | `tfdata` | `zpkdata`

Introduced before R2006a

polyest

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( ___ , 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

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

$A(q)$, $B(q)$, $F(q)$, $C(q)$ and $D(q)$ are polynomial matrices. $u(t)$ is the input, and `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 linear system `init_sys` to configure the initial parameterization.

`sys = polyest( ___ , opt)` estimates a polynomial model using the option set, `opt`, to specify estimation behavior.

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 *Ny*-by-*Ny* matrix of nonnegative integers. *Ny* is the number of outputs, and *Nu* 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 *Ny*-by-*Nu* matrix of nonnegative integers. *Ny* is the number of outputs, and *Nu* is the number of inputs.

`nc`

Order of the polynomial $C(q)$.

`nc` is a column vector of nonnegative integers of length *Ny*. *Ny* 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 *Ny*. *Ny* 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 *Ny*-by-*Nu* matrix of nonnegative integers. *Ny* is the number of outputs, and *Nu* is the number of inputs.

`nk`

Input delay in number of samples, expressed as fixed leading zeros of the *B* polynomial.

`nk` is an *Ny*-by-*Nu* matrix of nonnegative integers.

`nk` must be zero when estimating a continuous-time model.

`opt`

Estimation options.

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

- Estimation objective
- Handling of initial conditions
- Numerical search method to be used in estimation

`init_sys`

Linear system that configures the initial parameterization of `sys`.

You obtain `init_sys` by either performing an estimation using measured data or by direct construction.

If `init_sys` is an `idpoly` model, `polyest` 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 $A(q)$, $B(q)$, $F(q)$, $C(q)$, and $D(q)$. For example:

- To specify an initial guess for the $A(q)$ term of `init_sys`, set `init_sys.Structure.A.Value` as the initial guess.
- To specify constraints for 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 `init_sys` is not an `idpoly` model, the software first converts `init_sys` to a polynomial model. `polyest` uses the parameters of the resulting model as the initial guess for estimation.

If `opt` is not specified, and `init_sys` is 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. `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 in integer multiples of the sample time, `Ts`.

For a MIMO system with `Ny` outputs and `Nu` inputs, set `IODelay` to a `Ny`-by-`Nu` array. Each entry of this array is a numerical value that represents 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 delay for each input channel, specified as a scalar value or numeric vector. 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 sample time `Ts`. For example, `InputDelay = 3` means a delay of three sample times.

For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector. Each entry of this vector is a numerical value that represents 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

`'IntegrateNoise'`

Logical vector specifying integrators in the noise channel.

`IntegrateNoise` is a logical vector of length Ny, where Ny 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`

Polynomial model, returned as an `idpoly` model. This model is created using the specified model orders, delays, and estimation options.

If `data.Ts` is zero, `sys` is a continuous-time model representing:

$$Y(s) = \frac{B(s)}{F(s)}U(s) + E(s)$$

$Y(s)$, $U(s)$ and $E(s)$ are the Laplace transforms of the time-domain signals $y(t)$, $u(t)$ and $e(t)$, respectively.

Information about the estimation results and options used is stored in the `Report` property of the model. `Report` have the following fields:

Report Field	**Description**
`Status`	Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.
`Method`	Estimation command used.
`InitialCo`	Handling of initial conditions during model estimation, returned as one of the following values: • `'zero'` — The initial conditions were set to zero. • `'estimate'` — The initial conditions were treated as independent estimation parameters. • `'backcast'` — The initial conditions were estimated using the best least squares fit. This field is especially useful to view how the initial conditions were handled when the `InitialCondition` option in the estimation option set is `'auto'`.
`Fit`	Quantitative assessment of the estimation, returned as a structure. See “Loss Function and Model Quality Metrics” for more information on these quality metrics. The structure has the following fields:

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>FitPerc</code></td><td>Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as a percentage.</td></tr>
<tr><td><code>LossFcn</code></td><td>Value of the loss function when the estimation completes.</td></tr>
<tr><td><code>MSE</code></td><td>Mean squared error (MSE) measure of how well the response of the model fits the estimation data.</td></tr>
<tr><td><code>FPE</code></td><td>Final prediction error for the model.</td></tr>
<tr><td><code>AIC</code></td><td>Raw Akaike Information Criteria (AIC) measure of model quality.</td></tr>
<tr><td><code>AICc</code></td><td>Small sample-size corrected AIC.</td></tr>
<tr><td><code>nAIC</code></td><td>Normalized AIC.</td></tr>
<tr><td><code>BIC</code></td><td>Bayesian Information Criteria (BIC).</td></tr>
</table>
</td></tr>
<tr><td><code>Parameter</code></td><td>Estimated values of model parameters.</td></tr>
<tr><td><code>OptionsUs</code></td><td>Option set used for estimation. If no custom options were configured, this is a set of default options. See <code>polyestOptions</code> for more information.</td></tr>
<tr><td><code>RandState</code></td><td>State of the random number stream at the start of estimation. Empty, <code>[]</code>, if randomization was not used during estimation. For more information, see <code>rng</code> in the MATLAB documentation.</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation, returned as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
<tr><td><code>Type</code></td><td>Data type.</td></tr>
<tr><td><code>Length</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time.</td></tr>
<tr><td><code>InterSam</code></td><td>Input intersample behavior, returned as one of the following values:
<code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
<code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
<code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.</td></tr>
<tr><td><code>InputOff</code></td><td>Offset removed from time-domain input data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
<tr><td><code>OutputOf</code></td><td>Offset removed from time-domain output data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
</table></td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Terminati</code></td><td>Termination conditions for the iterative search used for prediction error minimization. Structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>WhyStop</code></td><td>Reason for terminating the numerical search.</td></tr>
<tr><td><code>Iteratio</code></td><td>Number of search iterations performed by the estimation algorithm.</td></tr>
<tr><td><code>FirstOrd</code></td><td>∞ -norm of the gradient search vector when the search algorithm terminates.</td></tr>
<tr><td><code>FcnCount</code></td><td>Number of times the objective function was called.</td></tr>
<tr><td><code>UpdateNo</code></td><td>Norm of the gradient search vector in the last iteration. Omitted when the search method is <code>'lsqnonlin'</code>.</td></tr>
<tr><td><code>LastImpr</code></td><td>Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is <code>'lsqnonlin'</code>.</td></tr>
<tr><td><code>Algorith</code></td><td>Algorithm used by <code>'lsqnonlin'</code> search method. Omitted when other search methods are used.</td></tr>
</table>
For estimation methods that do not require numerical search optimization, the <code>Termination</code> field is omitted.</td></tr>
</table>

For more information on using `Report`, see “Estimation Report”.

Examples

Estimate Polynomial Model with Redundant Parameterization

Estimate a model with redundant parameterization. That is, a model with all polynomials (A, B, C, D, and F) active.

Load estimation data.

```
load iddata2 z2;
```

Specify the model orders and delays.

```
na = 2;
nb = 2;
nc = 3;
nd = 3;
nf = 2;
nk = 1;
```

Estimate the model.

```
sys = polyest(z2,[na nb nc nd nf nk]);
```

Estimate Polynomial Model Using Regularization

Estimate a regularized polynomial model by converting a regularized ARX model.

Load estimation data.

```
load regularizationExampleData.mat m0simdata;
```

Estimate an unregularized polynomial model of order 20.

```
m1 = polyest(m0simdata(1:150),[0 20 20 20 20 1]);
```

Estimate a regularized polynomial model of the same order. Determine the Lambda value by trial and error.

```
opt = polyestOptions;
opt.Regularization.Lambda = 1;
m2 = polyest(m0simdata(1:150),[0 20 20 20 20 1],opt);
```

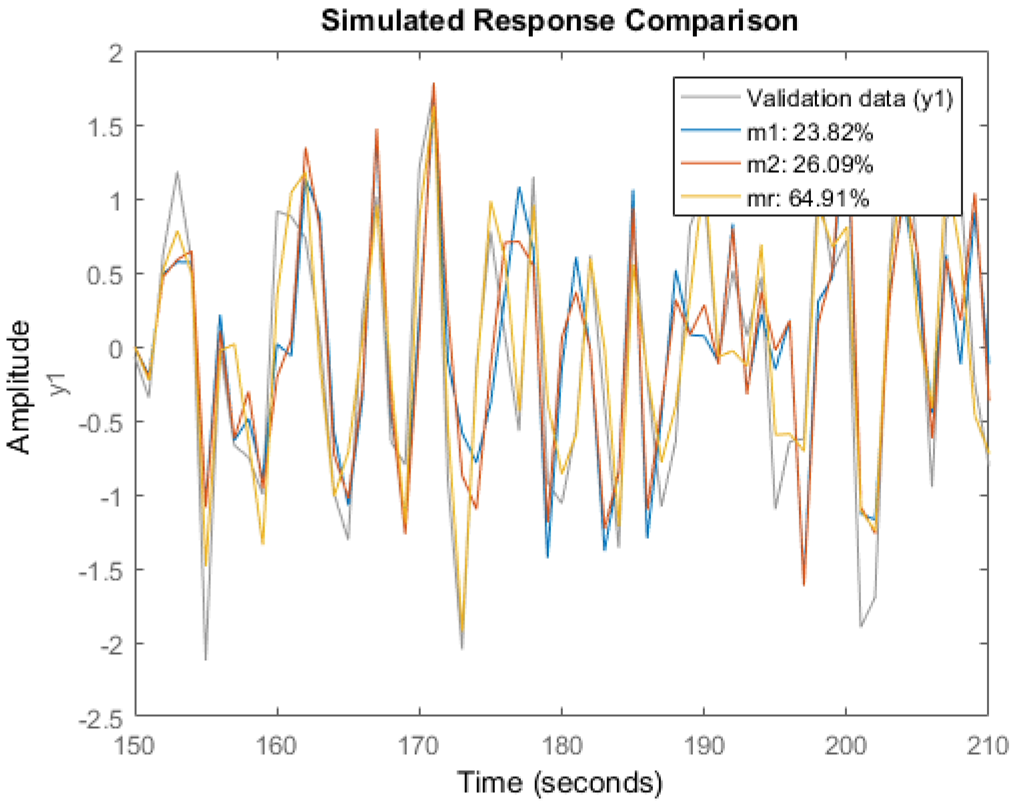
Obtain a lower-order polynomial model by converting a regularized ARX model and reducing its order. Use `arxregul` to determine the regularization parameters.

```
[L,R] = arxRegul(m0simdata(1:150),[30 30 1]);
opt1 = arxOptions;
opt1.Regularization.Lambda = L;
opt1.Regularization.R = R;
m0 = arx(m0simdata(1:150),[30 30 1],opt1);
mr = idpoly(balred(idss(m0),7));
```

Compare the model outputs against the data.

```
opt2 = compareOptions('InitialCondition','z');
```

```
compare(m0simdata(150:end),m1,m2,mr,opt2);
```



Estimate ARIMAX model

Load input/output data and create cumulative sum input and output signals for estimation.

```
load iddata1 z1
data = iddata(cumsum(z1.y),cumsum(z1.u),z1.Ts,'InterSample','foh');
```

Specify the model polynomial orders. Set the orders of the inactive polynomials, D and F, to `0`.

```
na = 2;
```

```
nb = 2;
nc = 2;
nd = 0;
nf = 0;
nk = 1;
```

Identify an ARIMAX model by setting the `'IntegrateNoise'` option to `true`.

```
sys = polyest(data,[na nb nc nd nf nk],'IntegrateNoise',true);
```

Estimate Multi-Output ARMAX Model

Estimate a multi-output ARMAX model for a multi-input, multi-output data set.

Load estimation data.

```
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.

Specify the model orders and delays. The `F` and `D` polynomials are inactive.

```
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];
```

Estimate the model.

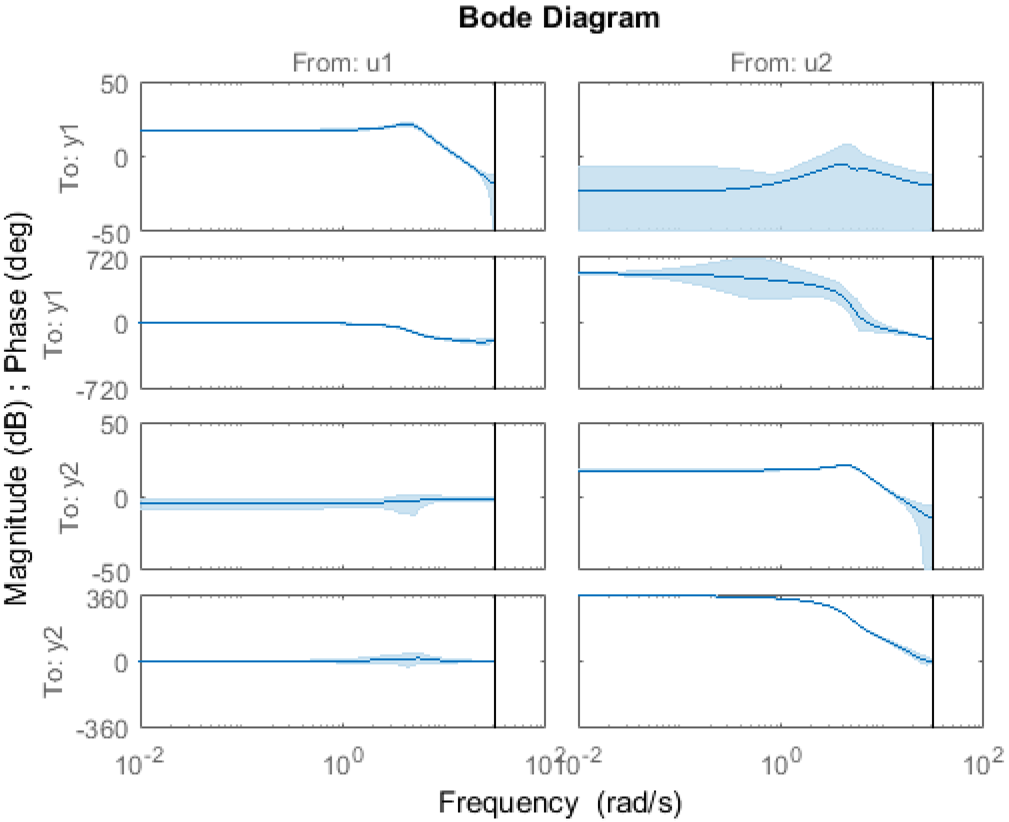
```
sys = polyest(data,[na nb nc nd nf nk]);
```

In the estimated ARMAX model, the cross terms, which model the effect of the first input on the second output and vice versa, are negligible. If you assigned higher orders to those dynamics, their estimation would show a high level of uncertainty.

Analyze the results.

```
h = bodeplot(sys);
```

```
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.

More About

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 estimate an Output-Error (OE) model by specifying `na`, `nc` and `nd` as zero.

  Alternatively, you can use a dedicated estimating function for the simplified model structure. Linear polynomial estimation functions include `oe`, `bj`, `arx` and `armax`.

- “Regularized Estimates of Model Parameters”

See Also

`ar` | `armax` | `arx` | `bj` | `forecast` | `iddata` | `idpoly` | `oe` | `pem` | `polyestOptions` | `procest` | `ssest` | `tfest`

Introduced in R2012a

polyestOptions

Option set for `polyest`

Syntax

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

Description

`opt = polyestOptions` creates the default option 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. `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'` — Handling of initial conditions
`'auto'` (default) | `'zero'` | `'estimate'` | `'backcast'`

Handling of initial conditions during estimation, specified as 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.

`'Focus'` — Error to be minimized

`'prediction'` (default) | `'simulation'`

Error to be minimized in the loss function during estimation, specified as the comma-separated pair consisting of `'Focus'` and one of the following values:

- `'prediction'` — The one-step ahead prediction error between measured and predicted outputs is minimized during estimation. As a result, the estimation focuses on producing a good predictor model.
- `'simulation'` — The simulation error between measured and simulated outputs is minimized during estimation. As a result, the estimation focuses on making a good fit for simulation of model response with the current inputs.

The `Focus` option can be interpreted as a weighting filter in the loss function. For more information, see “Loss Function and Model Quality Metrics”.

`'WeightingFilter'` — Weighting prefilter

`[ ]` (default) | vector | matrix | cell array | linear system

Weighting prefilter applied to the loss function to be minimized during estimation. To understand the effect of `WeightingFilter` on the loss function, see “Loss Function and Model Quality Metrics”.

Specify `WeightingFilter` as one of the following values:

- `[ ]` — No weighting prefilter is used.
- Passbands — Specify a row vector or matrix containing frequency values that define desired passbands. You select a frequency band where the fit between estimated model and estimation data is optimized. For example, `[wl,wh]` where `wl` and `wh` represent lower and upper limits of a passband. For a matrix with several rows defining frequency passbands, `[w1l,w1h;w2l,w2h;w3l,w3h;...]`, the estimation algorithm uses the union of the frequency ranges to define the estimation passband.

  Passbands are expressed in `rad/TimeUnit` for time-domain data and in `FrequencyUnit` for frequency-domain data, where `TimeUnit` and `FrequencyUnit` are the time and frequency units of the estimation data.
- SISO filter — Specify a single-input-single-output (SISO) linear filter in one of the following ways:

  - A SISO LTI model
  - `{A,B,C,D}` format, which specifies the state-space matrices of a filter with the same sample time as estimation data.
  - `{numerator,denominator}` format, which specifies the numerator and denominator of the filter as a transfer function with same sample time as estimation data.

    This option calculates the weighting function as a product of the filter and the input spectrum to estimate the transfer function.
- Weighting vector — Applicable for frequency-domain data only. Specify a column vector of weights. This vector must have the same length as 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.

`'EnforceStability'` — Control whether to enforce stability of model

`false` (default) | `true`

Control whether to enforce stability of estimated model, specified as the comma-separated pair consisting of `'EnforceStability'` and either `true` or `false`.

This option is not available for multi-output models with a non-diagonal *A* polynomial array.

Data Types: `logical`

`'EstCovar'` — Control whether to generate parameter covariance data

`true` (default) | `false`

Controls whether parameter covariance data is generated, specified as `true` or `false`.

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

`'Display'` — Specify whether to display the estimation progress

`'off'` (default) | `'on'`

Specify whether to display the estimation progress, specified as one of the following values:

- `'on'` — Information on model structure and estimation results are displayed in a progress-viewer window.

- `'off'` — No progress or results information is displayed.

`'InputOffset'` — Removal of offset from time-domain input data during estimation

[ ] (default) | vector of positive integers | matrix

Removal of offset from time-domain input data during estimation, specified as the comma-separated pair consisting of `'InputOffset'` and one of the following:

- A column vector of positive integers of length *Nu*, where *Nu* is the number of inputs.
- `[]` — Indicates no offset.
- *Nu*-by-*Ne* matrix — For multi-experiment data, specify `InputOffset` as an *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.

`'OutputOffset'` — Removal of offset from time-domain output data during estimation

[ ] (default) | vector | matrix

Removal of offset from time-domain output data during estimation, specified as the comma-separated pair consisting of `'OutputOffset'` and one of the following:

- A column vector of length *Ny*, where *Ny* is the number of outputs.
- `[]` — Indicates no offset.
- *Ny*-by-*Ne* matrix — For multi-experiment 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.

`'Regularization'` — Options for regularized estimation of model parameters

structure

Options for regularized estimation of model parameters. For more information on regularization, see “Regularized Estimates of Model Parameters”.

`Regularization` is a structure with the following fields:

- `Lambda` — Constant that determines the bias versus variance tradeoff.

  Specify a positive scalar to add the regularization term to the estimation cost.

The default value of zero implies no regularization.

Default: 0

- `R` — Weighting matrix.

  Specify a vector of nonnegative numbers or a square positive semi-definite matrix. The length must be equal to the number of free parameters of the model.

  For black-box models, using the default value is recommended. For structured and grey-box models, you can also specify a vector of `np` positive numbers such that each entry denotes the confidence in the value of the associated parameter.

  The default value of 1 implies a value of `eye(npfree)`, where `npfree` is the number of free parameters.

  Default: 1
- `Nominal` — The nominal value towards which the free parameters are pulled during estimation.

  The default value of zero implies that the parameter values are pulled towards zero. If you are refining a model, you can set the value to `'model'` to pull the parameters towards the parameter values of the initial model. The initial parameter values must be finite for this setting to work.

  Default: 0

`'SearchMethod'` — Search method used for iterative parameter estimation

`'auto'` (default) | `'gn'` | `'gna'` | `'lm'` | `'lsqnonlin'` | `'grad'`

Search method used for iterative parameter estimation, specified as one of the following values:

- `'gn'` — The subspace Gauss-Newton direction. Singular values of the Jacobian matrix less than `GnPinvConst*eps*max(size(J))*norm(J)` are discarded when computing the search direction. J is the Jacobian matrix. The Hessian matrix is approximated by J^TJ. 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 Ninne. Eigenvalues less than `gamma*max(sv)` of the Hessian are ignored, 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). *gamma* is increased by the factor `LMStep` each time the search fails to find a lower value of the criterion in less than 5 bisections. *gamma* is decreased by a factor of `2*LMStep` each time a search is successful without any bisections.

- `'lm'` — Uses the Levenberg-Marquardt method. The next parameter value is `-pinv(H+d*I)*grad` from the previous one. *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. This search method can handle only the Trace criterion.
- `'grad'` — The steepest descent gradient search method.
- `'auto'` — The algorithm chooses one of the preceding options. The descent direction is calculated using `'gn'`, `'gna'`, `'lm'`, and `'grad'` successively at each iteration. The iterations continue until a sufficient reduction in error is achieved.

`'SearchOption'` — Option set for the search algorithm

search option set

Option set for the search algorithm with fields that depend on the value of `SearchMethod`.

SearchOption structure when SearchMethod is specified as 'gn', 'gna', 'lm', 'grad', or 'auto'

Field Name	Description
`Toleran`	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 stop. The estimate of the expected loss-function improvement at the next iteration is based on the Gauss-Newton vector computed for the current parameter value. **Default:** `0.01`
`MaxIter`	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 start-up procedure.

Field Name	Description
	Use `sys.Report.Termination.Iterations` to get the actual number of iterations during an estimation, where *sys* is an `idtf` model. **Default:** 20

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td><code>Advance</code></td><td>
<p>Advanced search settings.</p>
<p>Specified as a structure with the following fields:</p>
<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td><code>GnPinvCon</code></td><td><p>Singular values of the Jacobian matrix that are smaller than <code>GnPinvConst*max(size(J)*norm(J)*eps)</code> are discarded when computing the search direction. Applicable when <code>SearchMethod</code> is <code>'gn'</code>.</p><p><code>GnPinvConst</code> must be a positive, real value.</p><p>Default: <code>10000</code></p></td></tr>
<tr><td><code>InitGnaTo</code></td><td><p>Initial value of <i>gamma</i>. Applicable when <code>SearchMethod</code> is <code>'gna'</code>.</p><p>Default: <code>0.0001</code></p></td></tr>
<tr><td><code>LMStartVa</code></td><td><p>Starting value of search-direction length <i>d</i> in the Levenberg-Marquardt method. Applicable when <code>SearchMethod</code> is <code>'lm'</code>.</p><p>Default: <code>0.001</code></p></td></tr>
<tr><td><code>LMStep</code></td><td><p>Size of the Levenberg-Marquardt step. The next value of the search-direction length <i>d</i> in the Levenberg-Marquardt method is <code>LMStep</code> times the previous one. Applicable when <code>SearchMethod</code> is <code>'lm'</code>.</p><p>Default: <code>2</code></p></td></tr>
<tr><td><code>MaxBisect</code></td><td><p>Maximum number of bisections used by the line search along the search direction.</p><p>Default: <code>25</code></p></td></tr>
<tr><td><code>MaxFunEva</code></td><td><p>Iterations stop if the number of calls to the model file exceeds this value.</p><p><code>MaxFunEvals</code> must be a positive, integer value.</p></td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>Default: <code>Inf</code></td></tr>
<tr><td><code>MinParCha</code></td><td>Smallest parameter update allowed per iteration.
<code>MinParChange</code> must be a positive, real value.
Default: <code>0</code></td></tr>
<tr><td><code>RelImprov</code></td><td>Iterations stop if the relative improvement of the criterion function is less than <code>RelImprovement</code>.
<code>RelImprovement</code> must be a positive, integer value.
Default: <code>0</code></td></tr>
<tr><td><code>StepReduc</code></td><td>Suggested parameter update is reduced by the factor <code>StepReduction</code> after each try. This reduction continues until either <code>MaxBisections</code> tries are completed or a lower value of the criterion function is obtained.
<code>StepReduction</code> must be a positive, real value that is greater than 1.
Default: <code>2</code></td></tr>
</table>
</td></tr>
</table>

SearchOption structure when SearchMethod is specified as 'lsqnonlin'

Field Name	Description
`TolFun`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values. The value of `TolFun` is the same as that of `opt.SearchOption.Advanced.TolFun`. **Default:** `1e-5`
`TolX`	Termination tolerance on the estimated parameter values.

Field Name	Description
	The value of `TolX` is the same as that of `opt.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 `opt.SearchOption.Advanced.MaxIter`. **Default:** `20`
`Advance`	Options set for `lsqnonlin`. For more information, see the Optimization Options table in “Optimization Options”. Use `optimset('lsqnonlin')` to create an options set for `lsqnonlin`, and then modify it to specify its various options.

`'Advanced'` — Additional advanced options

structure

Additional advanced options, specified as 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 loss function. The standard deviation is estimated robustly as the median of the absolute deviations from the median of the prediction errors, divided by `0.7`. For more information on robust norm choices, see section 15.2 of [2].

  `ErrorThreshold = 0` disables robustification and leads to a purely quadratic loss function. When estimating with frequency-domain data, the software sets `ErrorThreshold` to zero. For time-domain data that contains outliers, try setting `ErrorThreshold` to `1.6`.

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 condition.

  The initial condition is estimated when

  $$\frac{\left\| y_{p,z} - y_{meas} \right\|}{\left\| y_{p,e} - y_{meas} \right\|} > \text{AutoInitThreshold}$$

  - y_{meas} is the measured output.
  - $y_{p,z}$ is the predicted output of a model estimated using zero initial states.
  - $y_{p,e}$ is the predicted output of a model estimated using estimated initial states.

  Applicable when `InitialCondition` is `'auto'`.

  Default: `1.05`

Output Arguments

`opt` — Options set for `polyest`
`polyestOptions` option set

Option set for `polyest`, returned as an `polyestOptions` option set.

Examples

Create Default Option Set for Polynomial Estimation

```
opt = polyestOptions;
```

Specify Options for Polynomial Estimation

Create an option set for `polyest` where you enforce model stability and set the `Display` to `'on'`.

```
opt = polyestOptions('EnforceStability',true,'Display','on');
```

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

```
opt = polyestOptions;
opt.EnforceStability = true;
opt.Display = 'on';
```

More About

- “Loss Function and Model Quality Metrics”

References

[1] 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.

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

See Also

`polyest`

Introduced in R2012a

polyreg

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 $y(t-1)*u(\mathrm{t}-1)$.

`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 standards regressors) when *`CrossTermVal`* is `'on'`. By default, *`CrossTermVal`* is `'off'`.

Examples

Create Polynomial Regressors Up To Power 2

Estimate a nonlinear ARX model with $na = 2$, $nb = 2$, and $nk = 1$, and nonlinearity estimator `wavenet`.

```
load iddata1
m = nlarx(z1,[2 2 1]);
```

Create polynomial regressors.

```
R = polyreg(m);
```

Estimate the model.

```
m = nlarx(z1,[2 2 1],'wavenet','CustomReg',R);
```

View all model regressors (standard and custom).

```
getreg(m)
```

```
Regressors:
    y1(t-1)
    y1(t-2)
    u1(t-1)
    u1(t-2)
    y1(t-1).^2
    y1(t-2).^2
    u1(t-1).^2
    u1(t-2).^2
```

Create Polynomial Regressors Up To Power 3

Estimate a nonlinear ARX model with $na = 2$, $nb = 1$, and $nk = 1$, and nonlinearity estimator `wavenet`.

```
load iddata1
m = nlarx(z1,[2 1 1]);
```

Create polynomial regressors.

```
R = polyreg(m,'MaxPower',3,'CrossTerm','on')
```

```
16x1  array of Custom Regressors with fields: Function, Arguments, Delays, Vectorized.
```

If the model `m` has three standard regressors `a`, `b` and `c`, then `R` includes the terms a^2, b^2, c^2, ab, ac, bc, a^2b, a^2c, ab^2, abc, ac^2, b^2c, bc^2, a^3, b^3, and c^3.

Estimate the model.

```
m = nlarx(z1,[2 1 1],'wavenet','CustomReg',R);
```

More About

- “Identifying Nonlinear ARX Models”

See Also

`addreg` | `customreg` | `getreg` | `idnlarx` | `nlarx`

Introduced in R2007a

poly1d

Class representing single-variable polynomial nonlinear estimator for Hammerstein-Wiener models

Syntax

```
t=poly1d('Degree',n)
t=poly1d('Coefficients',C)
t=poly1d(n)
```

Description

`poly1d` is an object that stores the single-variable polynomial nonlinear estimator for Hammerstein-Wiener models.

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

`t=poly1d('Degree',n)` creates a polynomial nonlinearity estimator object of `n`th degree.

`t=poly1d('Coefficients',C)` creates a polynomial nonlinearity estimator object with coefficients `C`.

`t=poly1d(n)` a polynomial nonlinearity estimator object of `n`th degree.

Use `evaluate(p,x)` to compute the value of the function defined by the `poly1d` object `p` at `x`.

poly1d Properties

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

```
% List all property values
get(p)
% Get value of Coefficients property
```

```
p.Coefficients
```

Property Name	Description
`Degree`	Positive integer specifies the degree of the polynomial Default=`1`. For example: `poly1d('Degree',3)`
`Coefficients`	1-*by*-`(n+1)` matrix containing the polynomial coefficients.

Examples

Use `poly1s` to specify the single-variable polynomial nonlinearity estimator in Hammerstein-Wiener models. For example:

```
m=nlhw(Data,Orders,poly1d('deg',3),[]);
```

where `'deg'` is an abbreviation for the property `'Degree'`.

More About

Tips

Use `poly1d` to define a nonlinear function $y = F(x)$, where F is a single-variable polynomial function of x:

$$F(x) = c(1)x^n + c(2)x^{(n-1)} + \ldots + c(n)x + c(n+1)$$

See Also

`nlhw`

Introduced in R2007b

predict

Predict K-step ahead model output

This `predict` command computes the K-step ahead output of an identified model using measured input-output data. To identify the model, you first collect all the input-output data and then estimate the model parameters offline. To perform online state estimation of a nonlinear system using real-time data, use the `predict` command for extended and unscented Kalman filters instead.

Syntax

```
yp = predict(sys,data,K)
yp = predict(sys,data,K,opt)
[yp,x0,sys_pred] = predict( ___ )

predict(sys,data,K ___ )
predict(sys,Linespec,data,K, ___ )
predict(sys1,...,sysN,data,K, ___ )
predict(sys1,Linespec1,...,sysN,LinespecN,data,K, ___ )
```

Description

`yp = predict(sys,data,K)` predicts the output of an identified model `sys`, `K` steps ahead using the measured input-output data `data`.

`predict` command predicts the output response over the time span of measured data. In contrast, `forecast` performs prediction into the future in a time range beyond the last instant of measured data. Use `predict` to validate `sys` over the time span of measured data.

`yp = predict(sys,data,K,opt)` uses the option set `opt` to specify additional prediction options such as handling of initial conditions and data offsets.

`[yp,x0,sys_pred] = predict( ___ )` also returns the estimated values for initial states `x0` and a predictor model `sys_pred`. Use with any of the previous input argument combinations.

`predict(sys,data,K___)` plots the predicted output. Use with any of the previous input argument combinations. To change display options in the plot, right-click the plot to access the context menu. For more details about the menu, see “Tips” on page 1-1052.

You can also plot the predicted model response using the `compare` command.

`predict(sys,Linespec,data,K,___)` uses `Linespec` to specify the line type, marker symbol, and color.

`predict(sys1,...,sysN,data,K,___)` plots the predicted outputs for multiple identified models. `predict` automatically chooses colors and line styles.

`predict(sys1,Linespec1,...,sysN,LinespecN,data,K,___)` uses the line type, marker symbol, and color specified for each model.

Examples

Predict Time Series Model Response

Simulate time-series data.

```
init_sys = idpoly([1 -0.99],[],[1 -1 0.2]);
opt = simOptions('AddNoise',true);
u = iddata([],zeros(400,0),1);
data = sim(init_sys,u,opt);
```

`data` is an `iddata` object containing the simulated response data of a time series model.

Estimate an ARMAX model by using `data` as estimation data.

```
na = 1;
nb = 2;
sys = armax(data(1:200),[na nb]);
```

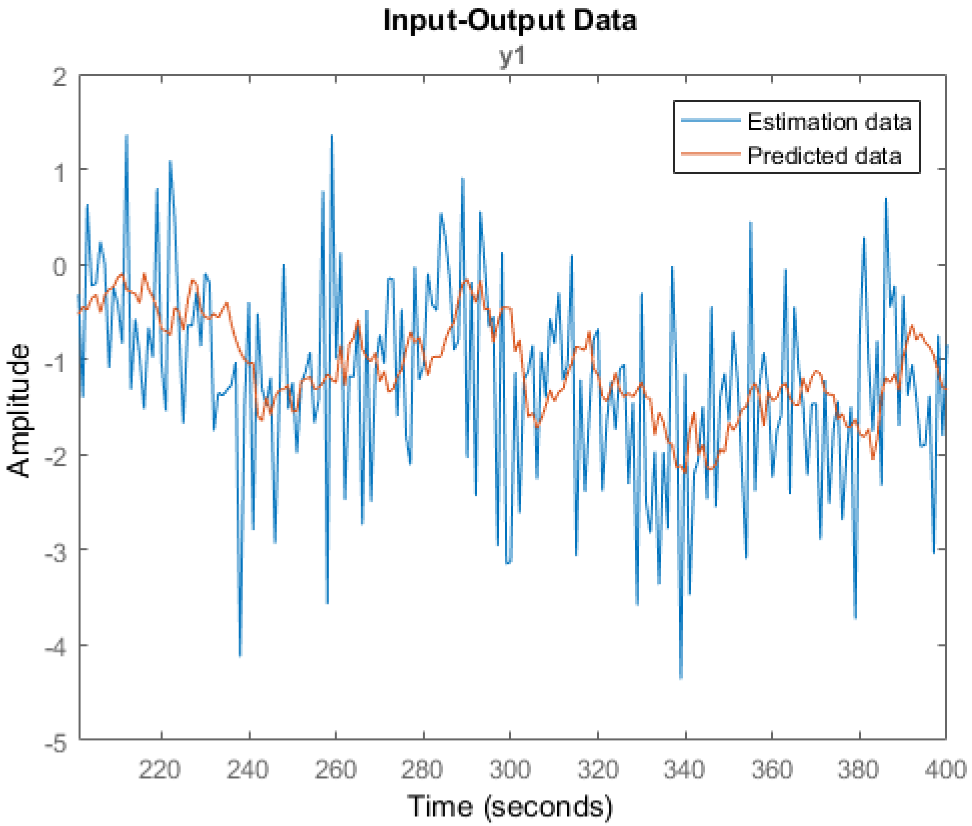
Predict the output of the model using a prediction horizon of 4.

```
K = 4;
yp = predict(sys,data,K);
```

`yp` is an `iddata` object. The predicted output is returned in the `OutputData` property of the object.

Compare the predicted and estimated data outputs.

```
plot(data(201:400),yp(201:400));
legend('Estimation data','Predicted data');
```



Alternatively, to plot the predicted response and estimation data, use `compare(sys,data,K)`.

Plot Predicted Output for Multiple Models

Load the estimation data.

```
load iddata1;
data = z1;
```

Estimate an ARX model of order [2 2 1].

```
sys1 = arx(data,[2 2 1]);
```

Estimate a transfer function with 2 poles.

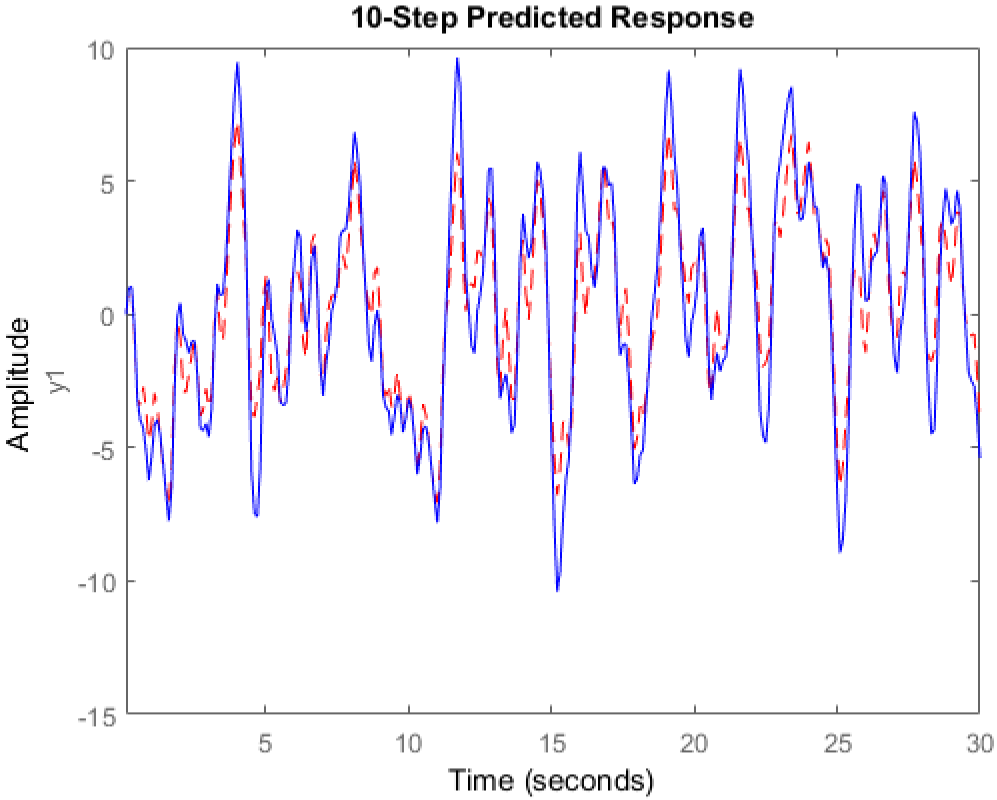
```
 sys2 = tfest(data,2);
```

Create a `predict` option set to specify zero initial conditions for prediction.

```
opt = predictOptions('InitialCondition','z');
```

Plot the predicted outputs for the estimated models. Use the specified prediction option set, `opt`, and specify prediction horizon as 10. Specify line styles for plotting the predicted output of each system.

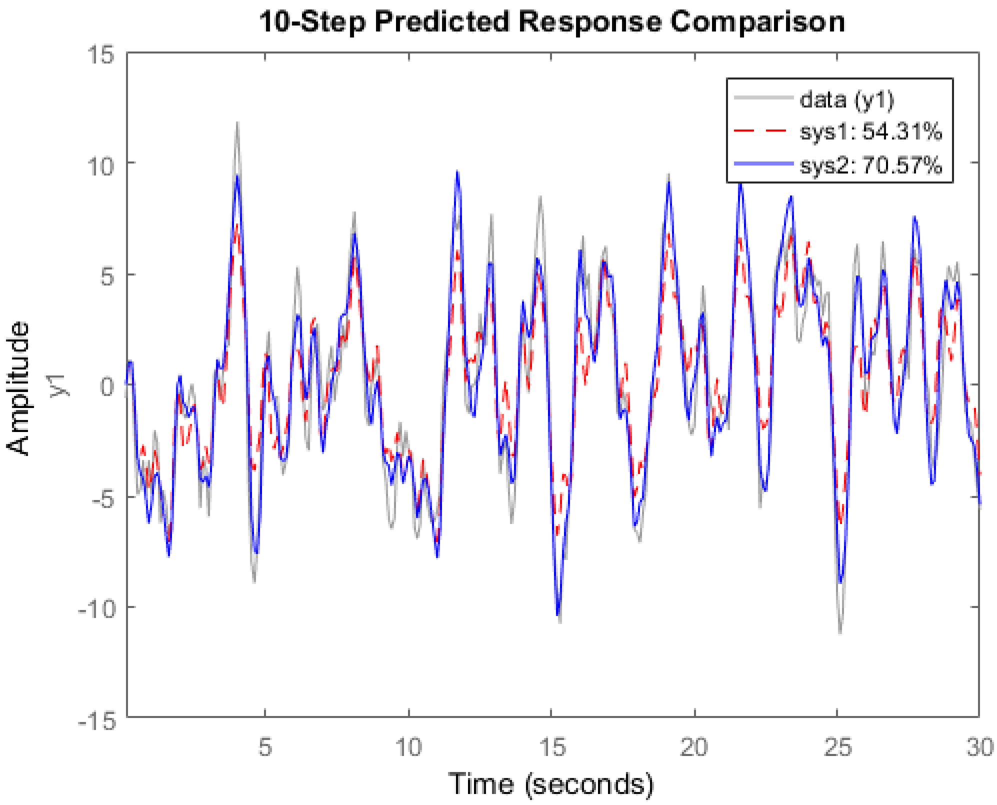
```
predict(sys1,'r--',sys2,'b',data,10,opt);
```



To change the display options, right-click the plot to access the context menu. For example, to view the estimation data, select **Show Validation Data** from the context menu. To view the prediction error, select **Prediction Error Plot**.

You can also plot the predicted response using the `compare` command. To do so, first create an option set for `compare` to specify the use of zero initial conditions.

```
opt = compareOptions('InitialCondition','z');
compare(data,sys1,'r--',sys2,'b',10,opt);
```



Reproduce Prediction Results by Simulation

Use estimation data to estimate a model, and then compute the predicted model output and predictor model using the `predict` command. Simulate the predictor model to reproduce the predicted output.

Load estimation data.

```
load iddata3 z3
data = z3;
```

Estimate a polynomial model from the data.

```
sys = polyest(z3,[2 2 2 0 0 1]);
```

Predict the system response using prediction horizon 4.

```
K = 4;
[yp,x0,sysp] = predict(sys,data,K);
```

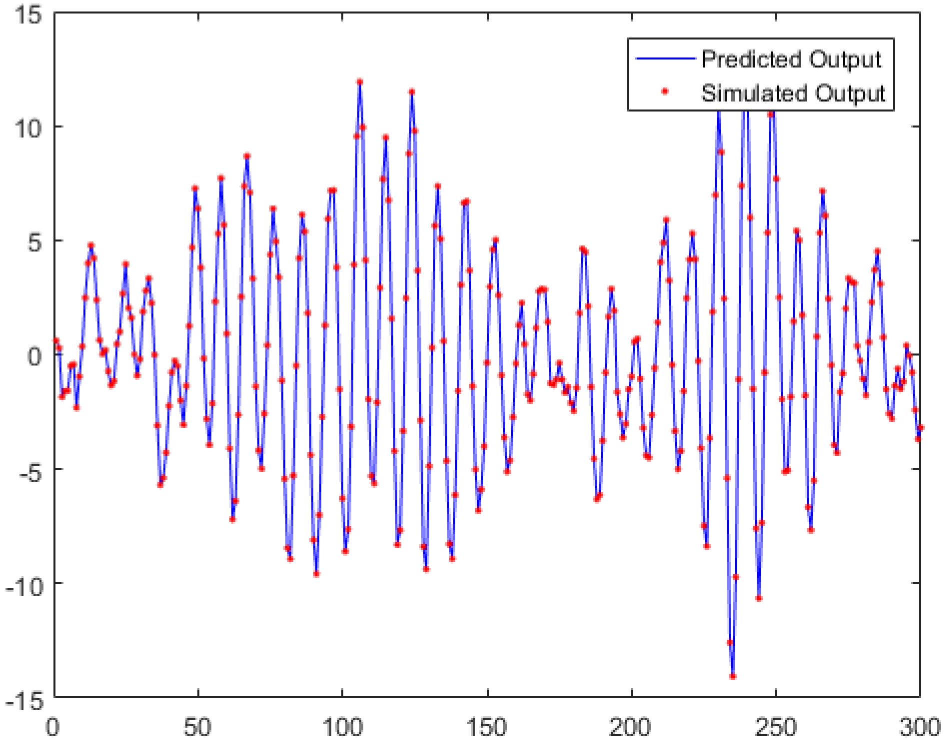
`yp` is the predicted model response, `x0` is the estimated value for initial states, and `sysp` is the predictor model.

Simulate the predictor model with inputs `[data.OutputData,data.InputData]` and initial conditions `x0`.

```
opt = simOptions;
opt.InitialCondition = x0;
ys = sim(sysp,[data.OutputData,data.InputData],opt);
```

Plot the predicted and simulated outputs.

```
t = yp.SamplingInstants;
plot(t,yp.OutputData,'b',t,ys,'.r');
legend('Predicted Output','Simulated Output')
```



Input Arguments

`sys` — Identified model

linear model | nonlinear model

Identified model whose output is to be predicted, specified as one of the following:

- Linear model — `idpoly`, `idproc`, `idss`, `idtf`, or `idgrey`
- Nonlinear model — `idnlgrey`, `idnlhw`, or `idnlarx`

  When `sys` is an `idnlhw` or `idnlgrey` model, the predicted output `yp` is the same as the simulated response computed using `data.InputData` as input.

If a model is unavailable, estimate `sys` from `data` using commands such as `ar`, `armax`, `tfest`, `nlarx`, and `ssest`.

`data` — Measured input-output data

`iddata` object | matrix of doubles

Measured input-output data, specified as one of the following:

- `iddata` object — Use observed input and output signals to create an `iddata` object. For time-series data (no inputs), specify as an `iddata` object with no inputs `iddata(output,[])`.
- Matrix of doubles — For models with*Nu* inputs and *Ny* outputs, specify `data` as an *N*-by-(*Ny*+*Nu*) matrix. Where, *N* is the number of observations.

  For time series data, specify as an *N*-by-*Ny* matrix.

`K` — Prediction horizon

`1` (default) | positive integer | `Inf`

Prediction horizon, specified as one of the following:

- Positive integer — Output `yp` is calculated `K` steps into the future, where `K` represents a multiple of `data` sample time.

  The output at time instant `t` is calculated using previously measured outputs up to time `t-K` and inputs up to the time instant `t`.
- `Inf` — No previous outputs are used in the computation, and `predict` returns the same result as simulation using the `sim` command.

For Output-Error models, there is no difference between the `K` step-ahead predictions and the simulated output. This is because Output-Error models only use past inputs to predict future outputs.

Note: For careful model validation, a one-step-ahead prediction (`K = 1`) is usually not a good test for validating the model `sys` over the time span of measured data. Even the trivial one step-ahead predictor, $\hat{y}(t) = y(t-1)$, can give good predictions. So a poor model may look fine for one-step-ahead prediction of data that has a small sample time. Prediction with `K = Inf`, which is the same as performing simulation with `sim` command, can lead to diverging outputs because low-frequency disturbances in the data are emphasized, especially for models with integration. Use a `K` value between `1` and `Inf` to capture the mid-frequency behavior of the measured data.

`opt` — Prediction options

`predictOptions` option set

Prediction options, specified as a `predictOptions` option set. Use the option set to specify prediction options such as handling of initial conditions and data offsets.

`Linespec` — Line style, marker, and color

character vector

Line style, marker, and color, specified as a character vector. For example, `'b'` or `'b+:'`.

For more information about configuring `Linespec`, see "Specify Line Style, Color, and Markers" in the MATLAB documentation.

Output Arguments

`yp` — Predicted output response

`iddata` object

Predicted output response, returned as an `iddata` object. The `OutputData` property of `yp` stores the values of the predicted output. The output at time instant `t` is calculated using previously measured outputs up to time `t-K` and inputs up to the time instant `t`. The time variable takes values in the range represented by `data.SamplingInstants`. 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.

`x0` — Estimated initial states

column vector | cell array

Estimated initial states corresponding to the predictor model `sys_pred`, returned as a column vector of size equal to number of states. If `data` is multi-experiment, `x0` is a cell array of size *Ne*, where *Ne* is the number of experiments. You can simulate `sys_pred` using `x0` as initial conditions, to reproduce the results of prediction. For an example, see "Reproduce Prediction Results by Simulation" on page 1-1047.

If you specify `opt.InitialCondition` as `'z'`, the initial conditions of data are not estimated and `x0` equals `0`.

If `sys` is an `idnlarx` model, `x0` is returned empty.

`sys_pred` — Predictor model

dynamic system model | array of models

Predictor model, returned as a dynamic system model. For multi-experiment data, `sys_pred` is an array of models, with one entry for each experiment. You can use the predictor model `sys_pred` and estimated initial states `x0` to reproduce the results of prediction:

- If `sys` is a linear model, the predictor model is returned as either a model of the same type as `sys` or as a state-space version of the model (`idss`). To reproduce the results of prediction, simulate `sys_pred` using `[data.OutputData data.InputData]` as input and `x0` as initial conditions. The simulation output is the same as the predicted output `yp.OutputData`. For an example, see “Reproduce Prediction Results by Simulation” on page 1-1047.
- When `sys` is a nonlinear grey-box model (`idnlgrey`) or Hammerstein-Wiener model (`idnlhw`), the noise-component of the model is trivial, and so the predictor model is the same as the model. `sys_pred` is returned empty. To reproduce the results of prediction, simulate `sys` using initial conditions `x0`. For a definition of the states of `idnlhw` models, see “Definition of `idnlhw` States” on page 1-567.
- If `sys` is a nonlinear ARX model (`idnlarx`), `sys_pred` and `x0` are returned empty. You cannot reproduce the prediction results by simulation.

For discrete-time data that is time-domain or frequency-domain data with sample time `Ts` greater than zero, `sys_pred` is a discrete-time model, even if `sys` is a continuous-time model.

More About

Tips

- Right-clicking the plot of the predicted output opens the context menu, where you can access the following options:
  - **Systems** — Select systems to view predicted response. By default, the response of all systems is plotted.

- **Data Experiment** — For multi-experiment data only. Toggle between data from different experiments.
- **Characteristics** — View the following data characteristics:
  - **Peak Value** — View the absolute peak value of the data. Applicable for time–domain data only.
  - **Peak Response** — View peak response of the data. Applicable for frequency-response data only.
  - **Mean Value** — View mean value of the data. Applicable for time–domain data only.
- **Show** — For frequency-domain and frequency-response data only.
  - **Magnitude** — View magnitude of frequency response of the system.
  - **Phase** — View phase of frequency response of the system.
- **Show Validation Data** — Plot data used to predict the model response.
- **I/O Grouping** — For datasets containing more than one input or output channel. Select grouping of input and output channels on the plot.
  - **None** — Plot input-output channels in their own separate axes.
  - **All** — Group all input channels together and all output channels together.
- **I/O Selector** — For datasets containing more than one input or output channel. Select a subset of the input and output channels to plot. By default, all output channels are plotted.
- **Grid** — Add grids to the plot.
- **Normalize** — Normalize the y-scale of all data in the plot.
- **Full View** — Return to full view. By default, the plot is scaled to full view.
- **Prediction Horizon** — Set the prediction horizon, or choose simulation.
- **Initial Condition** — Specify handling of initial conditions. Not applicable for frequency-response data.

  Specify as one of the following:

  - **Estimate** — Treat the initial conditions as estimation parameters.
  - **Zero** — Set all initial conditions to zero.

    - **Absorb delays and estimate** — Absorb nonzero delays into the model coefficients and treat the initial conditions as estimation parameters. Use this option for discrete-time models only.
  - **Predicted Response Plot** — Plot the predicted model response. By default, the response plot is shown.
  - **Prediction Error Plot** — Plot the error between the model response and prediction data.
  - **Properties** — Open the Property Editor dialog box to customize plot attributes.

- “Simulating and Predicting Model Output”
- “Simulation and Prediction at the Command Line”

See Also

`compare` | `forecast` | `iddata` | `pe` | `predictOptions` | `sim` | `simsd`

Introduced before R2006a

predict

Predict state and state estimation error covariance at next time step using extended or unscented Kalman filter

The `predict` command predicts the state and state estimation error covariance of an `extendedKalmanFilter` or `unscentedKalmanFilter` object at the next time step. To implement the extended or unscented Kalman filter algorithms, use the `predict` and `correct` commands together. If the current output measurement exists, you can use `predict` and `correct`. If the measurement is missing, you can only use `predict`. For information about the order in which to use the commands, see “Using predict and correct Commands” on page 1-1060.

Use this `predict` command for online state estimation using real-time data. When data is not available in real time, to compute the K-step ahead output of an identified model, use `predict` for offline estimation.

Syntax

```
[PredictedState,PredictedStateCovariance] = predict(obj)
[PredictedState,PredictedStateCovariance] = predict(obj,Us1,...Usn)
```

Description

`[PredictedState,PredictedStateCovariance] = predict(obj)` predicts state estimate and state estimation error covariance of an extended or unscented Kalman filter object `obj` at the next time step.

You create `obj` using the `extendedKalmanFilter` or `unscentedKalmanFilter` commands. You specify the state transition function and measurement function of your nonlinear system in `obj`. You also specify whether the process and measurement noise terms are additive or nonadditive in these functions. The `State` property of the object stores the latest estimated state value. Assume that at time step `k`, `obj.State` is $\hat{x}[k|k]$. This value is the state estimate for time `k`, estimated using measured outputs until time `k`. When you use the `predict` command, the software returns $\hat{x}[k+1|k]$ in the `PredictedState` output. Where $\hat{x}[k+1|k]$ is the state estimate for time `k`

`+1`, estimated using measured output until time `k`. The command returns the state estimation error covariance of $\hat{x}[k+1|k]$ in the `PredictedStateCovariance` output. The software also updates the `State` and `StateCovariance` properties of `obj` with these corrected values.

Use this syntax if the state transition function *f* that you specified in `obj.StateTransitionFcn` has one of the following forms:

- `x(k) = f(x(k-1))` — for additive process noise.
- `x(k) = f(x(k-1),w(k-1))` — for nonadditive process noise.

Where `x` and `w` are the state and process noise of the system. The only inputs to *f* are the states and process noise.

`[PredictedState,PredictedStateCovariance] = predict(obj,Us1,...Usn)` specifies additional input arguments, if the state transition function of the system requires these inputs. You can specify multiple arguments.

Use this syntax if your state transition function *f* has one of the following forms:

- `x(k) = f(x(k-1),Us1,...Usn)` — for additive process noise.
- `x(k) = f(x(k-1),w(k-1),Us1,...Usn)` — for nonadditive process noise.

Examples

Estimate States Online Using Unscented Kalman Filter

Estimate the states of a van der Pol oscillator using an unscented Kalman filter algorithm and measured output data. The oscillator has two states and one output.

Create an unscented Kalman filter object for the oscillator. Use previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`. These functions describe a discrete-approximation to a van der Pol oscillator with nonlinearity parameter, mu, equal to 1. The functions assume additive process and measurement noise in the system. Specify the initial state values for the two states as [1;0]. This is the guess for the state value at initial time `k`, using knowledge of system outputs until time `k-1`, $\hat{x}[k|k-1]$.

```
obj = unscentedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,[1;0]);
```

Load the measured output data, `y`, from the oscillator. In this example, use simulated static data for illustration. The data is stored in the `vdp_data.mat` file.

```
load vdp_data.mat y
```

Specify the process noise and measurement noise covariances of the oscillator.

```
obj.ProcessNoise = 0.01;
obj.MeasurementNoise = 0.16;
```

Implement the unscented Kalman filter algorithm to estimate the states of the oscillator by using the `correct` and `predict` commands. You first correct $\hat{x}[k|k-1]$ using measurements at time `k` to get $\hat{x}[k|k]$. Then, you predict the state value at next time step, $\hat{x}[k+1|k]$, using $\hat{x}[k|k]$, the state estimate at time step `k` that is estimated using measurements until time `k`.

To simulate real-time data measurements, use the measured data one time step at a time.

```
for k = 1:size(y)
    [CorrectedState,CorrectedStateCovariance] = correct(obj,y(k));
    [PredictedState,PredictedStateCovariance] = predict(obj);
end
```

When you use the `correct` command, `obj.State` and `obj.StateCovariance` are updated with the corrected state and state estimation error covariance values for time step `k`, `CorrectedState` and `CorrectedStateCovariance`. When you use the `predict` command, `obj.State` and `obj.StateCovariance` are updated with the predicted values for time step `k+1`, `PredictedState` and `PredictedStateCovariance`.

In this example, you used `correct` before `predict` because the initial state value was $\hat{x}[k|k-1]$, a guess for the state value at initial time `k` using system outputs until time `k-1`. If your initial state value is $\hat{x}[k-1|k-1]$, the value at previous time `k-1` using measurement until `k-1`, then use the `predict` command first. For more information about the order of using `predict` and `correct`, see “Using predict and correct Commands”.

Specify State Transition and Measurement Functions with Additional Inputs

Consider a nonlinear system with input `u` whose state `x` and measurement `y` evolve according to the folowing state transition and measurement equations:

$$x[k] = \sqrt{x[k-1] + u[k-1]} + w[k-1]$$

$$y[k] = x[k] + 2 * u[k] + v[k]^2$$

The process noise `w` of the system is additive while the measurement noise `v` is nonadditive.

Create the state transition function and measurement function for the system. Specify the functions with an additional input `u`.

```
f = @(x,u)(sqrt(x+u));
h = @(x,v,u)(x+2*u+v^2);
```

`f` and `h` are function handles to the anonymous functions that store the state transition and measurement functions, respectively. In the measurement function, because the measurement noise is nonadditive, `v` is also specified as an input. Note that `v` is specified as an input before the additional input `u`.

Create an extended Kalman filter object for estimating the state of the nonlinear system using the specified functions. Specify the initial value of the state as 1, and the measurement noise as nonadditive.

```
obj = extendedKalmanFilter(f,h,1,'HasAdditiveMeasurementNoise',false);
```

Specify the measurement noise covariance.

```
obj.MeasurementNoise = 0.01;
```

You can now estimate the state of the system using the `predict` and `correct` commands. You pass the values of `u` to `predict` and `correct`, which in turn pass them to the state transition and measurement functions, respectively.

Correct the state estimate with measurement y[k]=0.8 and input u[k]=0.2 at time step k.

```
correct(obj,0.8,0.2)
```

Predict the state at next time step, given u[k]=0.2.

```
predict(obj,0.2)
```

- “Nonlinear State Estimation Using Unscented Kalman Filter”

- “Fault Detection Using an Extended Kalman Filter”
- “Generate Code for Online State Estimation in MATLAB”

Input Arguments

`obj` — Extended or unscented Kalman filter object

`extendedKalmanFilter` object | `unscentedKalmanFilter` object

Extended or unscented Kalman filter object for online state estimation, created using one of the following commands:

- `extendedKalmanFilter` — Uses the extended Kalman filter algorithm.
- `unscentedKalmanFilter` — Uses the unscented Kalman filter algorithm.

`Us1,...Usn` — Additional input arguments to state transition function

input arguments of any type

Additional input arguments to state transition function, specified as input arguments of any type. The state transition function, *f*, is specified in the `StateTransitionFcn` property of the object. If the function requires input arguments in addition to the state and process noise values, you specify these inputs in the `predict` command syntax.

For example, suppose that your state transition function calculates the predicted state `x` at time step `k` using system inputs `u(k-1)` and time `k-1`, in addition to the state `x(k-1)`:

```
x(k) = f(x(k-1),u(k-1),k-1)
```

Then when you perform online state estimation at time step `k`, specify these additional inputs in the `predict` command syntax:

```
[PredictedState,PredictedStateCovariance] = predict(obj,u(k-1),k-1);
```

Output Arguments

`PredictedState` — Predicted state estimate

vector

Predicted state estimate, returned as a vector of size M, where M is the number of states of the system. If you specify the initial states of `obj` as a column vector then M is returned as a column vector, otherwise M is returned as a row vector.

For information about how to specify the initial states of the object, see the `extendedKalmanFilter` and `unscentedKalmanFilter` reference pages.

`PredictedStateCovariance` — Predicted state estimation error covariance
matrix

Predicted state estimation error covariance, returned as an M-by-M matrix, where M is the number of states of the system.

More About

Using `predict` and `correct` Commands

After you have created an extended or unscented Kalman filter object, `obj`, to implement the extended or unscented Kalman filter algorithms, use the `correct` and `predict` commands together.

At time step `k`, `correct` command returns the corrected value of states and state estimation error covariance using measured system outputs `y[k]` at the same time step. If your measurement function has additional input arguments U_m, you specify these as inputs to the `correct` command. The command passes these values to the measurement function.

```
[CorrectedState,CorrectedCovariance] = correct(obj,y,Um)
```

The `correct` command updates the `State` and `StateCovariance` properties of the object with the estimated values, `CorrectedState` and `CorrectedCovariance`.

The `predict` command returns the prediction of state and state estimation error covariance at the next time step. If your state transition function has additional input arguments U_s, you specify these as inputs to the `predict` command. The command passes these values to the state transition function.

```
[PredictedState,PredictedCovariance] = predict(obj,Us)
```

The `predict` command updates the `State` and `StateCovariance` properties of the object with the predicted values, `PredictedState` and `PredictedCovariance`.

If the current output measurement exists at a given time step, you can use `correct` and `predict`. If the measurement is missing, you can only use `predict`. For details about how these commands implement the algorithms, see “Extended and Unscented Kalman Filter Algorithms for Online State Estimation”.

The order in which you implement the commands depends on the availability of measured data `y`, U_s, and U_m for your system:

- `correct` then `predict` — Assume that at time step `k`, the value of `obj.State` is $\hat{x}[k|k-1]$. This value is the state of the system at time `k`, estimated using measured outputs until time `k-1`. You also have the measured output `y[k]` and inputs U_s`[k]` and U_m`[k]` at the same time step.

  Then you first execute the `correct` command with measured system data `y[k]` and additional inputs U_m`[k]`. The command updates the value of `obj.State` to be $\hat{x}[k|k]$, the state estimate for time `k`, estimated using measured outputs up to time `k`. When you then execute the `predict` command with input U_s`[k]`, `obj.State` now stores $\hat{x}[k+1|k]$. The algorithm uses this state value as an input to the `correct` command in the next time step.
- `predict` then `correct` — Assume that at time step `k`, the value of `obj.State` is $\hat{x}[k-1|k-1]$. You also have the measured output `y[k]` and input U_m`[k]` at the same time step but you have U_s`[k-1]` from the previous time step.

  Then you first execute the `predict` command with input U_s`[k-1]`. The command updates the value of `obj.State` to $\hat{x}[k|k-1]$. When you then execute the `correct` command with input arguments `y[k]` and U_m`[k]`, `obj.State` is updated with $\hat{x}[k|k]$. The algorithm uses this state value as an input to the `predict` command in the next time step.

Thus, while in both cases the state estimate for time `k`, $\hat{x}[k|k]$ is the same, if at time `k` you do not have access to the current state transition function inputs U_s`[k]`, and instead have U_s`[k-1]`, then use `predict` first and then `correct`.

For an example of estimating states using the `predict` and `correct` commands, see “Estimate States Online Using Unscented Kalman Filter” on page 1-1056.

- “What Is Online Estimation?”

- "Extended and Unscented Kalman Filter Algorithms for Online State Estimation"

See Also

`clone` | `correct` | `extendedKalmanFilter` | `unscentedKalmanFilter`

Introduced in R2016b

predictOptions

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. `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 only for state-space and nonlinear models.
- `io` — Structure with the following fields:
  - `Input`
  - `Output`

  Use the `Input` and `Output` fields to specify the history for a time interval. This interval must start 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 *Ne* elements, where *Ne* 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, `sys`, `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: `'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 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 before the input 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. *Ny* is the number of outputs, and *Ne* is the number of experiments.

Each entry in `OutputOffset` is subtracted from the corresponding output data before model prediction. The entry value is then added to the corresponding model output response after prediction.

Default: `[]`

`'OutputWeight'`

Weight of output for initial condition estimation.

`OutputWeight` takes one of the following:

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

Default: `[]`

Output Arguments

`opt`

Option set containing the specified options for `predict`.

Examples

Create Default Option Set for Model Prediction

```
opt = predictOptions;
```

Specify Options for Model Prediction

Create an option set for `predict` using zero initial conditions.

```
opt = predictOptions('InitialCondition','z');
```

Set the input offset to 5 using dot notation.

```
opt.InputOffset = 5;
```

See Also

`absorbdelay` | `idpar` | `predict`

Introduced in R2012a

present

Display model information, including estimated uncertainty

Syntax

```
present(m)
```

Description

`present(m)` displays the linear or nonlinear identified model `m` and the following information:

- Estimated one standard deviation of the parameters, which gives 68.27% confidence region
- Termination conditions for iterative estimation algorithms
- Status of the model — whether the model was constructed or estimated
- Fit to estimation data
- Akaike's Final Prediction Error (FPE) criterion
- Mean-square error (MSE)

Examples

Display Information About Identified Model

Estimate a transfer function model.

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

Display model information.

```
present(sys)
```

```
sys =
 
  From input "u1" to output "y1":
      2.455 (+/- 1.101) s + 177 (+/- 10.73)
  ---------------------------------------------
  s^2 + 3.163 (+/- 0.2522) s + 23.16 (+/- 1.115)
 
Continuous-time identified transfer function.

Parameterization:
   Number of poles: 2   Number of zeros: 1
   Number of free coefficients: 4
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Termination condition: Near (local) minimum, (norm(g) < tol).
Number of iterations: 1, Number of function evaluations: 3

Estimated using TFEST on time domain data "z1".
Fit to estimation data: 70.77%
FPE: 1.725, MSE: 1.658
More information in model's "Report" property.
```

More About

- "Estimation Report"
- "Loss Function and Model Quality Metrics"

See Also

`frdata` | `getcov` | `getpvec` | `idssdata` | `polydata` | `ssdata` | `tfdata` | `zpkdata`

Introduced before R2006a

procest

Estimate process model using time or frequency data

Syntax

```
sys = procest(data,type)
sys = procest(data,type,'InputDelay',InputDelay)

sys = procest(data,init_sys)

sys = procest( ___ ,opt)

[sys,offset] = procest( ___ )
```

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,'InputDelay',InputDelay)` specifies the input delay `InputDelay`.

`sys = procest(data,init_sys)` uses the process model `init_sys` to configure the initial parameterization.

`sys = procest( ___ ,opt)` specifies additional model estimation options. Use `opt` with any of the previous syntaxes.

`[sys,offset] = procest( ___ )` returns the estimated value of the offset in input signal. Input offset is automatically estimated when the model contains an integrator, or when you set the `InputOffset` estimation option to `'estimate'` using `procestOptions`. Use `offset` with any of the previous syntaxes.

Input Arguments

`data` — Estimation data

`iddata` | `idfrd` | `frd`

Estimation data, specified as an `iddata` object containing the input and output signal values, for time-domain estimation. 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.

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.

`type` — Process model structure

character vector | cell array of character vectors

Process model structure, specified for SISO models as a character vector representing an acronym for the model structure, such as `'P1D'` or `'P2DZ'`. The acronym 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` — Underdamped modes (complex-valued poles) permitted. If `U` is not included in `type`, all poles must 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 MIMO models, use a cell array of character vector acronyms, with one entry for each input/output pair.

`InputDelay` — Input delays

`0` for all input channels (default) | numeric vector

Input delays, specified as 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. Each entry of this vector is a numerical value that represents 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.

`init_sys` — System for configuring initial parametrization

`idproc` object

System for configuring initial parametrization of `sys`, specified as an `idproc` object. You obtain `init_sys` by either performing an estimation using measured data or by direct construction using `idproc`. The software 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 Kp, T_{p1}, T_{p2}, T_{p3}, T_w, $Zeta$, T_d, and T_z. For example:

- To specify an initial guess for the T_{p1} parameter of `init_sys`, set `init_sys.Structure.Tp1.Value` as the initial guess.
- To specify constraints for 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.

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

`opt` — Estimation options

`procestOptions` option set

Estimation options, specified as an `procestOptions` option set. The estimation options include:

- Estimation objective
- Handling on initial conditions and disturbance component
- Numerical search method to be used in estimation

Output Arguments

`sys` — Identified process model

`idproc` model

Identified process model, returned as an `idproc` model of a structure defined by `type`.

Information about the estimation results and options used is stored in the model's `Report` property. `Report` has the following fields:

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Status</code></td><td>Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.</td></tr>
<tr><td><code>Method</code></td><td>Estimation command used.</td></tr>
<tr><td><code>InitialCondition</code></td><td>Handling of initial conditions during model estimation, returned as one of the following values:
• <code>'zero'</code> — The initial conditions were set to zero.
• <code>'estimate'</code> — The initial conditions were treated as independent estimation parameters.
• <code>'backcast'</code> — The initial conditions were estimated using the best least squares fit.
This field is especially useful to view how the initial conditions were handled when the <code>InitialCondition</code> option in the estimation option set is <code>'auto'</code>.</td></tr>
<tr><td><code>Fit</code></td><td>Quantitative assessment of the estimation, returned as a structure. See “Loss Function and Model Quality Metrics” for more information on these quality metrics. The structure has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>FitPer</code></td><td>Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as a percentage.</td></tr>
<tr><td><code>LossFc</code></td><td>Value of the loss function when the estimation completes.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>MSE</code></td><td>Mean squared error (MSE) measure of how well the response of the model fits the estimation data.</td></tr>
<tr><td><code>FPE</code></td><td>Final prediction error for the model.</td></tr>
<tr><td><code>AIC</code></td><td>Raw Akaike Information Criteria (AIC) measure of model quality.</td></tr>
<tr><td><code>AICc</code></td><td>Small sample-size corrected AIC.</td></tr>
<tr><td><code>nAIC</code></td><td>Normalized AIC.</td></tr>
<tr><td><code>BIC</code></td><td>Bayesian Information Criteria (BIC).</td></tr>
</table>
</td></tr>
<tr><td><code>Parameters</code></td><td>Estimated values of model parameters.</td></tr>
<tr><td><code>OptionsUsed</code></td><td>Option set used for estimation. If no custom options were configured, this is a set of default options. See <code>procestOptions</code> for more information.</td></tr>
<tr><td><code>RandState</code></td><td>State of the random number stream at the start of estimation. Empty, <code>[]</code>, if randomization was not used during estimation. For more information, see <code>rng</code> in the MATLAB documentation.</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation. Structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
<tr><td><code>Type</code></td><td>Data type. For <code>idnlarx</code> models, this is set to <code>'Time domain data'</code>.</td></tr>
<tr><td><code>Length</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time. This is equivalent to <code>Data.Ts</code>.</td></tr>
<tr><td><code>InterSa</code></td><td>Input intersample behavior. One of the following values:
• <code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
• <code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
• <code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.
The value of <code>Intersample</code> has no effect on estimation results for discrete-time models.</td></tr>
<tr><td><code>InputOf</code></td><td>Empty, <code>[]</code>, for nonlinear estimation methods.</td></tr>
<tr><td><code>OutputO</code></td><td>Empty, <code>[]</code>, for nonlinear estimation methods.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Termination</code></td><td>Termination conditions for the iterative search used for prediction error minimization. Structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>WhyStop</code></td><td>Reason for terminating the numerical search.</td></tr>
<tr><td><code>Iterati</code></td><td>Number of search iterations performed by the estimation algorithm.</td></tr>
<tr><td><code>FirstOr</code></td><td>∞ -norm of the gradient search vector when the search algorithm terminates.</td></tr>
<tr><td><code>FcnCour</code></td><td>Number of times the objective function was called.</td></tr>
<tr><td><code>UpdateN</code></td><td>Norm of the gradient search vector in the last iteration. Omitted when the search method is <code>'lsqnonlin'</code>.</td></tr>
<tr><td><code>LastImp</code></td><td>Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is <code>'lsqnonlin'</code>.</td></tr>
<tr><td><code>Algorit</code></td><td>Algorithm used by <code>'lsqnonlin'</code> search method. Omitted when other search methods are used.</td></tr>
</table>
For estimation methods that do not require numerical search optimization, the <code>Termination</code> field is omitted.</td></tr>
</table>

For more information on using `Report`, see “Estimation Report”.

`offset` — Estimated value of input offset

vector

Estimated value of input offset, returned as a vector. When `data` has multiple experiments, `offset` is a matrix where each column corresponds to an experiment.

Examples

Estimate a First Order Plus Dead Time Model

Obtain the measured input-output data.

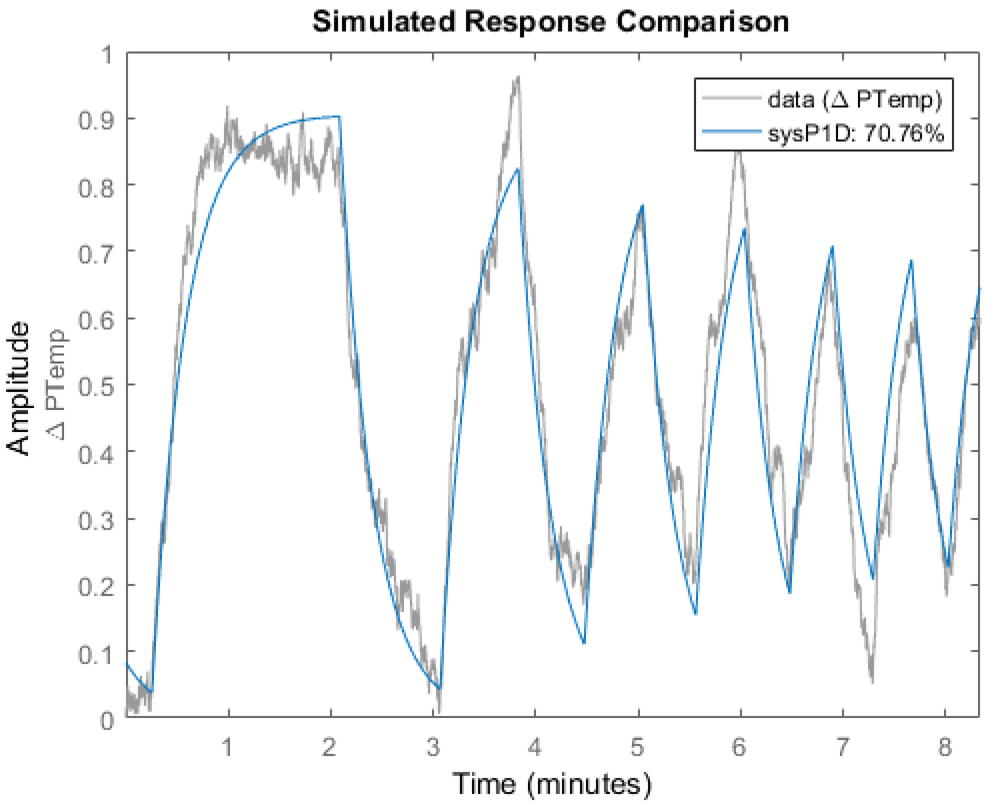
```
load iddemo_heatexchanger_data;
data = iddata(pt,ct,Ts);
data.InputName  = '\Delta CTemp';
data.InputUnit  = 'C';
data.OutputName = '\Delta PTemp';
data.OutputUnit = 'C';
data.TimeUnit   = 'minutes';
```

Estimate a first-order plus dead time process model.

```
type = 'P1D';
sysP1D = procest(data,type);
```

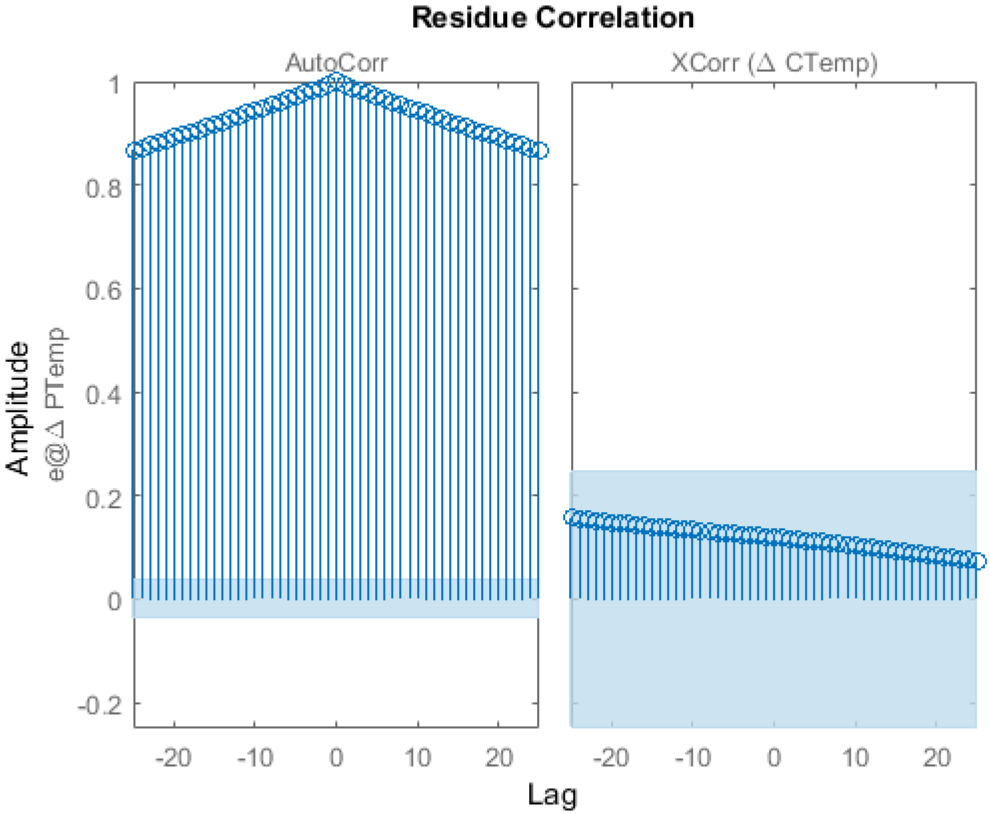
Compare the model with the data.

```
compare(data,sysP1D)
```



Plot the model residuals.

```
figure
resid(data,sysP1D);
```

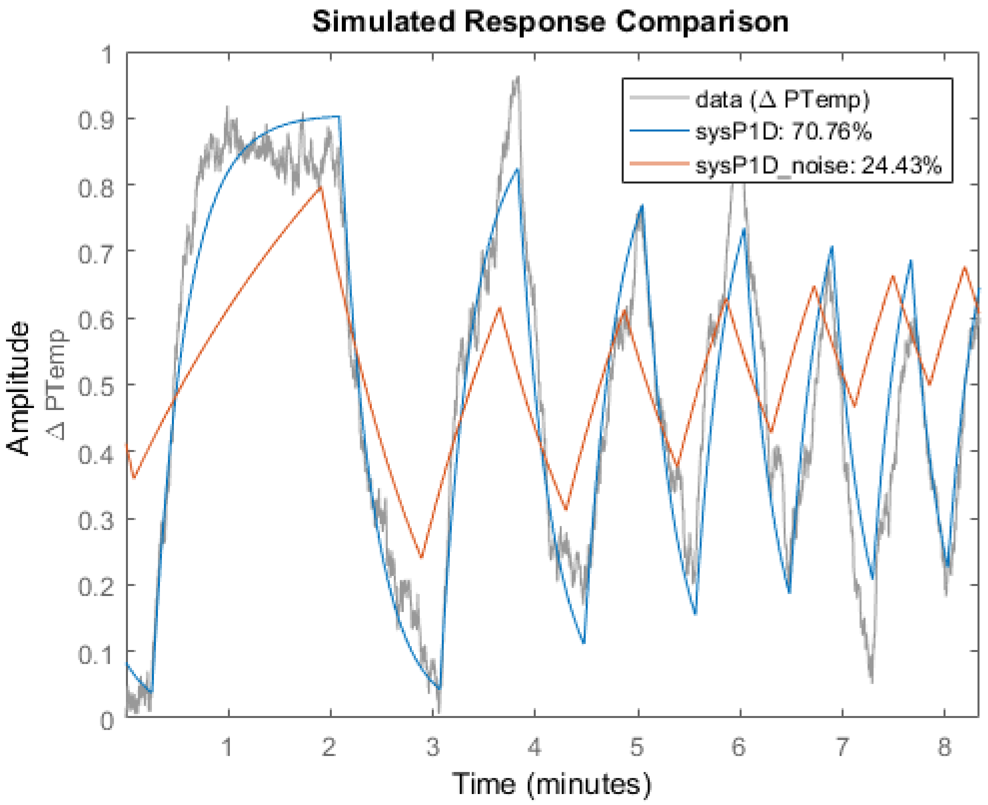


The figure shows that the residuals are correlated. To account for that, add a first order ARMA disturbance component to the process model.

```
opt = procestOptions('DisturbanceModel','ARMA1');
sysP1D_noise = procest(data,'p1d',opt);
```

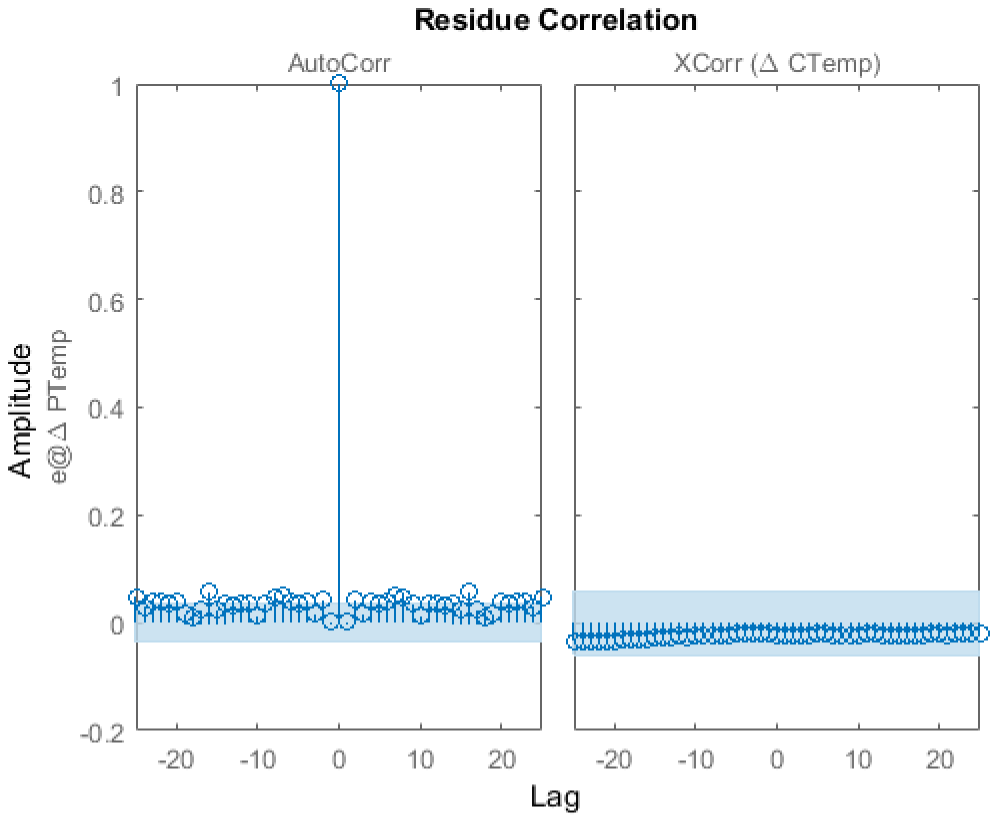
Compare the models.

```
compare(data,sysP1D,sysP1D_noise)
```



Plot the model residuals.

```
figure
resid(data,sysP1D_noise);
```



The residues of `sysP1D_noise` are uncorrelated.

Estimate Over-parameterized Process Model Using Regularization

Use regularization to estimate parameters of an over-parameterized process model.

Assume that gain is known with a higher degree of confidence than other model parameters.

Load data.

```
load iddata1 z1;
```

Estimate an unregularized process model.

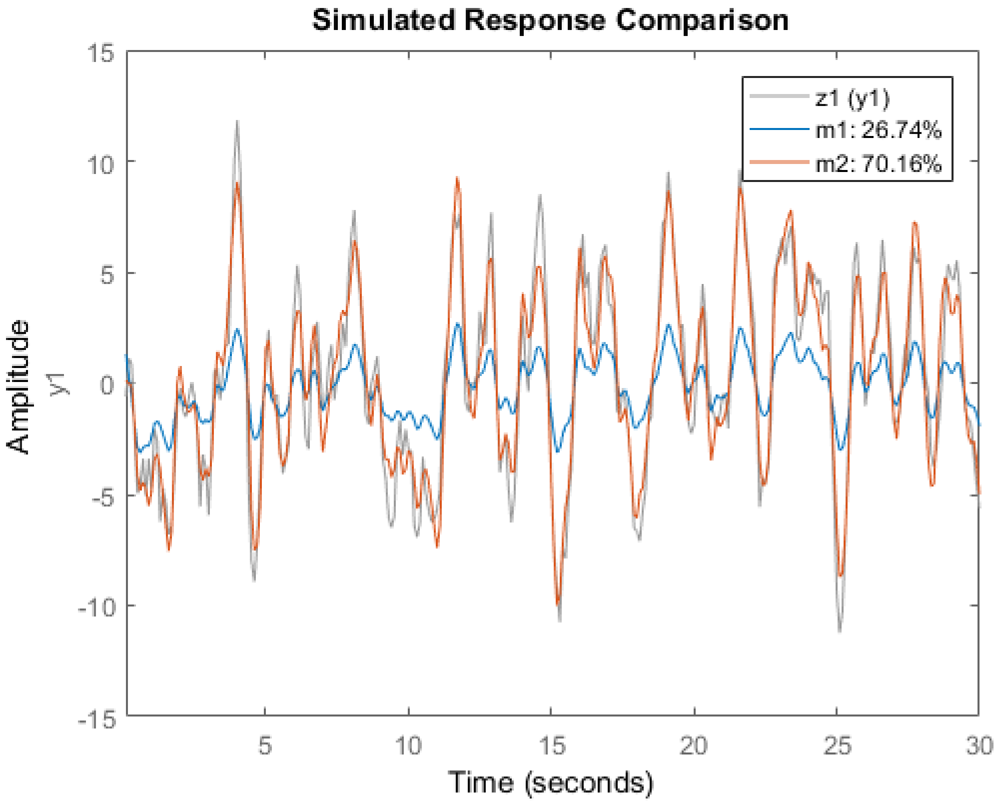
```
m = idproc('P3UZ','K',7.5,'Tw',0.25,'Zeta',0.3,'Tp3',20,'Tz',0.02);
m1 = procest(z1,m);
```

Estimate a regularized process model.

```
opt = procestOptions;
opt.Regularization.Nominal = 'model';
opt.Regularization.R = [100;1;1;1;1];
opt.Regularization.Lambda = 0.1;
m2 = procest(z1,m,opt);
```

Compare the model outputs with data.

```
compare(z1,m1,m2);
```



Regularization helps steer the estimation process towards the correct parameter values.

Specify Parameter Initial Values for Estimated Process Model

Estimate a process model after specifying initial guesses for parameter values and bounding them.

Obtain input/output data.

```
data = idfrd(idtf([10 2],[1 1.3 1.2],'iod',0.45),logspace(-2,2,256));
```

Specify the parameters of the estimation initialization model.

```
type = 'P2UZD';
init_sys = idproc(type);

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;
```

Specify the estimation options.

```
opt = procestOptions('Display','full','InitialCondition','Zero');
opt.SearchMethod = 'lm';
opt.SearchOption.MaxIter = 100;
```

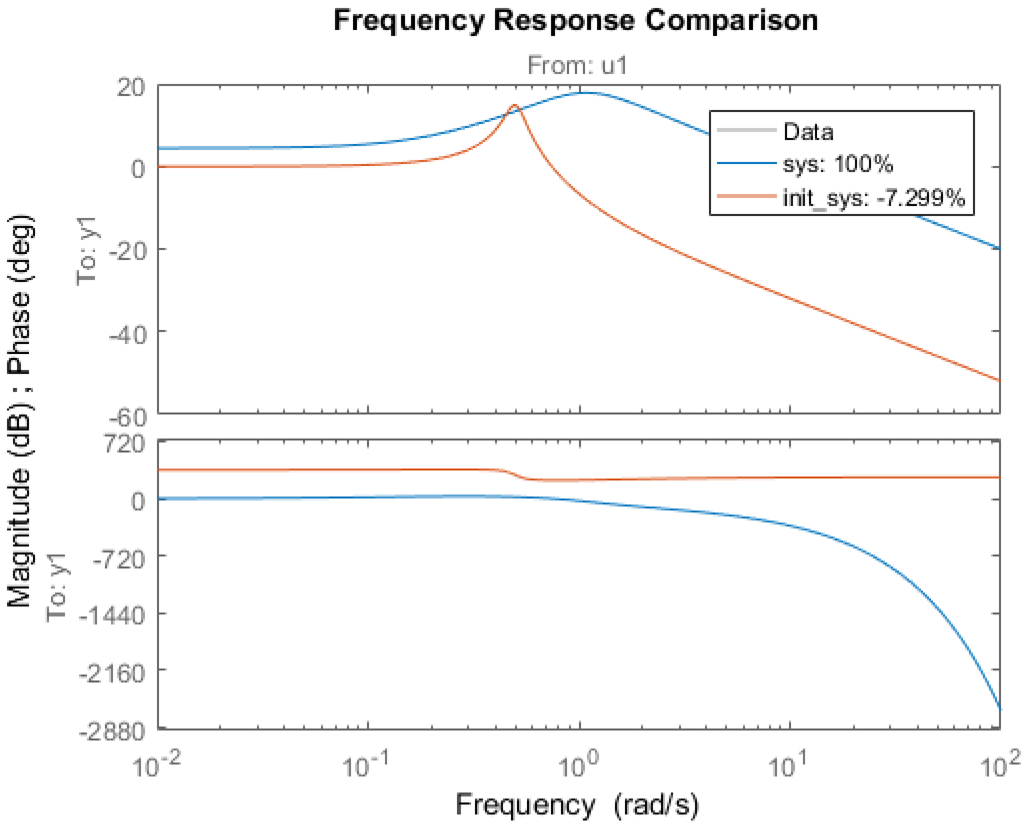
Estimate the process model.

```
sys = procest(data,init_sys,opt);
```

Since the `'Display'` option is specified as `'full'`, the estimation progress is displayed in a separate **Plant Identification Progress** window.

Compare the data to the estimated model.

```
compare(data,sys,init_sys);
```



Detect Overparameterization of Estimated Model

Obtain input/output data.

```
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, modeling 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 show a high level of uncertainty.

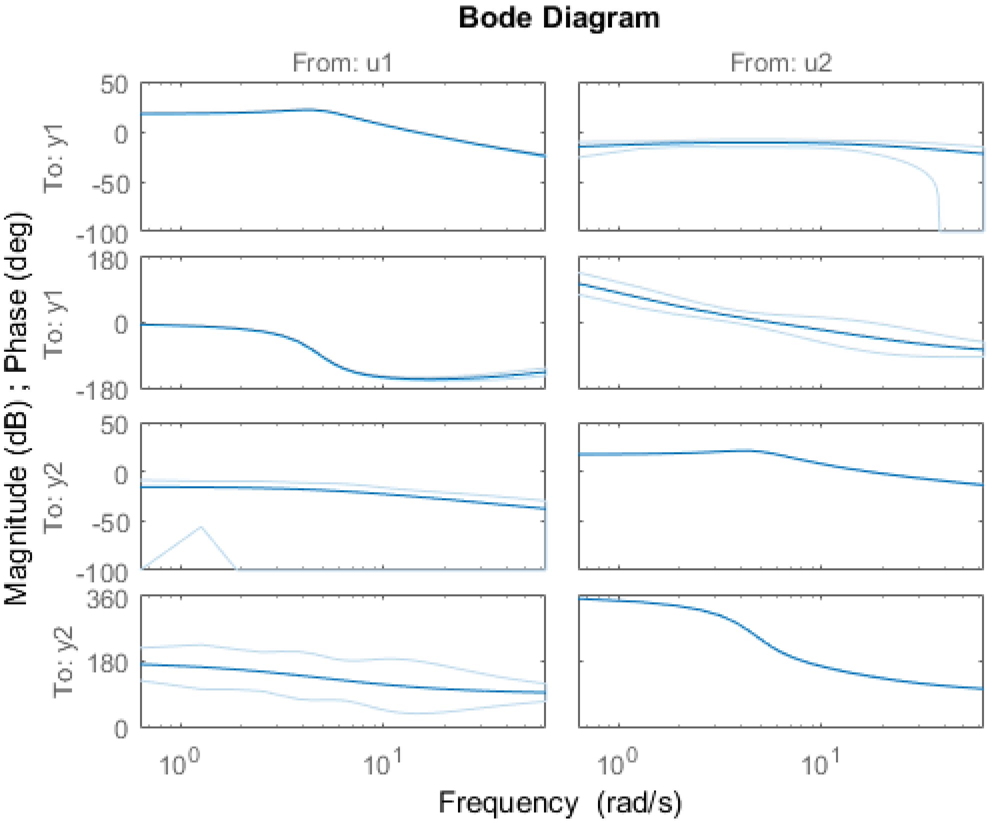
Estimate the process model.

```
type = 'P2UZ';
sys = procest(data,type);
```

The `type` variable denotes a model with complex-conugate pair of poles, a zero, and a delay.

To evaluate the uncertainties, plot the frequency response.

```
w = linspace(0,20*pi,100);
h = bodeplot(sys,w);
showConfidence(h);
```



Return Input Offsets Estimated During Process Model Estimation

```
load iddata1
[sys,offset] = procest(z1,'P1DI');
offset


offset =

    0.0412
```

More About

- “What Is a Process Model?”
- “Regularized Estimates of Model Parameters”

See Also

`ar` | `arx` | `bj` | `idproc` | `oe` | `polyest` | `procestOptions` | `ssest` | `tfest`

Introduced in R2012a

procestOptions

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. `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'` — Handling of initial conditions

`'auto'` (default) | `'zero'` | `'estimate'` | `'backcast'`

Handling of initial conditions during estimation, specified as 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.

`'DisturbanceModel'` — Handling of additive noise

`'estimate'` (default) | `'none'` | `'ARMA1'` | `'ARMA2'` | `'fixed'`

Handling of additive noise (H) during estimation for the model

$$y = G(s)u + H(s)e$$

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` is specified as one of the following values:

- `'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

  $$\frac{1+c_1 s+c_2 s^2}{1+d_1 s+d_2 s^2}$$
- `'fixed'` — The software fixes the value of the `NoiseTF` property of the `idproc` model as the value of H.

Note: A noise model cannot be estimated using frequency domain data.

`'Focus'` — Error to be minimized

`'prediction'` (default) | `'simulation'`

Error to be minimized in the loss function during estimation, specified as the comma-separated pair consisting of `'Focus'` and one of the following values:

- `'prediction'` — The one-step ahead prediction error between measured and predicted outputs is minimized during estimation. As a result, the estimation focuses on producing a good predictor model.
- `'simulation'` — The simulation error between measured and simulated outputs is minimized during estimation. As a result, the estimation focuses on making a good fit for simulation of model response with the current inputs.

The `Focus` option can be interpreted as a weighting filter in the loss function. For more information, see “Loss Function and Model Quality Metrics”.

`'WeightingFilter'` — Weighting prefilter

[ ] (default) | vector | matrix | cell array | linear system

Weighting prefilter applied to the loss function to be minimized during estimation. To understand the effect of `WeightingFilter` on the loss function, see “Loss Function and Model Quality Metrics”.

Specify `WeightingFilter` as one of the following values:

- `[]` — No weighting prefilter is used.
- Passbands — Specify a row vector or matrix containing frequency values that define desired passbands. You select a frequency band where the fit between estimated model and estimation data is optimized. For example, `[wl,wh]` where `wl` and `wh` represent lower and upper limits of a passband. For a matrix with several rows defining frequency passbands, `[w1l,w1h;w2l,w2h;w3l,w3h;...]`, the estimation algorithm uses the union of the frequency ranges to define the estimation passband.

  Passbands are expressed in `rad/TimeUnit` for time-domain data and in `FrequencyUnit` for frequency-domain data, where `TimeUnit` and `FrequencyUnit` are the time and frequency units of the estimation data.
- SISO filter — Specify a single-input-single-output (SISO) linear filter in one of the following ways:
  - A SISO LTI model
  - `{A,B,C,D}` format, which specifies the state-space matrices of a filter with the same sample time as estimation data.
  - `{numerator,denominator}` format, which specifies the numerator and denominator of the filter as a transfer function with same sample time as estimation data.

This option calculates the weighting function as a product of the filter and the input spectrum to estimate the transfer function.

- Weighting vector — Applicable for frequency-domain data only. Specify a column vector of weights. This vector must have the same length as 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.

`'EstCovar'` — Control whether to generate parameter covariance data

`true` (default) | `false`

Controls whether parameter covariance data is generated, specified as `true` or `false`.

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

`'Display'` — Specify whether to display the estimation progress

`'off'` (default) | `'on'`

Specify whether to display the estimation progress, specified as one of the following values:

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

`'InputOffset'` — Removal of offset from time-domain input data

`'auto'` (default) | `'estimate'` | vector | matrix | object | `[]`

Removal of offset from time-domain input data during estimation, specified as one of the following values:

- `'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. *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.

- A parameter object, constructed using `param.Continuous`, that imposes constraints on how the software estimates 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;
  ```

`'OutputOffset'` — Removal of offset from time-domain output data during estimation

`[]` (default) | vector | matrix

Removal of offset from time-domain output data during estimation, specified as the comma-separated pair consisting of `'OutputOffset'` and one of the following:

- A column vector of length *Ny*, where *Ny* is the number of outputs.
- `[]` — Indicates no offset.
- *Ny*-by-*Ne* matrix — For multi-experiment 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.

`'OutputWeight'` — Weighting of prediction errors in multi-output estimations

`[]` (default) | `'noise'` | positive semidefinite symmetric matrix

Weighting of prediction errors in multi-output estimations, specified as one of the following values:

- `'noise'` — Minimize $\det(E'^{*}E/N)$, where E represents the prediction error and `N` is the number of data samples. This choice is optimal in a statistical sense and leads to maximum likelihood estimates if 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 semidefinite symmetric matrix (`W`) — Minimize the trace of the weighted prediction error matrix `trace(E'*E*W/N)` where:
    - *E* is the matrix of prediction errors, with one column for each output, and *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-output models, or the reliability of corresponding data.
    - `N` is the number of data samples.
- `[]` — The software chooses between the `'noise'` or using the identity matrix for `W`.

This option is relevant for only multi-output models.

`'Regularization'` — Options for regularized estimation of model parameters

structure

Options for regularized estimation of model parameters. For more information on regularization, see “Regularized Estimates of Model Parameters”.

`Regularization` is a structure with the following fields:

- `Lambda` — Constant that determines the bias versus variance tradeoff.

  Specify a positive scalar to add the regularization term to the estimation cost.

  The default value of zero implies no regularization.

  Default: 0
- `R` — Weighting matrix.

  Specify a vector of nonnegative numbers or a square positive semi-definite matrix. The length must be equal to the number of free parameters of the model.

  For black-box models, using the default value is recommended. For structured and grey-box models, you can also specify a vector of `np` positive numbers such that each entry denotes the confidence in the value of the associated parameter.

The default value of 1 implies a value of `eye(npfree)`, where `npfree` is the number of free parameters.

Default: 1

- `Nominal` — The nominal value towards which the free parameters are pulled during estimation.

  The default value of zero implies that the parameter values are pulled towards zero. If you are refining a model, you can set the value to `'model'` to pull the parameters towards the parameter values of the initial model. The initial parameter values must be finite for this setting to work.

  Default: 0

`'SearchMethod'` — Numerical search method used for iterative parameter estimation

`'auto'` (default) | `'gn'` | `'gna'` | `'lm'` | `'grad'` | `'lsqnonlin'`

Numerical search method used for iterative parameter estimation, specified as one of the following values:

- `'gn'` — The subspace Gauss-Newton direction. Singular values of the Jacobian matrix less than `GnPinvConst*eps*max(size(J))*norm(J)` are discarded when computing the search direction. J is the Jacobian matrix. The Hessian matrix is approximated by J^TJ. 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 `gamma*max(sv)` of the Hessian are ignored, 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). *gamma* is increased by the factor `LMStep` each time the search fails to find a lower value of the criterion in less than 5 bisections. *gamma* is decreased by a factor of `2*LMStep` each time a search is successful without any bisections.
- `'lm'` — Uses the Levenberg-Marquardt method so that the next parameter value is `-pinv(H+d*I)*grad` from the previous one, where H is the Hessian, I is the identity matrix, and *grad* is the gradient. d is a number that is increased until a lower value of the criterion is found.
- `'lsqnonlin'` — Uses `lsqnonlin` optimizer from Optimization Toolbox software. This search method can handle only the Trace criterion.

- `'grad'` — The steepest descent gradient search method.
- `'auto'` — The algorithm chooses one of the preceding options. The descent direction is calculated using `'gn'`, `'gna'`, `'lm'`, and `'grad'` successively at each iteration. The iterations continue until a sufficient reduction in error is achieved.

`'SearchOption'` — Option set for the search algorithm

search option set

Option set for the search algorithm with fields that depend on the value of `SearchMethod`.

SearchOption structure when SearchMethod is specified as 'gn', 'gna', 'lm', 'grad', or 'auto'

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td><code>Toleran</code></td><td>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 <code>Tolerance</code>, the iterations stop. The estimate of the expected loss-function improvement at the next iteration is based on the Gauss-Newton vector computed for the current parameter value.

Default: <code>0.01</code></td></tr>
<tr><td><code>MaxIter</code></td><td>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>Tolerance</code>.

Setting <code>MaxIter = 0</code> returns the result of the start-up procedure.

Use <code>sys.Report.Termination.Iterations</code> to get the actual number of iterations during an estimation, where <i>sys</i> is an <code>idtf</code> model.

Default: <code>20</code></td></tr>
<tr><td><code>Advance</code></td><td>Advanced search settings.

Specified as a structure with the following fields:

<table><tr><th>Field Name</th><th>Description</th></tr><tr><td><code>GnPinvCon</code></td><td>Singular values of the Jacobian matrix that are smaller than <code>GnPinvConst*max(size(J)*norm(J)*eps)</code> are discarded</td></tr></table></td></tr>
</table>

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>when computing the search direction. Applicable when <code>SearchMethod</code> is <code>'gn'</code>.
<code>GnPinvConst</code> must be a positive, real value.
Default: <code>10000</code></td></tr>
<tr><td><code>InitGnaTo</code></td><td>Initial value of <i>gamma</i>. Applicable when <code>SearchMethod</code> is <code>'gna'</code>.
Default: <code>0.0001</code></td></tr>
<tr><td><code>LMStartVa</code></td><td>Starting value of search-direction length <i>d</i> in the Levenberg-Marquardt method. Applicable when <code>SearchMethod</code> is <code>'lm'</code>.
Default: <code>0.001</code></td></tr>
<tr><td><code>LMStep</code></td><td>Size of the Levenberg-Marquardt step. The next value of the search-direction length <i>d</i> in the Levenberg-Marquardt method is <code>LMStep</code> times the previous one. Applicable when <code>SearchMethod</code> is <code>'lm'</code>.
Default: <code>2</code></td></tr>
<tr><td><code>MaxBisect</code></td><td>Maximum number of bisections used by the line search along the search direction.
Default: <code>25</code></td></tr>
<tr><td><code>MaxFunEva</code></td><td>Iterations stop if the number of calls to the model file exceeds this value.
<code>MaxFunEvals</code> must be a positive, integer value.
Default: <code>Inf</code></td></tr>
<tr><td><code>MinParCha</code></td><td>Smallest parameter update allowed per iteration.
<code>MinParChange</code> must be a positive, real value.
Default: <code>0</code></td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td><code>RelImprov</code></td><td>Iterations stop if the relative improvement of the criterion function is less than <code>RelImprovement</code>.
<code>RelImprovement</code> must be a positive, integer value.
Default: <code>0</code></td></tr>
<tr><td><code>StepReduc</code></td><td>Suggested parameter update is reduced by the factor <code>StepReduction</code> after each try. This reduction continues until either <code>MaxBisections</code> tries are completed or a lower value of the criterion function is obtained.
<code>StepReduction</code> must be a positive, real value that is greater than 1.
Default: <code>2</code></td></tr>
</table>
</td></tr>
</table>

SearchOption structure when SearchMethod is specified as 'lsqnonlin'

Field Name	Description
`TolFun`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values. The value of `TolFun` is the same as that of `opt.SearchOption.Advanced.TolFun`. **Default:** `1e-5`
`TolX`	Termination tolerance on the estimated parameter values. The value of `TolX` is the same as that of `opt.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.

Field Name	Description
	The value of `MaxIter` is the same as that of `opt.SearchOption.Advanced.MaxIter`. **Default:** `20`
`Advance`	Options set for `lsqnonlin`. For more information, see the Optimization Options table in "Optimization Options". Use `optimset('lsqnonlin')` to create an options set for `lsqnonlin`, and then modify it to specify its various options.

`'Advanced'` — Additional advanced options

structure

`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 loss function. The standard deviation is estimated robustly as the median of the absolute deviations from the median of the prediction errors, divided by `0.7`. For more information on robust norm choices, see section 15.2 of [1].

  `ErrorThreshold = 0` disables robustification and leads to a purely quadratic loss function. When estimating with frequency-domain data, the software sets `ErrorThreshold` to zero. For time-domain data that contains outliers, try setting `ErrorThreshold` to `1.6`.

  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 condition.

  The initial condition is estimated when

  $$\frac{\left\| y_{p,z} - y_{meas} \right\|}{\left\| y_{p,e} - y_{meas} \right\|} > \text{AutoInitThreshold}$$

  - y_{meas} is the measured output.
  - $y_{p,z}$ is the predicted output of a model estimated using zero initial states.
  - $y_{p,e}$ is the predicted output of a model estimated using estimated initial states.

  Applicable when `InitialCondition` is `'auto'`.

  Default: `1.05`

Output Arguments

`opt` — Option set for `procest`

`procestOptions` option set

Option set for `procest`, returned as a `procestOptions` option set.

Examples

Create Default Option Set for Process Model Estimation

```
opt = procestOptions;
```

Specify Options for Process Model Estimation

Create an option set for `procest` setting `Focus` to `'simulation'` and turning on the `Display`.

```
opt = procestOptions('Focus','simulation','Display','on');
```

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

```
opt = procestOptions;
opt.Focus = 'simulation';
opt.Display = 'on';
```

More About

- “Loss Function and Model Quality Metrics”

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

`idfilt` | `idproc` | `procest`

Introduced in R2012a

pwlinear

Create a piecewise-linear nonlinearity estimator object

Syntax

```
NL = pwlinear
NL = pwlinear(Name,Value)
```

Description

`NL = pwlinear` creates a default piecewise-linear nonlinearity estimator object with 10 break points for estimating Hammerstein-Wiener models. The value of the nonlinearity at the break points are set to `[]`. The initial value of the nonlinearity is determined from the estimation data range during estimation using `nlhw`. Use dot notation to customize the object properties, if needed.

`NL = pwlinear(Name,Value)` creates a piecewise-linear nonlinearity estimator object with properties specified by one or more `Name,Value` pair arguments. The properties that you do not specify retain their default value.

Object Description

`pwlinear` is an object that stores the piecewise-linear nonlinearity estimator for estimating Hammerstein-Wiener models.

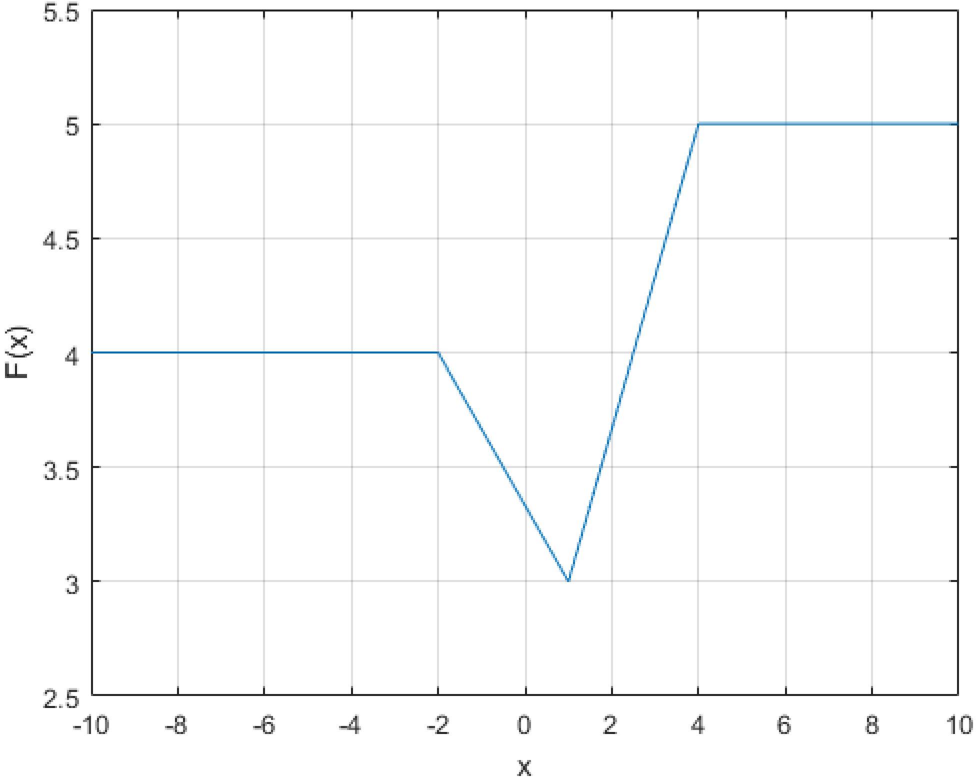
Use `pwlinear` to define a nonlinear function $y = F(x,\theta)$, where y and x are scalars, and θ represents the parameters specifying the number of break points and the value of nonlinearity at the break points.

The nonlinearity function, F, is a piecewise-linear (affine) function of x. There are `n` breakpoints (x_k,y_k), $k = 1,...,n$, such that $y_k = \mathrm{F}(x_k)$. F is linearly interpolated between the breakpoints.

F is also linear to the left and right of the extreme breakpoints. The slope of these extensions is a function of x_i and y_i breakpoints. The breakpoints are ordered by ascending `x`-values, which is important when you set a specific breakpoint to a different value.

There are minor difference between the breakpoint values you set and the values stored in the object because the toolbox has a different internal representation of breakpoints.

For example, in the following plot, the breakpoints are x_k = [-2,1,4] and the corresponding nonlinearity values are y_k = [4,3,5].



The value `F(x)` is computed by `evaluate(NL,x)`, where `NL` is the `pwlinear` object. When using `evaluate`, the break points have to be initialized manually.

For `pwlinear` object properties, see “Properties” on page 1-1103.

Examples

Create a Default Piecewise-Linear Nonlinearity Estimator

```
NL = pwlinear;
```

Specify the number of break points.

```
NL.NumberOfUnits = 5;
```

Estimate a Hammerstein Model with Piecewise-Linear Nonlinearity

Load estimation data.

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

Create a `pwlinear` object, and specify the breakpoints.

```
InputNL = pwlinear('BreakPoints',[-2,1,4]);
```

Since `BreakPoints` is specified as a vector, the specified vector is interpreted as the *x*-values of the break points. The *y*-values of the break points are set to 0, and are determined during model estimation.

Estimate model with no output nonlinearity.

```
sys = nlhw(z1,[2 3 0],InputNL,[]);
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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 `pwlinear` nonlinearity. For example, `NL= pwlinear('NumberofUnits',5)` creates a piecewise-linear nonlinearity estimator object with 5 breakpoints.

Properties

`pwlinear` object properties include:

`NumberofUnits`

Number of breakpoints, specified as an integer.

Default: `10`

`BreakPoints`

Break points, x_k, and the corresponding nonlinearity values at the breakpoints, y_k, specified as one of the following:

- 2-by-n matrix — The x and y values for each of the n break points are specified as $[x_1,x_2, ..., x_n;y_1, y_2, ..., y_n]$.
- 1-by-n vector — The specified vector is interpreted as the x values of the break points: $x_1,x_2, ..., x_n$. All the y values of the break points are set to `0`.

When the nonlinearity object is created, the breakpoints are ordered by ascending `x`-values. This is important to consider if you set a specific breakpoint to a different value after creating the object.

Default: `[]`

Output Arguments

`NL` — Piecewise-linear nonlinearity estimator object

`pwlinear` object

Piecewise-linear nonlinearity estimator object, returned as a `pwlinear` object.

See Also

`nlhw`

Introduced in R2007a

pzmap

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.

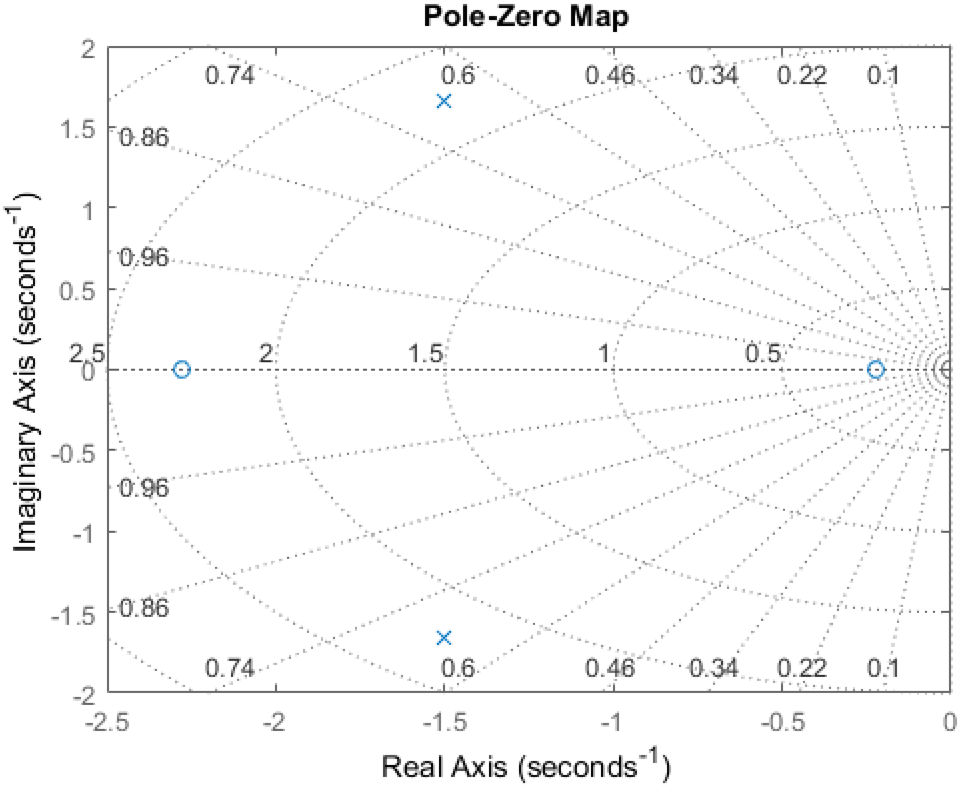
Examples

Pole-Zero Plot of Dynamic System

Plot the poles and zeros of the continuous-time system represented by the following transfer function:

$$H(s) = \frac{2s^2 + 5s + 1}{s^2 + 3s + 5}.$$

```
H = tf([2 5 1],[1 3 5]);
pzmap(H)
grid on
```



Turning on the grid displays lines of constant damping ratio (zeta) and lines of constant natural frequency (wn). This system has two real zeros, marked by o on the plot. The system also has a pair of complex poles, marked by x.

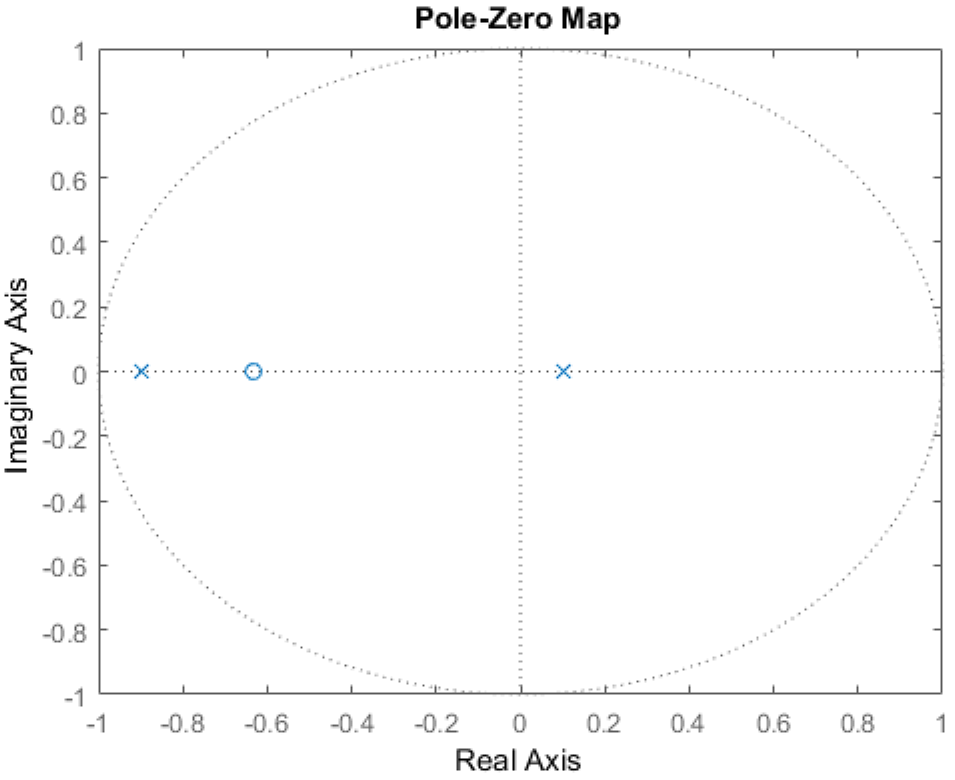
Pole-Zero Plot of Identified System

Plot the pole-zero map of a discrete time identified state-space (`idss`) model. In practice you can obtain an `idss` model by estimation based on input-output measurements of a system. For this example, create one from state-space data.

```
A = [0.1 0; 0.2 -0.9];
B = [.1 ; 0.1];
C = [10 5];
D = [0];
sys = idss(A,B,C,D,'Ts',0.1);
```

Examine the pole-zero map.

```
pzmap(sys)
```



System poles are marked by x, and zeros are marked by o.

More About

Tips

- For MIMO models, `pzmap` shows all system poles and transmission zeros on a single plot. To map poles and zeros for individual I/O pairs, use `iopzmap`.
- For additional options for customizing the appearance of the pole-zero plot, use `pzplot`.

Algorithms

`pzmap` uses a combination of `pole` and `zero`.

See Also

`damp` | `esort` | `dsort` | `pole` | `rlocus` | `sgrid` | `zgrid` | `zero` | `iopzmap` | `pzplot`

Introduced before R2006a

pzoptions

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 and System Identification 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 values: `'off' | 'on'` **Default**: `'off'`
GridColor	Color of the grid lines, specified as one of the following: Vector of RGB values in the range `[0,1]|` character vector of color name `|'none'`. For example, for yellow color, specify as one of the following: `[1 1 0]`, `'yellow'`, or `'y'`. **Default**: `[0.15,0.15,0.15]`
XlimMode, YlimMode	Limit modes

Option	**Description**
Xlim, Ylim	Axes limits
IOGrouping	Grouping of input-output pairs, specified as one of the following values: `'none'` |`'inputs'`|`'outputs'`|`'all'` **Default**: `'none'`
InputLabels, OutputLabels	Input and output label styles
InputVisible, OutputVisible	Visibility of input and output channels

Option	Description
FreqUnits	Frequency units, specified as one of the following values: • `'Hz'` • `'rad/second'` • `'rpm'` • `'kHz'` • `'MHz'` • `'GHz'` • `'rad/nanosecond'` • `'rad/microsecond'` • `'rad/millisecond'` • `'rad/minute'` • `'rad/hour'` • `'rad/day'` • `'rad/week'` • `'rad/month'` • `'rad/year'` • `'cycles/nanosecond'` • `'cycles/microsecond'` • `'cycles/millisecond'` • `'cycles/hour'` • `'cycles/day'` • `'cycles/week'` • `'cycles/month'` • `'cycles/year'` **Default**: `'rad/s'` You can also specify `'auto'` which uses frequency units `rad/TimeUnit` relative

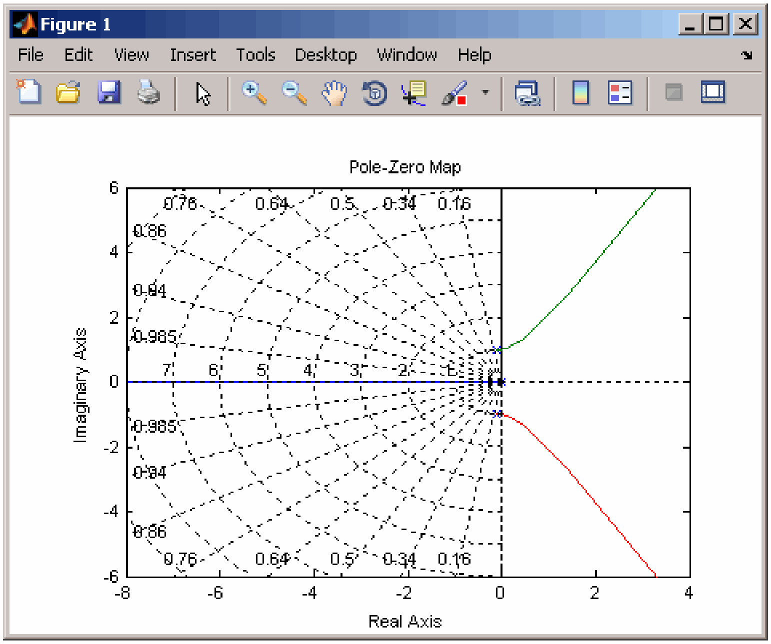
Option	Description
	to system time units specified in the `TimeUnit` property. For multiple systems with different time units, the units of the first system are used.
TimeUnits	Time units, specified as one of the following values: • `'nanoseconds'` • `'microseconds'` • `'milliseconds'` • `'seconds'` • `'minutes'` • `'hours'` • `'days'` • `'weeks'` • `'months'` • `'years'` **Default**: `'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.
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`

Introduced in R2012a

pzplot

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. For more information on the ways to change properties of your plots, see “Ways to Customize Plots”.

`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.

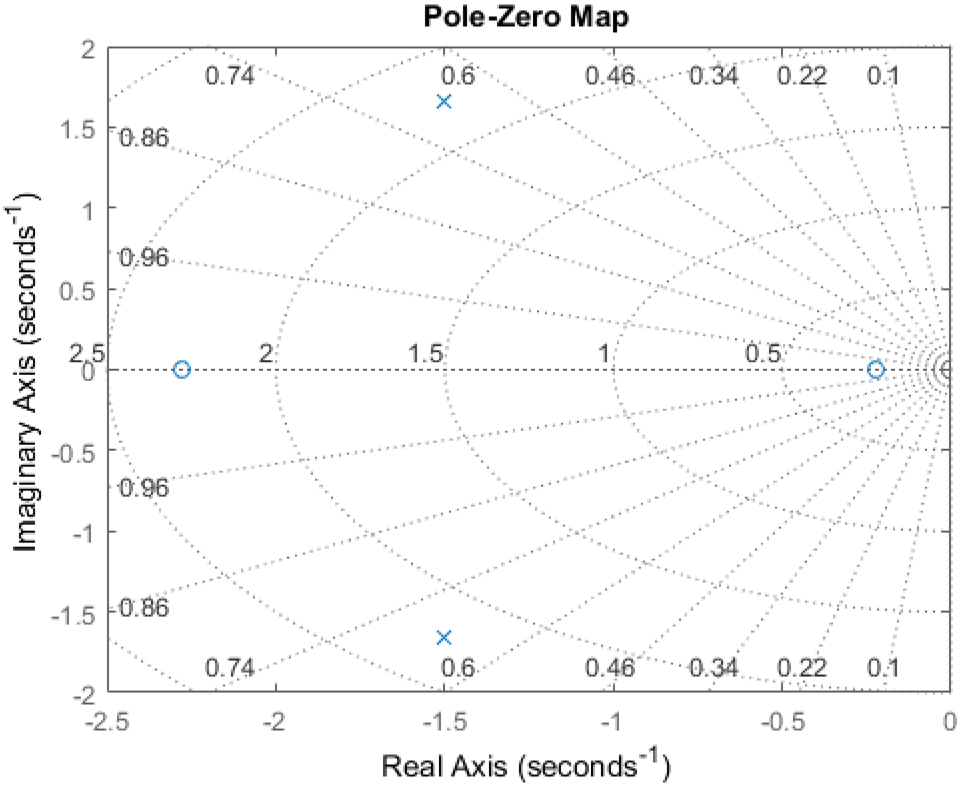
Examples

Pole-Zero Plot with Custom Plot Title

Plot the poles and zeros of the continuous-time system represented by the following transfer function:

$$sys\left(s\right)=\frac{2s^2+5s+1}{s^2+3s+5}.$$

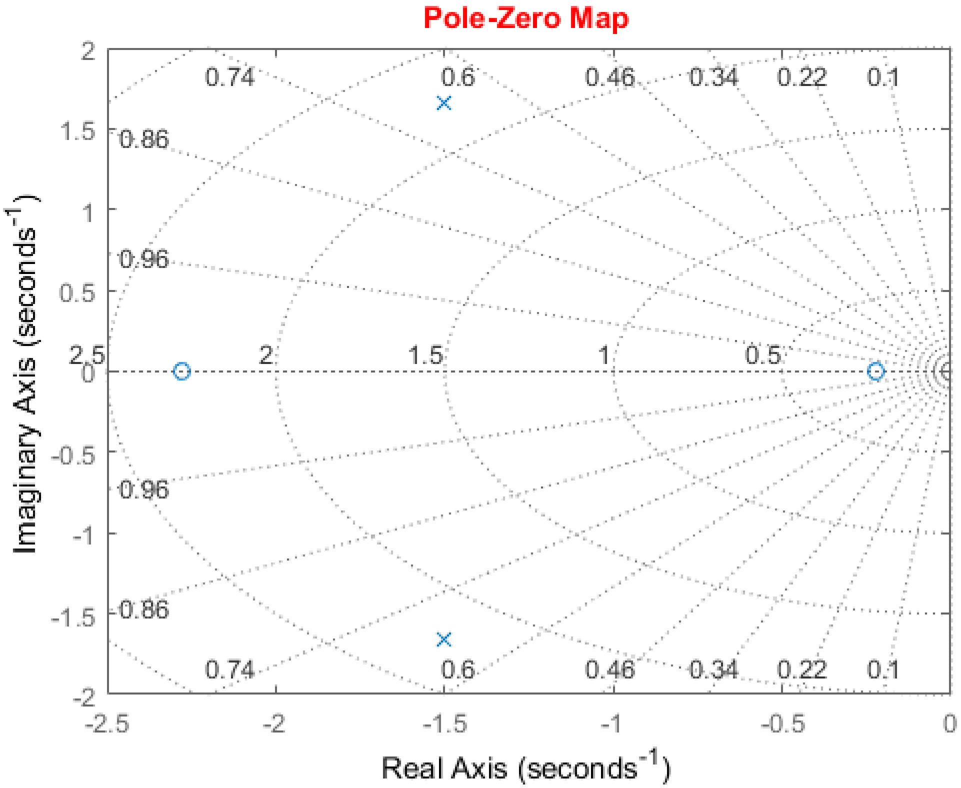
```
sys = tf([2 5 1],[1 3 5]);
h = pzplot(sys);
grid on
```



Turning on the grid displays lines of constant damping ratio (zeta) and lines of constant natural frequency (wn). This system has two real zeros, marked by o on the plot. The system also has a pair of complex poles, marked by x.

Change the color of the plot title. To do so, use the plot handle, `h`.

```
p = getoptions(h);
p.Title.Color = [1,0,0];
setoptions(h,p);
```



See Also

`getoptions` | `pzmap` | `setoptions` | `iopzplot`

Introduced in R2012a

rarmax

(To be removed) Estimate recursively parameters of ARMAX or ARMA models

---

Note: `rarmax` will be removed in a future release. Use `recursiveARMA` or `recursiveARMAX` instead.

Syntax

```
thm = rarmax(z,nn,adm,adg)
```

```
[thm,yhat,P,phi,psi] = rarmax(z,nn,adm,adg,th0,P0,phi0,psi0)
```

Description

The parameters of the ARMAX model structure

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

are estimated using a recursive prediction error method.

The input-output data is contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. `nn` is given as

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

where `na`, `nb`, and `nc` are the orders of the ARMAX model, and `nk` is the delay. Specifically,

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

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

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

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 `nk = 0`, shift the input sequence appropriately and use `nk = 1`.

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)
```

More About

Algorithms

The general recursive prediction error algorithm (11.44), (11.47) through (11.49) of Ljung (1999) is implemented. See “Recursive Algorithms for Online Parameter Estimation” for more information.

- “Recursive Algorithms for Online Parameter Estimation”

See Also

`nkshift` | `recursiveARMA` | `recursiveARMAX` | `rpem` | `rplr`

Introduced before R2006a

rarx

(To be removed) Estimate parameters of ARX or AR models recursively

---

Note: `rarx` will be removed in a future release. Use `recursiveAR` or `recursiveARX` instead.

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:

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

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

Models with several inputs are defined, as follows:

$$A(q)y(t) = B_1(q)u_1(t{-}nk_1)+\ldots+B_{nu}u_{nu}(t{-}nk_{nu})+e(t)$$

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 = [u1 ... unu]
```

`nn`

For input-output models, specifies the structure of the ARX model as:

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

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

For multiple-input models, `nb` and `nk` are row vectors that define orders and delays for each input.

For time-series models, `nn = na`, where `na` is the order of the AR model.

Note: The delay `nk` must be larger than `0`. If you want `nk = 0`, shift the input sequence appropriately and use `nk = 1` (see `nkshift`).

`adm` and `adg`

`adm = 'ff'` and `adg = lam` specify the *forgetting factor* algorithm with the forgetting factor λ=`lam`. This algorithm is also known as recursive least squares (RLS). In this case, the matrix `P` has the following interpretation: R_2`/2 * P` is approximately equal to the covariance matrix of the estimated parameters.R_2 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:

$\varphi(t) = [y(t-1),\ldots,y(t-na),u(t-1),\ldots,u(t-nb-nk+1)]$

If `z = [y(1),u(1); ... ;y(N),u(N)]`, phi0 = $\varphi(1)$ and 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:

$\varphi(t) = [y(t-1),...,y(t-na),u(t-1),...,u(t-nb-nk+1)]$

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 - 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.

More About

- “Recursive Algorithms for Online Parameter Estimation”

See Also

`nkshift` | `recursiveAR` | `recursiveARX` | `rpem` | `rplr`

Introduced before R2006a

rbj

(To be removed) Estimate recursively parameters of Box-Jenkins models

---

Note: `rbj` will be removed in a future release. Use `recursiveBJ` instead.

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,

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

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

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

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

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]
```

`yhat` is the predicted value of the output, according to the current model; that is, row `k` of `yhat` contains the predicted value of `y(k)` based on all past data.

The actual algorithm is selected with the two arguments `adm` and `adg`. These are described under `rarx`.

The input argument `th0` contains the initial value of the parameters, a row vector consistent with the rows of `thm`. The default value of `th0` is all zeros.

The arguments `P0` and `P` are the initial and final values, respectively, of the scaled covariance matrix of the parameters. See `rarx`. The default value of `P0` is 10^4 times the unit matrix. The arguments `phi0`, `psi0`, `phi`, and `psi` contain initial and final values of the data vector and the gradient vector, respectively. The sizes of these depend on the chosen model orders. The normal choice of `phi0` and `psi0` is to use the outputs from a previous call to `rbj` with the same model orders. (This call could be a dummy call with default input arguments.) The default values of `phi0` and `psi0` are all zeros.

Note that the function requires that the delay `nk` be larger than `0`. If you want `nk = 0`, shift the input sequence appropriately and use `nk = 1`.

More About

Algorithms

The general recursive prediction error algorithm (11.44) of Ljung (1999) is implemented. See also “Recursive Algorithms for Online Parameter Estimation”.

- “Recursive Algorithms for Online Parameter Estimation”

See Also

`nkshift` | `recursiveBJ` | `rpem` | `rplr`

Introduced before R2006a

realdata

Determine whether `iddata` is based on real-valued signals

Syntax

```
realdata(data)
```

Description

`realdata` returns 1 if

- `data` contains only real-valued signals.
- `data` contains frequency-domain signals, obtained by Fourier transformation of real-valued signals.

Otherwise `realdata` returns `0`.

Examples

Determine if Data is Based on Real-Valued Signals

Load data.

```
load iddata1
```

Transform the data to frequency domain.

```
zf = fft(z1);
```

Determine if the time-domain data values are real.

```
isreal(z1)
```

```
ans =

     1
```

Determine if the transformed data values are real.

```
isreal(zf)
```

```
ans =

     0
```

Determine if the data is based on real-valued signals.

```
realdata(zf)
```

```
ans =

  logical

   1
```

Add negative frequencies to `zf` and rerun the command.

```
zf = complex(zf);
realdata(zf)
```

```
ans =

  logical

   1
```

The command still returns 1.

Introduced before R2006a

recursiveAR

Create System object for online parameter estimation of AR model

Use `recursiveAR` command for parameter estimation with real-time data. If all data necessary for estimation is available at once, and you are estimating a time-invariant model, use the offline estimation command, `ar`.

Syntax

```
obj = recursiveAR
obj = recursiveAR(na)
obj = recursiveAR(na,A0)
obj = recursiveAR(___,Name,Value)
```

Description

`obj = recursiveAR` creates a System object for online parameter estimation of a default single output AR model structure. The default model structure has a polynomial of order 1 and initial polynomial coefficient value `eps`.

After creating the object, use the `step` command to update model parameter estimates using recursive estimation algorithms and real-time data.

`obj = recursiveAR(na)` specifies the polynomial order of the AR model to be estimated.

`obj = recursiveAR(na,A0)` specifies the polynomial order and initial values of the polynomial coefficients.

`obj = recursiveAR(___,Name,Value)` specifies additional attributes of the AR model structure and recursive estimation algorithm using one or more `Name,Value` pair arguments.

Object Description

`recursiveAR` creates a System object for online parameter estimation of single output AR models using a recursive estimation algorithm.

A System object is a specialized MATLAB object designed specifically for implementing and simulating dynamic systems with inputs that change over time. System objects use internal states to store past behavior, which is used in the next computational step.

After you create a System object, you use commands to process data or obtain information from or about the object. System objects use a minimum of two commands to process data — a constructor to create the object and the `step` command to update object parameters using real-time data. This separation of declaration from execution lets you create multiple, persistent, reusable objects, each with different settings.

You can use the following commands with the online estimation System objects in System Identification Toolbox:

Command	Description
`step`	Update model parameter estimates using recursive estimation algorithms and real-time data. `step` puts the object into a locked state. In a locked state, you cannot change any nontunable properties or input specifications, such as model order, data type, or estimation algorithm. During execution, you can only change tunable properties.
`release`	Unlock the System object. Use this command to enable setting of nontunable parameters.
`reset`	Reset the internal states of a locked System object to the initial values, and leave the object locked.
`clone`	Create another System object with the same object property values. Do not create additional objects using syntax `obj2 = obj`. Any changes made to the properties of the new object created this way (`obj2`) also change the properties of the original object (`obj`).

Command	Description
`isLocked`	Query locked status for input attributes and nontunable properties of the System object.

Use the `recursiveAR` command to create an online estimation System object. Then estimate the AR model parameter, `A`, and output using the `step` command with output data `y`.

```
[A,EstimatedOutput] = step(obj,y)
```

For `recursiveAR` object properties, see “Properties” on page 1-1135.

Examples

Estimate AR Model Online

Create a System object™ for online parameter estimation of an AR model using recursive estimation algorithms.

```
obj = recursiveAR;
```

The AR model has a default structure with polynomial of order 1 and initial polynomial coefficient values, `eps`.

Load the time-series estimation data. In this example, use a static data set for illustration.

```
load iddata9 z9;
output = z9.y;
```

Estimate AR model parameters online using `step`.

```
for i = 1:numel(output)
[A,EstimatedOutput] = step(obj,output(i));
end
```

View the current estimated values of polynomial `A` coefficients.

```
obj.A
```

```
ans =
```

```
    1.0000   -0.9592
```

View the current covariance estimate of the parameters.

```
obj.ParameterCovariance
```

```
ans =

   1.6204e-04
```

View the current estimated output.

```
EstimatedOutput
```

```
EstimatedOutput =

    0.7830
```

Create Online Estimation System Object for AR Model With Known Polynomial Order

Specify AR model polynomial order.

```
na = 2;
```

Create a System object™ for online estimation of an AR model with the specified polynomial order.

```
obj = recursiveAR(na);
```

Create Online Estimation System Object for AR Model With Known Initial Parameters

Specify AR model order.

```
na = 2;
```

Create a System object for online estimation of AR model with known initial polynomial coefficients.

```
A0 = [1 0.5 0.3];
obj = recursiveAR(na,A0);
```

Specify the initial parameter covariance.

```
obj.InitialParameterCovariance = 0.1;
```

`InitialParameterCovariance` represents the uncertainty in your guess for the initial parameters. Typically, the default `InitialParameterCovariance` (10000) is too large relative to the parameter values. This results in initial guesses being given less importance during estimation. If you have confidence in the initial parameter guesses, specify a smaller initial parameter covariance.

Specify Estimation Method for Online Estimation of AR Model

Create a System object that uses the normalized gradient algorithm for online parameter estimation of an AR model.

```
obj = recursiveAR(2,'EstimationMethod','NormalizedGradient');
```

- "Perform Online Parameter Estimation at the Command Line"
- "Validate Online Parameter Estimation at the Command Line"

Input Arguments

na — Model order

positive integer

Model order of the polynomial $A(q)$ of an AR model, specified as a positive integer.

A0 — Initial value of polynomial coefficients

row vector of real values | `[]`

Initial value of coefficients of the polynomial $A(q)$, specified as a 1-by-(`na+1`) row vector of real values with 1 as the first element. Specify the coefficients in order of ascending powers of q^{-1}.

Specifying as `[]`, uses the default value of `eps` for the polynomial coefficients.

Note: If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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 writable properties of `recursiveAR` System object during object creation. For example, `obj = recursiveAR(2,'EstimationMethod','Gradient')` creates a System object to estimate an AR model using the `'Gradient'` recursive estimation algorithm.

Properties

`recursiveAR` System object properties consist of read-only and writable properties. The writable properties are tunable and nontunable properties. The nontunable properties cannot be changed when the object is locked, that is, after you use the `step` command.

Use `Name,Value` arguments to specify writable properties of `recursiveAR` objects during object creation. After object creation, use dot notation to modify the tunable properties.

```
obj = recursiveAR;
obj.ForgettingFactor = 0.99;
```

A

Estimated coefficients of polynomial $A(q)$, returned as a row vector of real values specified in order of ascending powers of q^{-1}.

`A` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

InitialA

Initial values for the coefficients of polynomial $A(q)$ of order `na`, specified as a row vector of length `na+1`, with 1 as the first element. Specify the coefficients in order of ascending powers of q^{-1}.

If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

`InitialA` is a tunable property. You can change it when the object is in a locked state.

Default: `[1 eps]`

`ParameterCovariance`

Estimated covariance of the parameters, returned as an N-by-N symmetric positive-definite matrix. N is the number of parameters to be estimated.

Applicable when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'`.

`ParameterCovariance` is computed assuming that the residuals (difference between estimated and measured outputs) are white noise, and the variance of these residuals is 1.

`ParameterCovariance` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

`InitialParameterCovariance`

Covariance of the initial parameter estimates, specified as one of the following:

- Real positive scalar, a — Covariance matrix is an N-by-N diagonal matrix, with a as the diagonal elements. N is the number of parameters to be estimated.
- Vector of real positive scalars, $[a_1,...,a_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[a_1,...,a_N]$ as the diagonal elements.
- N-by-N symmetric positive-definite matrix.

`InitialParameterCovariance` represents the uncertainty in the initial parameter estimates. For large values of `InitialParameterCovariance`, less importance is placed on the initial parameter values and more on the measured data during beginning of estimation using `step`.

Use only when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'`.

`InitialParameterCovariance` is a tunable property. You can change it when the object is in a locked state.

Default: `10000`

`EstimationMethod`

Recursive estimation algorithm used for online estimation of model parameters, specified as one of the following values:

- `'ForgettingFactor'` — Algorithm used for parameter estimation
- `'KalmanFilter'` — Algorithm used for parameter estimation
- `'NormalizedGradient'` — Algorithm used for parameter estimation
- `'Gradient'` — Unnormalized gradient algorithm used for parameter estimation

Forgetting factor and Kalman filter algorithms are more computationally intensive than gradient and unnormalized gradient methods. However, they have better convergence properties. For information about these algorithms, see “Recursive Algorithms for Online Parameter Estimation”.

`EstimationMethod` is a nontunable property. You cannot change it during execution, that is after the object is locked using the `step` command. If you want to deploy code using MATLAB Coder, `EstimationMethod` can only be assigned once.

Default: `'ForgettingFactor'`

`ForgettingFactor`

Forgetting factor, λ, relevant for parameter estimation, specified as a scalar in the range (0,1].

Suppose that the system remains approximately constant over T_0 samples. You can choose λ such that:

$$T_0 = \frac{1}{1-\lambda}$$

- Setting $\lambda = 1$ corresponds to “no forgetting” and estimating constant coefficients.
- Setting $\lambda < 1$ implies that past measurements are less significant for parameter estimation and can be “forgotten”. Set $\lambda < 1$ to estimate time-varying coefficients.

Typical choices of λ are in the range `[0.98 0.995]`.

Use only when `EstimationMethod` is `'ForgettingFactor'`.

`ForgettingFactor` is a tunable property. You can change it when the object is in a locked state.

Default: `1`

EnableAdapation

Enable or disable parameter estimation, specified as one of the following:

- `true` or `1`— The `step` command estimates the parameter values for that time step and updates the parameter values.
- `false` or `0` — The `step` command does not update the parameters for that time step and instead outputs the last estimated value. You can use this option when your system enters a mode where the parameter values do not vary with time.

Note: If you set `EnableAdapation` to `false`, you must still execute the `step` command. Do not skip `step` to keep parameter values constant, because parameter estimation depends on current and past I/O measurements. `step` ensures past I/O data is stored, even when it does not update the parameters.

`EnableAdapation` is a tunable property. You can change it when the object is in a locked state.

Default: `true`

DataType

Floating point precision of parameters, specified as one of the following values:

- `'double'` — Double-precision floating point
- `'single'` — Single-precision floating point

Setting `DataType` to `'single'` saves memory, but leads to loss of precision. Specify `DataType` based on the precision required by the target processor where you will deploy generated code.

`DataType` is a nontunable property. It can only be set during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `'double'`

`ProcessNoiseCovariance`

Covariance matrix of parameter variations, specified as one of the following:

- Real nonnegative scalar, a — Covariance matrix is an N-by-N diagonal matrix, with a as the diagonal elements.
- Vector of real nonnegative scalars, $[a_1,...,a_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[a_1,...,a_N]$ as the diagonal elements.
- N-by-N symmetric positive semidefinite matrix.

N is the number of parameters to be estimated.

`ProcessNoiseCovariance` is applicable when `EstimationMethod` is `'KalmanFilter'`.

Kalman filter algorithm treats the parameters as states of a dynamic system and estimates these parameters using a Kalman filter. `ProcessNoiseCovariance` is the covariance of the process noise acting on these parameters. Zero values in the noise covariance matrix correspond to estimating constant coefficients. Values larger than 0 correspond to time-varying parameters. Use large values for rapidly changing parameters. However, the larger values result in noisier parameter estimates.

`ProcessNoiseCovariance` is a tunable property. You can change it when the object is in a locked state.

Default: `0.1`

`AdaptationGain`

Adaptation gain, γ, used in gradient recursive estimation algorithms, specified as a positive scalar.

`AdaptationGain` is applicable when `EstimationMethod` is `'Gradient'` or `'NormalizedGradient'`.

Specify a large value for `AdaptationGain` when your measurements have a high signal-to-noise ratio.

`AdaptationGain` is a tunable property. You can change it when the object is in a locked state.

Default: `1`

`NormalizationBias`

Bias in adaptation gain scaling used in the `'NormalizedGradient'` method, specified as a nonnegative scalar.

`NormalizationBias` is applicable when `EstimationMethod` is `'NormalizedGradient'`.

The normalized gradient algorithm divides the adaptation gain at each step by the square of the two-norm of the gradient vector. If the gradient is close to zero, this can cause jumps in the estimated parameters. `NormalizationBias` is the term introduced in the denominator to prevent these jumps. Increase `NormalizationBias` if you observe jumps in estimated parameters.

`NormalizationBias` is a tunable property. You can change it when the object is in a locked state.

Default: `eps`

Output Arguments

`obj` — System object for online parameter estimation of AR model

`recursiveAR` System object

System object for online parameter estimation of AR model, returned as a `recursiveAR` System object. This object is created using the specified model orders and properties. Use `step` command to estimate the coefficients of the AR model polynomials. You can then access the estimated coefficients and parameter covariance using dot notation. For example, type `obj.A` to view the estimated A polynomial coefficients.

More About

AR Model Structure

The AR model structure is:

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

where,

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

Here,

- *y(t)*— Output at time *t*. Data is a time series that has no input channels and one output channel.
- *na* — Number of *A* polynomial coefficients
- *e(t)* — White-noise disturbance value at time *t*
- q^{-1} — Time-shift operator

Tips

- Starting in R2016b, instead of using the `step` command to update model parameter estimates, you can call the System object with input arguments, as if it were a function. For example, `[A,EstimatedOutput] = step(obj,y)` and `[A,EstimatedOutput] = obj(y)` perform equivalent operations.
- “What Is Online Estimation?”
- “Recursive Algorithms for Online Parameter Estimation”

See Also

`ar` | `clone` | `isLocked` | `Recursive Polynomial Model Estimator` | `recursiveARMA` | `recursiveARMAX` | `recursiveARX` | `recursiveBJ` | `recursiveLS` | `recursiveOE` | `release` | `reset` | `step`

Introduced in R2015b

recursiveARMA

Create System object for online parameter estimation of ARMA model

Use `recursiveARMA` command for parameter estimation with real-time data. If all data necessary for estimation is available at once, and you are estimating a time-invariant model, use the offline estimation command, `armax`.

Syntax

```
obj = recursiveARMA
obj = recursiveARMA(Orders)
obj = recursiveARMA(Orders,A0,C0)
obj = recursiveARMA(___,Name,Value)
```

Description

`obj = recursiveARMA` creates a System object for online parameter estimation of a default single output ARMA model structure. The default model structure has polynomials of order 1 and initial polynomial coefficient values `eps`.

After creating the object, use the `step` command to update model parameter estimates using recursive estimation algorithms and real-time data.

`obj = recursiveARMA(Orders)` specifies the polynomial orders of the ARMA model to be estimated.

`obj = recursiveARMA(Orders,A0,C0)` specifies the polynomial orders and initial values of the polynomial coefficients. Specify initial values to potentially avoid local minima during estimation. If the initial values are small compared to the default `InitialParameterCovariance` property value, and you have confidence in your initial values, also specify a smaller `InitialParameterCovariance`.

`obj = recursiveARMA(___,Name,Value)` specifies additional attributes of the ARMA model structure and recursive estimation algorithm using one or more `Name,Value` pair arguments.

Object Description

`recursiveARMA` creates a System object for online parameter estimation of single output ARMA models using a recursive estimation algorithm.

A System object is a specialized MATLAB object designed specifically for implementing and simulating dynamic systems with inputs that change over time. System objects use internal states to store past behavior, which is used in the next computational step.

After you create a System object, you use commands to process data or obtain information from or about the object. System objects use a minimum of two commands to process data — a constructor to create the object and the `step` command to update object parameters using real-time data. This separation of declaration from execution lets you create multiple, persistent, reusable objects, each with different settings.

You can use the following commands with the online estimation System objects in System Identification Toolbox:

Command	Description
`step`	Update model parameter estimates using recursive estimation algorithms and real-time data. `step` puts the object into a locked state. In a locked state, you cannot change any nontunable properties or input specifications, such as model order, data type, or estimation algorithm. During execution, you can only change tunable properties.
`release`	Unlock the System object. Use this command to enable setting of nontunable parameters.
`reset`	Reset the internal states of a locked System object to the initial values, and leave the object locked.
`clone`	Create another System object with the same object property values.

Command	Description
	Do not create additional objects using syntax `obj2 = obj`. Any changes made to the properties of the new object created this way (`obj2`) also change the properties of the original object (`obj`).
`isLocked`	Query locked status for input attributes and nontunable properties of the System object.

Use the `recursiveARMA` command to create an online estimation System object. Then estimate the ARMA model parameters (`A` and `C`) and output using the `step` command with output data `y`.

```
[A,C,EstimatedOutput] = step(obj,y)
```

For `recursiveARMA` object properties, see “Properties” on page 1-1147.

Examples

Estimate ARMA Model Online

Create a System object for online parameter estimation of an ARMA model.

```
obj = recursiveARMA;
```

The ARMA model has a default structure with polynomials of order 1 and initial polynomial coefficient values, `eps`.

Load the time-series estimation data. In this example, use a static data set for illustration.

```
load iddata9 z9;
output = z9.y;
```

Estimate ARMA model parameters online using `step`.

```
for i = 1:numel(output)
[A,C,EstimatedOutput] = step(obj,output(i));
end
```

View the current estimated values of polynomial `C` coefficients.

```
obj.C
```

```
ans =

    1.0000    0.2315
```

View the current covariance estimate of the parameters.

```
obj.ParameterCovariance
```

```
ans =

   1.0e-03 *

    0.6372   -0.0257
   -0.0257    0.0017
```

View the current estimated output.

```
EstimatedOutput
```

```
EstimatedOutput =

   11.8121
```

Create Online Estimation System Object for ARMA Model With Known Orders

Specify ARMA model orders.

```
na = 2;
nc = 1;
```

Create a System object for online estimation of an ARMA model with the specified orders.

```
obj = recursiveARMA([na nc]);
```

Create Online Estimation System Object for ARMA Model With Known Initial Parameters

Specify ARMA model orders.

```
na = 2;
nc = 1;
```

Create a System object for online estimation of ARMA model with known initial polynomial coefficients.

```
A0 = [1 0.5 0.3];
C0 = [1 0.7];
obj = recursiveARMA([na nc],A0,C0);
```

Specify the initial parameter covariance.

```
obj.InitialParameterCovariance = 0.1;
```

`InitialParameterCovariance` represents the uncertainty in your guess for the initial parameters. Typically, the default `InitialParameterCovariance` (10000) is too large relative to the parameter values. This results in initial guesses being given less importance during estimation. If you have confidence in the initial parameter guesses, specify a smaller initial parameter covariance.

Specify Estimation Method for Online Estimation of ARMA Model

Create a System object that uses the unnormalized gradient algorithm for online parameter estimation of an ARMA model.

```
obj = recursiveARMA([2 1],'EstimationMethod','Gradient');
```

- “Perform Online Parameter Estimation at the Command Line”
- “Validate Online Parameter Estimation at the Command Line”

Input Arguments

`Orders` — Model orders

1-by-2 vector of integers

Model orders of an ARMA model, specified as a 1-by-2 vector of integers, `[na nc]`.

- `na` — Order of the polynomial $A(q)$, specified as a nonnegative integer.
- `nc` — Order of the polynomial $C(q)$, specified as a nonnegative integer.

`A0,C0` — Initial value of polynomial coefficients

row vectors of real values | [ ]

Initial value of polynomial coefficients, specified as row vectors of real values with elements in order of ascending powers of q^{-1}.

- `A0` — Initial guess for the coefficients of the polynomial $A(q)$, specified as a 1-by-(`na+1`) vector with 1 as the first element.
- `C0` — Initial guess for the coefficients of the polynomial $C(q)$, specified as a 1-by-(`nc+1`) vector with 1 as the first element.

  The coefficients in `C0` must define a stable discrete-time polynomial with roots within a unit disk. For example,

  ```
  C0 = [1 0.5 0.5];
  all(abs(roots(C0))<1)
  ```

  ```
  ans =

       1
  ```

Specifying as `[]`, uses the default value of `eps` for the polynomial coefficients.

Note: If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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 writable properties of `recursiveARMA` System object during object creation. For example, `obj = recursiveARMA([2 1],'EstimationMethod','Gradient')` creates a System object to estimate an ARMA model using the `'Gradient'` recursive estimation algorithm.

Properties

`recursiveARMA` System object properties consist of read-only and writable properties. The writable properties are tunable and nontunable properties. The nontunable properties cannot be changed when the object is locked, that is, after you use the `step` command.

Use `Name,Value` arguments to specify writable properties of `recursiveARMA` objects during object creation. After object creation, use dot notation to modify the tunable properties.

```
obj = recursiveARMA;
obj.ForgettingFactor = 0.99;
```

A

Estimated coefficients of polynomial $A(q)$, returned as a row vector of real values specified in order of ascending powers of q^{-1}.

`A` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

C

Estimated coefficients of polynomial $C(q)$, returned as a vector of real values specified in order of ascending powers of q^{-1}.

`C` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

InitialA

Initial values for the coefficients of polynomial $A(q)$ of order `na`, specified as a row vector of length `na+1`, with 1 as the first element. Specify the coefficients in order of ascending powers of q^{-1}.

If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

`InitialA` is a tunable property. You can change it when the object is in a locked state.

Default: `[1 eps]`

InitialC

Initial values for the coefficients of polynomial $C(q)$ of order `nc`, specified as a row vector of length `nc+1`, with 1 as the first element. Specify the coefficients in order of ascending powers of q^{-1}.

The coefficients in `InitialC` must define a stable discrete-time polynomial with roots within a unit circle. For example,

```
InitialC = [1 0.5 0.5];
all(abs(roots(InitialC))<1)

ans =

     1
```

If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

`InitialC` is a tunable property. You can change it when the object is in a locked state.

Default: `[1 eps]`

ParameterCovariance

Estimated covariance of the parameters, returned as an N-by-N symmetric positive-definite matrix. N is the number of parameters to be estimated.

Applicable when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'`.

`ParameterCovariance` is computed assuming that the residuals (difference between estimated and measured outputs) are white noise, and the variance of these residuals is 1.

`ParameterCovariance` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

InitialParameterCovariance

Covariance of the initial parameter estimates, specified as one of the following:

- Real positive scalar, a — Covariance matrix is an N-by-N diagonal matrix, with a as the diagonal elements. N is the number of parameters to be estimated.
- Vector of real positive scalars, $[a_1,...,a_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[a_1,...,a_N]$ as the diagonal elements.
- N-by-N symmetric positive-definite matrix.

`InitialParameterCovariance` represents the uncertainty in the initial parameter estimates. For large values of `InitialParameterCovariance`, less importance is placed on the initial parameter values and more on the measured data during beginning of estimation using `step`.

Use only when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'`.

`InitialParameterCovariance` is a tunable property. You can change it when the object is in a locked state.

Default: `10000`

`EstimationMethod`

Recursive estimation algorithm used for online estimation of model parameters, specified as one of the following values:

- `'ForgettingFactor'` — Algorithm used for parameter estimation
- `'KalmanFilter'` — Algorithm used for parameter estimation
- `'NormalizedGradient'` — Algorithm used for parameter estimation
- `'Gradient'` — Unnormalized gradient algorithm used for parameter estimation

Forgetting factor and Kalman filter algorithms are more computationally intensive than gradient and unnormalized gradient methods. However, they have better convergence properties. For information about these algorithms, see "Recursive Algorithms for Online Parameter Estimation".

`EstimationMethod` is a nontunable property. You cannot change it during execution, that is after the object is locked using the `step` command. If you want to deploy code using MATLAB Coder, `EstimationMethod` can only be assigned once.

Default: `'ForgettingFactor'`

`ForgettingFactor`

Forgetting factor, λ, relevant for parameter estimation, specified as a scalar in the range (0,1].

Suppose that the system remains approximately constant over T_0 samples. You can choose λ such that:

$$T_0 = \frac{1}{1-\lambda}$$

- Setting $\lambda = 1$ corresponds to “no forgetting” and estimating constant coefficients.
- Setting $\lambda < 1$ implies that past measurements are less significant for parameter estimation and can be “forgotten”. Set $\lambda < 1$ to estimate time-varying coefficients.

Typical choices of λ are in the range `[0.98 0.995]`.

Use only when `EstimationMethod` is `'ForgettingFactor'`.

`ForgettingFactor` is a tunable property. You can change it when the object is in a locked state.

Default: `1`

EnableAdapation

Enable or disable parameter estimation, specified as one of the following:

- `true` or `1`— The `step` command estimates the parameter values for that time step and updates the parameter values.
- `false` or `0` — The `step` command does not update the parameters for that time step and instead outputs the last estimated value. You can use this option when your system enters a mode where the parameter values do not vary with time.

Note: If you set `EnableAdapation` to `false`, you must still execute the `step` command. Do not skip `step` to keep parameter values constant, because parameter estimation depends on current and past I/O measurements. `step` ensures past I/O data is stored, even when it does not update the parameters.

`EnableAdapation` is a tunable property. You can change it when the object is in a locked state.

Default: `true`

DataType

Floating point precision of parameters, specified as one of the following values:

- `'double'` — Double-precision floating point

- `'single'` — Single-precision floating point

Setting `DataType` to `'single'` saves memory, but leads to loss of precision. Specify `DataType` based on the precision required by the target processor where you will deploy generated code.

`DataType` is a nontunable property. It can only be set during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `'double'`

`ProcessNoiseCovariance`

Covariance matrix of parameter variations, specified as one of the following:

- Real nonnegative scalar, a — Covariance matrix is an N-by-N diagonal matrix, with a as the diagonal elements.
- Vector of real nonnegative scalars, $[a_1,...,a_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[a_1,...,a_N]$ as the diagonal elements.
- N-by-N symmetric positive semidefinite matrix.

N is the number of parameters to be estimated.

`ProcessNoiseCovariance` is applicable when `EstimationMethod` is `'KalmanFilter'`.

Kalman filter algorithm treats the parameters as states of a dynamic system and estimates these parameters using a Kalman filter. `ProcessNoiseCovariance` is the covariance of the process noise acting on these parameters. Zero values in the noise covariance matrix correspond to estimating constant coefficients. Values larger than 0 correspond to time-varying parameters. Use large values for rapidly changing parameters. However, the larger values result in noisier parameter estimates.

`ProcessNoiseCovariance` is a tunable property. You can change it when the object is in a locked state.

Default: `0.1`

`AdaptationGain`

Adaptation gain, γ, used in gradient recursive estimation algorithms, specified as a positive scalar.

`AdaptationGain` is applicable when `EstimationMethod` is `'Gradient'` or `'NormalizedGradient'`.

Specify a large value for `AdaptationGain` when your measurements have a high signal-to-noise ratio.

`AdaptationGain` is a tunable property. You can change it when the object is in a locked state.

Default: `1`

`NormalizationBias`

Bias in adaptation gain scaling used in the `'NormalizedGradient'` method, specified as a nonnegative scalar.

`NormalizationBias` is applicable when `EstimationMethod` is `'NormalizedGradient'`.

The normalized gradient algorithm divides the adaptation gain at each step by the square of the two-norm of the gradient vector. If the gradient is close to zero, this can cause jumps in the estimated parameters. `NormalizationBias` is the term introduced in the denominator to prevent these jumps. Increase `NormalizationBias` if you observe jumps in estimated parameters.

`NormalizationBias` is a tunable property. You can change it when the object is in a locked state.

Default: `eps`

Output Arguments

`obj` — System object for online parameter estimation of ARMA model

`recursiveARMA` System object

System object for online parameter estimation of ARMA model, returned as a `recursiveARMA` System object. This object is created using the specified model orders and properties. Use `step` command to estimate the coefficients of the ARMA model polynomials. You can then access the estimated coefficients and parameter covariance using dot notation. For example, type `obj.A` to view the estimated *A* polynomial coefficients.

More About

ARMA Model Structure

The ARMA model structure is:

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

where,

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

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

Here,

- *y(t)*— Output at time *t*. Data is a time series that has no input channels and one output channel.
- *na* — Number of *A* polynomial coefficients
- *nc* — Number of *C* polynomial coefficients
- *e(t)* — White-noise disturbance value at time *t*
- q^{-1} — Time-shift operator

Tips

- Starting in R2016b, instead of using the `step` command to update model parameter estimates, you can call the System object with input arguments, as if it were a function. For example, `[A,C,EstimatedOutput] = step(obj,y)` and `[A,C,EstimatedOutput] = obj(y)` perform equivalent operations.

- “What Is Online Estimation?”
- “Recursive Algorithms for Online Parameter Estimation”

See Also

`armax` | `clone` | `isLocked` | `Recursive Polynomial Model Estimator` | `recursiveAR` | `recursiveARMAX` | `recursiveARX` | `recursiveBJ` | `recursiveLS` | `recursiveOE` | `release` | `reset` | `step`

Introduced in R2015b

recursiveARMAX

Create System object for online parameter estimation of ARMAX model

Use `recursiveARMAX` command for parameter estimation with real-time data. If all data necessary for estimation is available at once, and you are estimating a time-invariant model, use the offline estimation command, `armax`.

Syntax

```
obj = recursiveARMAX
obj = recursiveARMAX(Orders)
obj = recursiveARMAX(Orders,A0,B0,C0)
obj = recursiveARMAX(___,Name,Value)
```

Description

`obj = recursiveARMAX` creates a System object for online parameter estimation of default single-input single-output (SISO) ARMAX model structure. The default model structure has polynomials of order 1 and initial polynomial coefficient values `eps`.

After creating the object, use the `step` command to update model parameter estimates using recursive estimation algorithms and real-time data.

`obj = recursiveARMAX(Orders)` specifies the polynomial orders of the ARMAX model to be estimated.

`obj = recursiveARMAX(Orders,A0,B0,C0)` specifies the polynomial orders and initial values of the polynomial coefficients. Specify initial values to potentially avoid local minima during estimation. If the initial values are small compared to the default `InitialParameterCovariance` property value, and you have confidence in your initial values, also specify a smaller `InitialParameterCovariance`.

`obj = recursiveARMAX(___,Name,Value)` specifies additional attributes of the ARMAX model structure and recursive estimation algorithm using one or more `Name,Value` pair arguments.

Object Description

`recursiveARMAX` creates a System object for online parameter estimation of SISO ARMAX models using a recursive estimation algorithm.

A System object is a specialized MATLAB object designed specifically for implementing and simulating dynamic systems with inputs that change over time. System objects use internal states to store past behavior, which is used in the next computational step.

After you create a System object, you use commands to process data or obtain information from or about the object. System objects use a minimum of two commands to process data — a constructor to create the object and the `step` command to update object parameters using real-time data. This separation of declaration from execution lets you create multiple, persistent, reusable objects, each with different settings.

You can use the following commands with the online estimation System objects in System Identification Toolbox:

Command	Description
`step`	Update model parameter estimates using recursive estimation algorithms and real-time data. `step` puts the object into a locked state. In a locked state, you cannot change any nontunable properties or input specifications, such as model order, data type, or estimation algorithm. During execution, you can only change tunable properties.
`release`	Unlock the System object. Use this command to enable setting of nontunable parameters.
`reset`	Reset the internal states of a locked System object to the initial values, and leave the object locked.
`clone`	Create another System object with the same object property values.

Command	Description
	Do not create additional objects using syntax `obj2 = obj`. Any changes made to the properties of the new object created this way (`obj2`) also change the properties of the original object (`obj`).
`isLocked`	Query locked status for input attributes and nontunable properties of the System object.

Use the `recursiveARMAX` command to create an online estimation System object. Then estimate the ARMAX model parameters (`A`, `B`, and `C`) and output using the `step` command with incoming input and output data, `u`, and `y`.

```
[A,B,C,EstimatedOutput] = step(obj,y,u)
```

For `recursiveARMAX` object properties, see “Properties” on page 1-1162.

Examples

Estimate an ARMAX Model Online

Create a System object for online parameter estimation of an ARMAX model.

```
obj = recursiveARMAX;
```

The ARMAX model has a default structure with polynomials of order 1 and initial polynomial coefficient values, `eps`.

Load the estimation data. In this example, use a static data set for illustration.

```
load iddata1 z1;
output = z1.y;
input = z1.u;
```

Estimate ARMAX model parameters online using `step`.

```
for i = 1:numel(input)
[A,B,C,EstimatedOutput] = step(obj,output(i),input(i));
end
```

View the current estimated values of polynomial `A` coefficients.

```
obj.A


ans =

    1.0000   -0.8298
```

View the current covariance estimate of the parameters.

```
obj.ParameterCovariance


ans =

    0.0001    0.0001    0.0001
    0.0001    0.0032    0.0000
    0.0001    0.0000    0.0001
```

View the current estimated output.

```
EstimatedOutput


EstimatedOutput =

   -4.5595
```

Create System Object for ARMAX Model With Known Polynomial Orders

Specify ARMAX model orders and delays.

```
na = 1;
nb = 2;
nc = 1;
nk = 1;
```

Create a System object for online estimation of ARMAX model with the specified orders and delays.

```
obj = recursiveARMAX([na nb nc nk]);
```

Create Online Estimation System Object for ARMAX Model With Known Initial Parameters

Specify ARMAX model orders and delays.

```
na = 1;
nb = 2;
nc = 1;
nk = 1;
```

Create a System object for online estimation of ARMAX model with known initial polynomial coefficients.

```
A0 = [1 0.5];
B0 = [0 1 1];
C0 = [1 0.5];
obj = recursiveARMAX([na nb nc nk],A0,B0,C0);
```

Specify the initial parameter covariance.

```
obj.InitialParameterCovariance = 0.1;
```

`InitialParameterCovariance` represents the uncertainty in your guess for the initial parameters. Typically, the default `InitialParameterCovariance` (10000) is too large relative to the parameter values. This results in initial guesses being given less importance during estimation. If you have confidence in the initial parameter guesses, specify a smaller initial parameter covariance.

Specify Estimation Method for Online Estimation of ARMAX Model

Create a System object that uses the Kalman filter algorithm for online parameter estimation of an ARMAX model.

```
obj = recursiveARMAX([1 2 1 1],'EstimationMethod','KalmanFilter');
obj.ProcessNoiseCovariance = 0.01;
```

`ProcessNoiseCovariance` property of `obj` is applicable only when Kalman filter algortihm is used for estimation.

- “Perform Online Parameter Estimation at the Command Line”
- “Validate Online Parameter Estimation at the Command Line”

Input Arguments

`Orders` — Model orders and delays

1-by-4 vector of integers

Model orders and delays of an ARMAX model, specified as a 1-by-4 vector of integers, `[na nb nc nk]`.

- `na` — Order of the polynomial *A*(*q*), specified as a nonnegative integer. `na` represents the number of poles in your system.
- `nb` — Order of the polynomial *B*(*q*) + 1, specified as a positive integer. `nb` represents the number of zeroes in your system plus 1.
- `nc` — Order of the polynomial *C*(*q*), specified as a nonnegative integer.
- `nk` — Input-output delay, specified as a nonnegative integer. `nk` is number of input samples that occur before the input affects the output. `nk` is expressed as fixed leading zeros of the *B* polynomial.

`A0,B0,C0` — Initial value of polynomial coefficients

row vectors of real values | `[]`

Initial value of polynomial coefficients, specified as row vectors of real values with elements in order of ascending powers of q^{-1}.

- `A0` — Initial guess for the coefficients of the polynomial *A*(*q*), specified as a 1-by-(`na+1`) vector with 1 as the first element.
- `B0` — Initial guess for the coefficients of the polynomial *B*(*q*), specified as a 1-by-(`nb+nk`) vector with `nk` leading zeros.
- `C0` — Initial guess for the coefficients of the polynomial *C*(*q*), specified as a 1-by-(`nc+1`) vector with 1 as the first element.

  The coefficients in `C0` must define a stable discrete-time polynomial with roots within a unit disk. For example,

  ```
  C0 = [1 0.5 0.5];
  all(abs(roots(C0))<1)
  ```

  ```
  ans =

       1
  ```

Specifying as `[]`, uses the default value of `eps` for the polynomial coefficients.

Note: If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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 writable properties of `recursiveARMAX` System object during object creation. For example, `obj = recursiveARMAX([2 2 1 1],'EstimationMethod','Gradient')` creates a System object to estimate an ARMAX model using the `'Gradient'` recursive estimation algorithm.

Properties

`recursiveARMAX` System object properties consist of read-only and writable properties. The writable properties are tunable and nontunable properties. The nontunable properties cannot be changed when the object is locked, that is, after you use the `step` command.

Use `Name,Value` arguments to specify writable properties of `recursiveARMAX` objects during object creation. After object creation, use dot notation to modify the tunable properties.

```
obj = recursiveARMAX;
obj.ForgettingFactor = 0.99;
```

A

Estimated coefficients of polynomial $A(q)$, returned as a row vector of real values specified in order of ascending powers of q^{-1}.

`A` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

B

Estimated coefficients of polynomial $B(q)$, returned as a vector of real values specified in order of ascending powers of q^{-1}.

`B` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

`C`

Estimated coefficients of polynomial *C*(*q*), returned as a vector of real values specified in order of ascending powers of q^{-1}.

`C` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

`InitialA`

Initial values for the coefficients of polynomial *A*(*q*) of order `na`, specified as a row vector of length `na+1`, with 1 as the first element. Specify the coefficients in order of ascending powers of q^{-1}.

If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

`InitialA` is a tunable property. You can change it when the object is in a locked state.

Default: `[1 eps]`

`InitialB`

Initial values for the coefficients of polynomial *B*(*q*) of order `nb-1`, specified as a row vector of length `nb+nk`, with `nk` leading zeros. `nk` is the input-output delay. Specify the coefficients in order of ascending powers of q^{-1}.

If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

`InitialB` is a tunable property. You can change it when the object is in a locked state.

Default: `[0 eps]`

`InitialC`

Initial values for the coefficients of polynomial *C*(*q*) of order `nc`, specified as a row vector of length `nc+1`, with 1 as the first element. Specify the coefficients in order of ascending powers of q^{-1}.

The coefficients in `InitialC` must define a stable discrete-time polynomial with roots within a unit circle. For example,

```
InitialC = [1 0.5 0.5];
all(abs(roots(InitialC))<1)

ans =

     1
```

If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

`InitialC` is a tunable property. You can change it when the object is in a locked state.

Default: `[1 eps]`

`ParameterCovariance`

Estimated covariance of the parameters, returned as an N-by-N symmetric positive-definite matrix. N is the number of parameters to be estimated.

Applicable when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'`.

`ParameterCovariance` is computed assuming that the residuals (difference between estimated and measured outputs) are white noise, and the variance of these residuals is 1.

`ParameterCovariance` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

`InitialParameterCovariance`

Covariance of the initial parameter estimates, specified as one of the following:

- Real positive scalar, a — Covariance matrix is an N-by-N diagonal matrix, with a as the diagonal elements. N is the number of parameters to be estimated.
- Vector of real positive scalars, $[a_1,...,a_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[a_1,...,a_N]$ as the diagonal elements.
- N-by-N symmetric positive-definite matrix.

`InitialParameterCovariance` represents the uncertainty in the initial parameter estimates. For large values of `InitialParameterCovariance`, less importance is placed on the initial parameter values and more on the measured data during beginning of estimation using `step`.

Use only when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'`.

`InitialParameterCovariance` is a tunable property. You can change it when the object is in a locked state.

Default: `10000`

`EstimationMethod`

Recursive estimation algorithm used for online estimation of model parameters, specified as one of the following values:

- `'ForgettingFactor'` — Algorithm used for parameter estimation
- `'KalmanFilter'` — Algorithm used for parameter estimation
- `'NormalizedGradient'` — Algorithm used for parameter estimation
- `'Gradient'` — Unnormalized gradient algorithm used for parameter estimation

Forgetting factor and Kalman filter algorithms are more computationally intensive than gradient and unnormalized gradient methods. However, they have better convergence properties. For information about these algorithms, see "Recursive Algorithms for Online Parameter Estimation".

`EstimationMethod` is a nontunable property. You cannot change it during execution, that is after the object is locked using the `step` command. If you want to deploy code using MATLAB Coder, `EstimationMethod` can only be assigned once.

Default: `'ForgettingFactor'`

`ForgettingFactor`

Forgetting factor, λ, relevant for parameter estimation, specified as a scalar in the range (0,1].

Suppose that the system remains approximately constant over T_0 samples. You can choose λ such that:

$$T_0 = \frac{1}{1-\lambda}$$

- Setting $\lambda = 1$ corresponds to “no forgetting” and estimating constant coefficients.
- Setting $\lambda < 1$ implies that past measurements are less significant for parameter estimation and can be “forgotten”. Set $\lambda < 1$ to estimate time-varying coefficients.

Typical choices of λ are in the range `[0.98 0.995]`.

Use only when `EstimationMethod` is `'ForgettingFactor'`.

`ForgettingFactor` is a tunable property. You can change it when the object is in a locked state.

Default: `1`

EnableAdapation

Enable or disable parameter estimation, specified as one of the following:

- `true` or `1`— The `step` command estimates the parameter values for that time step and updates the parameter values.
- `false` or `0` — The `step` command does not update the parameters for that time step and instead outputs the last estimated value. You can use this option when your system enters a mode where the parameter values do not vary with time.

Note: If you set `EnableAdapation` to `false`, you must still execute the `step` command. Do not skip `step` to keep parameter values constant, because parameter estimation depends on current and past I/O measurements. `step` ensures past I/O data is stored, even when it does not update the parameters.

`EnableAdapation` is a tunable property. You can change it when the object is in a locked state.

Default: `true`

DataType

Floating point precision of parameters, specified as one of the following values:

- `'double'` — Double-precision floating point

- `'single'` — Single-precision floating point

Setting `DataType` to `'single'` saves memory, but leads to loss of precision. Specify `DataType` based on the precision required by the target processor where you will deploy generated code.

`DataType` is a nontunable property. It can only be set during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `'double'`

ProcessNoiseCovariance

Covariance matrix of parameter variations, specified as one of the following:

- Real nonnegative scalar, a — Covariance matrix is an N-by-N diagonal matrix, with a as the diagonal elements.
- Vector of real nonnegative scalars, $[a_1,...,a_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[a_1,...,a_N]$ as the diagonal elements.
- N-by-N symmetric positive semidefinite matrix.

N is the number of parameters to be estimated.

`ProcessNoiseCovariance` is applicable when `EstimationMethod` is `'KalmanFilter'`.

Kalman filter algorithm treats the parameters as states of a dynamic system and estimates these parameters using a Kalman filter. `ProcessNoiseCovariance` is the covariance of the process noise acting on these parameters. Zero values in the noise covariance matrix correspond to estimating constant coefficients. Values larger than 0 correspond to time-varying parameters. Use large values for rapidly changing parameters. However, the larger values result in noisier parameter estimates.

`ProcessNoiseCovariance` is a tunable property. You can change it when the object is in a locked state.

Default: `0.1`

AdaptationGain

Adaptation gain, γ, used in gradient recursive estimation algorithms, specified as a positive scalar.

`AdaptationGain` is applicable when `EstimationMethod` is `'Gradient'` or `'NormalizedGradient'`.

Specify a large value for `AdaptationGain` when your measurements have a high signal-to-noise ratio.

`AdaptationGain` is a tunable property. You can change it when the object is in a locked state.

Default: `1`

`NormalizationBias`

Bias in adaptation gain scaling used in the `'NormalizedGradient'` method, specified as a nonnegative scalar.

`NormalizationBias` is applicable when `EstimationMethod` is `'NormalizedGradient'`.

The normalized gradient algorithm divides the adaptation gain at each step by the square of the two-norm of the gradient vector. If the gradient is close to zero, this can cause jumps in the estimated parameters. `NormalizationBias` is the term introduced in the denominator to prevent these jumps. Increase `NormalizationBias` if you observe jumps in estimated parameters.

`NormalizationBias` is a tunable property. You can change it when the object is in a locked state.

Default: `eps`

Output Arguments

`obj` — System object for online parameter estimation of ARMAX model

`recursiveARMAX` System object

System object for online parameter estimation of ARMAX model, returned as a `recursiveARMAX` System object. This object is created using the specified model orders and properties. Use `step` command to estimate the coefficients of the ARMAX model polynomials. You can then access the estimated coefficients and parameter covariance using dot notation. For example, type `obj.A` to view the estimated *A* polynomial coefficients.

More About

ARMAX Model Structure

The ARMAX model structure is:

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

A more compact way to write the difference equation is:

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

where,

- $y(t)$ — Output at time t.
- n_a — Number of poles.
- n_b — Number of zeroes plus 1.
- n_c — Number of C coefficients.
- n_k — Number of input samples that occur before the input affects the output, also called the *dead time* in the system.
- $y(t-1)\ldots y(t-n_a)$ — Previous outputs on which the current output depends.
- $u(t-n_k)\ldots u(t-n_k-n_b+1)$ — Previous and delayed inputs on which the current output depends.
- $e(t-1)\ldots e(t-n_c)$ — White-noise disturbance value.

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

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

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

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

Tips

- Starting in R2016b, instead of using the `step` command to update model parameter estimates, you can call the System object with input arguments, as if it were a function. For example, `[A,B,C,EstimatedOutput] = step(obj,y,u)` and `[A,B,C,EstimatedOutput] = obj(y,u)` perform equivalent operations.
- “What Is Online Estimation?”
- “Recursive Algorithms for Online Parameter Estimation”

See Also

`armax` | `clone` | `isLocked` | `Recursive Polynomial Model Estimator` | `recursiveAR` | `recursiveARMA` | `recursiveARX` | `recursiveBJ` | `recursiveLS` | `recursiveOE` | `release` | `reset` | `step`

Introduced in R2015b

recursiveARX

Create System object for online parameter estimation of ARX model

Use `recursiveARX` command for parameter estimation with real-time data. If all data necessary for estimation is available at once, and you are estimating a time-invariant model, use the offline estimation command, `arx`.

Syntax

```
obj = recursiveARX
obj = recursiveARX(Orders)
obj = recursiveARX(Orders,A0,B0)
obj = recursiveARX(___,Name,Value)
```

Description

`obj = recursiveARX` creates a System object for online parameter estimation of a default ARX model structure. The default model structure has polynomials of order 1 and initial polynomial coefficient values `eps`.

After creating the object, use the `step` command to update model parameter estimates using recursive estimation algorithms and real-time data.

`obj = recursiveARX(Orders)` specifies the polynomial orders of the ARX model to be estimated.

`obj = recursiveARX(Orders,A0,B0)` specifies the polynomial orders and initial values of the polynomial coefficients.

`obj = recursiveARX(___,Name,Value)` specifies additional attributes of the ARX model structure and recursive estimation algorithm using one or more `Name,Value` pair arguments.

Object Description

`recursiveARX` creates a System object for online parameter estimation of single-input single-output (SISO) or multiple-input single-output (MISO) ARX models using a recursive estimation algorithm.

A System object is a specialized MATLAB object designed specifically for implementing and simulating dynamic systems with inputs that change over time. System objects use internal states to store past behavior, which is used in the next computational step.

After you create a System object, you use commands to process data or obtain information from or about the object. System objects use a minimum of two commands to process data — a constructor to create the object and the `step` command to update object parameters using real-time data. This separation of declaration from execution lets you create multiple, persistent, reusable objects, each with different settings.

You can use the following commands with the online estimation System objects in System Identification Toolbox:

Command	Description
`step`	Update model parameter estimates using recursive estimation algorithms and real-time data. `step` puts the object into a locked state. In a locked state, you cannot change any nontunable properties or input specifications, such as model order, data type, or estimation algorithm. During execution, you can only change tunable properties.
`release`	Unlock the System object. Use this command to enable setting of nontunable parameters.
`reset`	Reset the internal states of a locked System object to the initial values, and leave the object locked.
`clone`	Create another System object with the same object property values.

Command	Description
	Do not create additional objects using syntax `obj2 = obj`. Any changes made to the properties of the new object created this way (`obj2`) also change the properties of the original object (`obj`).
`isLocked`	Query locked status for input attributes and nontunable properties of the System object.

Use the `recursiveARX` command to create an online estimation System object. Then estimate the ARX model parameters (`A` and `B`) and output using the `step` command with incoming input and output data, `u` and `y`.

```
[A,B,EstimatedOutput] = step(obj,y,u)
```

For `recursiveARX` object properties, see “Properties” on page 1-1177.

Examples

Estimate a SISO ARX Model Online

Create a System object for online parameter estimation of a SISO ARX model.

```
obj = recursiveARX;
```

The ARX model has a default structure with polynomials of order 1 and initial polynomial coefficient values, `eps`.

Load the estimation data. In this example, use a static data set for illustration.

```
load iddata1 z1;
output = z1.y;
input = z1.u;
```

Estimate ARX model parameters online using `step`.

```
for i = 1:numel(input)
[A,B,EstimatedOutput] = step(obj,output(i),input(i));
end
```

View the current estimated values of polynomial `B` coefficients.

```
obj.B
```

```
ans =

         0    0.7974
```

View the current covariance estimate of the parameters.

```
obj.ParameterCovariance
```

```
ans =

    0.0002    0.0001
    0.0001    0.0034
```

View the current estimated output.

```
EstimatedOutput
```

```
EstimatedOutput =

   -4.7766
```

Create System Object for SISO ARX Model With Known Initial Parameters

Specify ARX model orders and delays.

```
na = 1;
nb = 2;
nk = 1;
```

Create a System object for online estimation of SISO ARX model with known initial polynomial coefficients.

```
A0 = [1 0.5];
B0 = [0 1 1];
obj = recursiveARX([na nb nk],A0,B0);
```

Specify the initial parameter covariance.

```
obj.InitialParameterCovariance = 0.1;
```

`InitialParameterCovariance` represents the uncertainty in your guess for the initial parameters. Typically, the default `InitialParameterCovariance` (10000) is too large relative to the parameter values. This results in initial guesses being given less importance during estimation. If you have confidence in the initial parameter guesses, specify a smaller initial parameter covariance.

Create System Object for MISO ARX Model With Known Initial Parameters

Specify orders and delays for ARX model with two inputs and one output.

```
na = 1;
nb = [2 1];
nk = [1 3];
```

`nb` and `nk` are specified as row vectors of length equal to number of inputs, *Nu*.

Specify initial polynomial coefficients.

```
A0 = [1 0.5];
B0 = [0 1 1 0; 0 0 0 0.8];
```

`B0` has *Nu* rows and `max(nb+nk)` columns. The *i*-th row corresponds to *i*-th input and is specified as having `nk(i)` zeros, followed by `nb(i)` inital values. Values after `nb(i)+nk(i)` are ignored.

Create a System object for online estimation of ARX model with known initial polynomial coefficients.

```
obj = recursiveARX([na nb nk],A0,B0);
```

Specify Estimation Method for Online Estimation of ARX Model

Create a System object that uses the normalized gradient algorithm for online parameter estimation of an ARX model.

```
obj = recursiveARX([1 2 1],'EstimationMethod','NormalizedGradient');
```

- “Perform Online Parameter Estimation at the Command Line”
- “Validate Online Parameter Estimation at the Command Line”

- “Online ARX Parameter Estimation for Tracking Time-Varying System Dynamics”

Input Arguments

`Orders` — Model orders and delays

1-by-3 vector of integers | 1-by-3 vector of vectors

Model orders and delays of an ARX model, specified as a 1-by-3 vector of integers or vectors, `[na nb nk]`.

- `na` — Order of the polynomial $A(q)$, specified as a nonnegative integer.
- `nb` — Order of the polynomial $B(q) + 1$, specified as 1–by-Nu vector of positive integers. Nu is the number of inputs.

  For MISO models, there are as many $B(q)$ polynomials as the number of inputs. `nb(i)` is the order of ith polynomial $B_i(q)$+1 for the ith input.
- `nk` — Input-output delay, specified as a 1–by-Nu vector of nonnegative integers. Nu is the number of inputs.

  For MISO models, there are as many $B(q)$ polynomials as the number of inputs. `nk(i)` is the input-output delay time corresponding to the ith input.

`A0`,`B0` — Initial value of polynomial coefficients

row vector and matrix of real values | `[]`

Initial value of coefficients of $A(q)$ and $B(q)$ polynomials, specified as row vector and matrix or real values, respectively. Specify the elements in order of ascending powers of q^{-1}.

- `A0` — Initial guess for the coefficients of the polynomial $A(q)$, specified as a 1-by-(`na+1`) row vector with 1 as the first element.
- `B0` — Initial guess for the coefficients of the polynomial $B(q)$, specified as Nu-by-`max(nb+nk)` matrix. Nu is the number of inputs.

  For MISO models, there are as many $B(q)$ polynomials as the number of inputs. The ith row of `B0` corresponds to the ith input and must contain `nk(i)` leading zeros, followed by `nb(i)` parameter guesses. Entries beyond `nk(i)+nb(i)` are ignored.

`na`, `nb`, and `nk` are the `Orders` of the model.

Specifying as `[]`, uses the default value of `eps` for the polynomial coefficients.

Note: If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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 writable properties of `recursiveARX` System object during object creation. For example, `obj = recursiveARX([2 2 1],'EstimationMethod','Gradient')` creates a System object to estimate an ARX model using the `'Gradient'` recursive estimation algorithm.

Properties

`recursiveARX` System object properties consist of read-only and writable properties. The writable properties are tunable and nontunable properties. The nontunable properties cannot be changed when the object is locked, that is, after you use the `step` command.

Use `Name,Value` arguments to specify writable properties of `recursiveARX` objects during object creation. After object creation, use dot notation to modify the tunable properties.

```
obj = recursiveARX;
obj.ForgettingFactor = 0.99;
```

A

Estimated coefficients of polynomial $A(q)$, returned as a row vector of real values specified in order of ascending powers of q^{-1}.

`A` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

`B`

Estimated coefficients of polynomial $B(q)$, returned as a Nu-by-`max(nb+nk)` matrix of real values. Nu is the number of inputs.

The ith row of `B` corresponds to the ith input and contains `nk(i)` leading zeros, followed by `nb(i)` estimated parameters, specified in order of ascending powers of q^{-1}. Ignore zero entries beyond `nk(i)+nb(i)`.

`B` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

`InitialA`

Initial values for the coefficients of polynomial $A(q)$ of order `na`, specified as a row vector of length `na+1`, with 1 as the first element. Specify the coefficients in order of ascending powers of q^{-1}.

If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

`InitialA` is a tunable property. You can change it when the object is in a locked state.

Default: `[1 eps]`

`InitialB`

Initial values for the coefficients of polynomial $B(q)$, specified as an Nu-by-`max(nb+nk)` matrix. Nu is the number of inputs.

For MISO models, there are as many $B(q)$ polynomials as the number of inputs. The ith row of `B0` corresponds to the ith input and must contain `nk(i)` zeros, followed by `nb(i)` parameter guesses. Entries beyond `nk(i)+nb(i)` are ignored.

If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

`InitialB` is a tunable property. You can change it when the object is in a locked state.

Default: `[0 eps]`

`ParameterCovariance`

Estimated covariance of the parameters, returned as an N-by-N symmetric positive-definite matrix. N is the number of parameters to be estimated.

Applicable when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'`.

`ParameterCovariance` is computed assuming that the residuals (difference between estimated and measured outputs) are white noise, and the variance of these residuals is 1.

`ParameterCovariance` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

`InitialParameterCovariance`

Covariance of the initial parameter estimates, specified as one of the following:

- Real positive scalar, a — Covariance matrix is an N-by-N diagonal matrix, with a as the diagonal elements. N is the number of parameters to be estimated.
- Vector of real positive scalars, $[a_1,...,a_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[a_1,...,a_N]$ as the diagonal elements.
- N-by-N symmetric positive-definite matrix.

`InitialParameterCovariance` represents the uncertainty in the initial parameter estimates. For large values of `InitialParameterCovariance`, less importance is placed on the initial parameter values and more on the measured data during beginning of estimation using `step`.

Use only when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'`.

`InitialParameterCovariance` is a tunable property. You can change it when the object is in a locked state.

Default: `10000`

`EstimationMethod`

Recursive estimation algorithm used for online estimation of model parameters, specified as one of the following values:

- `'ForgettingFactor'` — Algorithm used for parameter estimation
- `'KalmanFilter'` — Algorithm used for parameter estimation
- `'NormalizedGradient'` — Algorithm used for parameter estimation
- `'Gradient'` — Unnormalized gradient algorithm used for parameter estimation

Forgetting factor and Kalman filter algorithms are more computationally intensive than gradient and unnormalized gradient methods. However, they have better convergence properties. For information about these algorithms, see “Recursive Algorithms for Online Parameter Estimation”.

`EstimationMethod` is a nontunable property. You cannot change it during execution, that is after the object is locked using the `step` command. If you want to deploy code using MATLAB Coder, `EstimationMethod` can only be assigned once.

Default: `'ForgettingFactor'`

`ForgettingFactor`

Forgetting factor, λ, relevant for parameter estimation, specified as a scalar in the range (0,1].

Suppose that the system remains approximately constant over T_0 samples. You can choose λ such that:

$$T_0 = \frac{1}{1-\lambda}$$

- Setting $\lambda = 1$ corresponds to “no forgetting” and estimating constant coefficients.
- Setting $\lambda < 1$ implies that past measurements are less significant for parameter estimation and can be “forgotten”. Set $\lambda < 1$ to estimate time-varying coefficients.

Typical choices of λ are in the range `[0.98 0.995]`.

Use only when `EstimationMethod` is `'ForgettingFactor'`.

`ForgettingFactor` is a tunable property. You can change it when the object is in a locked state.

Default: 1

`EnableAdapation`

Enable or disable parameter estimation, specified as one of the following:

- `true` or `1`— The `step` command estimates the parameter values for that time step and updates the parameter values.
- `false` or `0` — The `step` command does not update the parameters for that time step and instead outputs the last estimated value. You can use this option when your system enters a mode where the parameter values do not vary with time.

Note: If you set `EnableAdapation` to `false`, you must still execute the `step` command. Do not skip `step` to keep parameter values constant, because parameter estimation depends on current and past I/O measurements. `step` ensures past I/O data is stored, even when it does not update the parameters.

`EnableAdapation` is a tunable property. You can change it when the object is in a locked state.

Default: `true`

`DataType`

Floating point precision of parameters, specified as one of the following values:

- `'double'` — Double-precision floating point
- `'single'` — Single-precision floating point

Setting `DataType` to `'single'` saves memory, but leads to loss of precision. Specify `DataType` based on the precision required by the target processor where you will deploy generated code.

`DataType` is a nontunable property. It can only be set during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `'double'`

`ProcessNoiseCovariance`

Covariance matrix of parameter variations, specified as one of the following:

- Real nonnegative scalar, a — Covariance matrix is an N-by-N diagonal matrix, with a as the diagonal elements.

- Vector of real nonnegative scalars, [a_1,...,a_N] — Covariance matrix is an N-by-N diagonal matrix, with [a_1,...,a_N] as the diagonal elements.
- N-by-N symmetric positive semidefinite matrix.

N is the number of parameters to be estimated.

`ProcessNoiseCovariance` is applicable when `EstimationMethod` is `'KalmanFilter'`.

Kalman filter algorithm treats the parameters as states of a dynamic system and estimates these parameters using a Kalman filter. `ProcessNoiseCovariance` is the covariance of the process noise acting on these parameters. Zero values in the noise covariance matrix correspond to estimating constant coefficients. Values larger than 0 correspond to time-varying parameters. Use large values for rapidly changing parameters. However, the larger values result in noisier parameter estimates.

`ProcessNoiseCovariance` is a tunable property. You can change it when the object is in a locked state.

Default: `0.1`

`AdaptationGain`

Adaptation gain, γ, used in gradient recursive estimation algorithms, specified as a positive scalar.

`AdaptationGain` is applicable when `EstimationMethod` is `'Gradient'` or `'NormalizedGradient'`.

Specify a large value for `AdaptationGain` when your measurements have a high signal-to-noise ratio.

`AdaptationGain` is a tunable property. You can change it when the object is in a locked state.

Default: `1`

`NormalizationBias`

Bias in adaptation gain scaling used in the `'NormalizedGradient'` method, specified as a nonnegative scalar.

`NormalizationBias` is applicable when `EstimationMethod` is `'NormalizedGradient'`.

The normalized gradient algorithm divides the adaptation gain at each step by the square of the two-norm of the gradient vector. If the gradient is close to zero, this can cause jumps in the estimated parameters. `NormalizationBias` is the term introduced in the denominator to prevent these jumps. Increase `NormalizationBias` if you observe jumps in estimated parameters.

`NormalizationBias` is a tunable property. You can change it when the object is in a locked state.

Default: `eps`

Output Arguments

`obj` — System object for online parameter estimation of ARX model

`recursiveARX` System object

System object for online parameter estimation of ARX model, returned as a `recursiveARX` System object. This object is created using the specified model orders and properties. Use `step` command to estimate the coefficients of the ARX model polynomials. You can then access the estimated coefficients and parameter covariance using dot notation. For example, type `obj.A` to view the estimated A polynomial coefficients.

More About

ARX Model Structure

The ARX model structure is :

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

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

- $y(t)$ — Output at time t.

- n_a — Number of poles.
- n_b — Number of zeroes plus 1.
- n_k — Number of input samples that occur before the input affects the output, also called the *dead time* in the system.
- $y(t-1)\ldots y(t-n_a)$ — Previous outputs on which the current output depends.
- $u(t-n_k)\ldots u(t-n_k-n_b+1)$ — Previous and delayed inputs on which the current output depends.
- $e(t)$ — White-noise disturbance value.

A more compact way to write the difference equation is

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

q is the delay operator. Specifically,

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

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

Tips

- Starting in R2016b, instead of using the `step` command to update model parameter estimates, you can call the System object with input arguments, as if it were a function. For example, `[A,B,EstimatedOutput] = step(obj,y,u)` and `[A,B,EstimatedOutput] = obj(y,u)` perform equivalent operations.
- “What Is Online Estimation?”
- “Recursive Algorithms for Online Parameter Estimation”

See Also

`arx` | `clone` | `isLocked` | `Recursive Polynomial Model Estimator` | `recursiveAR` | `recursiveARMA` | `recursiveARMAX` | `recursiveBJ` | `recursiveLS` | `recursiveOE` | `release` | `reset` | `step`

Introduced in R2015b

recursiveBJ

Create System object for online parameter estimation of Box-Jenkins polynomial model

Use `recursiveBJ` command for parameter estimation with real-time data. If all data necessary for estimation is available at once, and you are estimating a time-invariant model, use the offline estimation command, `bj`.

Syntax

```
obj = recursiveBJ
obj = recursiveBJ(Orders)
obj = recursiveBJ(Orders,B0,C0,D0,F0)
obj = recursiveBJ(___,Name,Value)
```

Description

`obj = recursiveBJ` creates a System object for online parameter estimation of a default single-input single-output (SISO) Box-Jenkins polynomial model structure. The default model structure has polynomials of order 1 and initial polynomial coefficient values `eps`.

After creating the object, use the `step` command to update model parameter estimates using recursive estimation algorithms and real-time data.

`obj = recursiveBJ(Orders)` specifies the polynomial orders of the Box-Jenkins model to be estimated.

`obj = recursiveBJ(Orders,B0,C0,D0,F0)` specifies the polynomial orders and initial values of the polynomial coefficients. Specify initial values to potentially avoid local minima during estimation. If the initial values are small compared to the default `InitialParameterCovariance` property value, and you have confidence in your initial values, also specify a smaller `InitialParameterCovariance`.

`obj = recursiveBJ(___,Name,Value)` specifies additional attributes of the Box-Jenkins model structure and recursive estimation algorithm using one or more `Name,Value` pair arguments.

Object Description

`recursiveBJ` creates a System object for online parameter estimation of SISO Box-Jenkins polynomial models using a recursive estimation algorithm.

A System object is a specialized MATLAB object designed specifically for implementing and simulating dynamic systems with inputs that change over time. System objects use internal states to store past behavior, which is used in the next computational step.

After you create a System object, you use commands to process data or obtain information from or about the object. System objects use a minimum of two commands to process data — a constructor to create the object and the `step` command to update object parameters using real-time data. This separation of declaration from execution lets you create multiple, persistent, reusable objects, each with different settings.

You can use the following commands with the online estimation System objects in System Identification Toolbox:

Command	**Description**
`step`	Update model parameter estimates using recursive estimation algorithms and real-time data. `step` puts the object into a locked state. In a locked state, you cannot change any nontunable properties or input specifications, such as model order, data type, or estimation algorithm. During execution, you can only change tunable properties.
`release`	Unlock the System object. Use this command to enable setting of nontunable parameters.
`reset`	Reset the internal states of a locked System object to the initial values, and leave the object locked.
`clone`	Create another System object with the same object property values.

Command	Description
	Do not create additional objects using syntax `obj2 = obj`. Any changes made to the properties of the new object created this way (`obj2`) also change the properties of the original object (`obj`).
`isLocked`	Query locked status for input attributes and nontunable properties of the System object.

Use the `recursiveBJ` command to create an online estimation System object. Then estimate the Box-Jenkins polynomial model parameters (`B`, `C`, `D`, and `F`) and output using the `step` command with incoming input and output data, `u` and `y`.

```
[B,C,D,F,EstimatedOutput] = step(obj,y,u)
```

For `recursiveBJ` object properties, see “Properties” on page 1-1193.

Examples

Estimate Box-Jenkins Polynomial Model Online

Create a System object for online parameter estimation of a Box-Jenkins polynomial model.

```
obj = recursiveBJ;
```

The Box-Jenkins model has a default structure with polynomials of order 1 and initial polynomial coefficient values, `eps`.

Load the estimation data. In this example, use a static data set for illustration.

```
load iddata1 z1;
output = z1.y;
input = z1.u;
```

Estimate Box-Jenkins model parameters online using `step`.

```
for i = 1:numel(input)
```

```
[B,C,D,F,EstimatedOutput] = step(obj,output(i),input(i));
end
```

View the current estimated values of polynomial `D` coefficients.

```
obj.D
```

```

ans =

    1.0000   -0.6876

```

View the current covariance estimate of the parameters.

```
obj.ParameterCovariance
```

```

ans =

    0.0020   -0.0004   -0.0001    0.0002
   -0.0004    0.0007    0.0006   -0.0001
   -0.0001    0.0006    0.0007   -0.0000
    0.0002   -0.0001   -0.0000    0.0001

```

View the current estimated output.

```
EstimatedOutput
```

```

EstimatedOutput =

   -4.1905

```

Create System Object for Box-Jenkins Model With Known Orders and Delays

Specify Box-Jenkins polynomial model orders and delays.

```
nb = 1;
nc = 1;
nd = 2;
nf = 1;
```

```
nk = 1;
```

Create a System object for online estimation of Box-Jenkins model with the specified orders and delays.

```
obj = recursiveBJ([nb nc nd nf nk]);
```

Create System Object for Box-Jenkins Model With Known Initial Parameters

Specify Box-Jenkins polynomial model orders and delays.

```
nb = 1;
nc = 1;
nd = 1;
nf = 2;
nk = 1;
```

Create a System object for online estimation of Box-Jenkins model with known initial polynomial coefficients.

```
B0 = [0 1];
C0 = [1 0.5];
D0 = [1 0.9];
F0 = [1 0.7 0.8];
obj = recursiveBJ([nb nc nd nf nk],B0,C0,D0,F0);
```

Specify the initial parameter covariance.

```
obj.InitialParameterCovariance = 0.1;
```

`InitialParameterCovariance` represents the uncertainty in your guess for the initial parameters. Typically, the default `InitialParameterCovariance` (10000) is too large relative to the parameter values. This results in initial guesses being given less importance during estimation. If you have confidence in the initial parameter guesses, specify a smaller initial parameter covariance.

Specify Estimation Method for Online Estimation of Box-Jenkins Model

Create a System object that uses the normalized gradient algorithm for online parameter estimation of a Box-Jenkins model.

```
obj = recursiveBJ([1 1 1 2 1],'EstimationMethod','NormalizedGradient');
```

- “Perform Online Parameter Estimation at the Command Line”

- “Validate Online Parameter Estimation at the Command Line”

Input Arguments

`Orders` — Model orders and delays

1-by-5 vector of integers

Model orders and delays of a Box-Jenkins polynomial model, specified as a 1-by-5 vector of integers, `[nb nc nd nf nk]`.

- `nb` — Order of the polynomial $B(q) + 1$, specified as a positive integer.
- `nc` — Order of the polynomial $C(q)$, specified as a nonnegative integer.
- `nd` — Order of the polynomial $D(q)$, specified as a nonnegative integer.
- `nf` — Order of the polynomial $F(q)$, specified as a nonnegative integer.
- `nk` — Input-output delay, specified as a positive integer. `nk` is number of input samples that occur before the input affects the output. `nk` is expressed as fixed leading zeros of the *B* polynomial.

`B0,C0,D0,F0` — Initial value of polynomial coefficients

row vectors of real values | `[]`

Initial value of polynomial coefficients, specified as row vectors of real values with elements in order of ascending powers of q^{-1}.

- `B0` — Initial guess for the coefficients of the polynomial $B(q)$, specified as a 1-by-(`nb+nk`) vector with `nk` leading zeros.
- `C0` — Initial guess for the coefficients of the polynomial $C(q)$, specified as a 1-by-(`nc+1`) vector with 1 as the first element.

  The coefficients in `C0` must define a stable discrete-time polynomial with roots within a unit disk. For example,

  ```
  C0 = [1 0.5 0.5];
  all(abs(roots(C0))<1)

  ans =

       1
  ```

- `D0` — Initial guess for the coefficients of the polynomial *D*(*q*), specified as a 1-by-(`nd+1`) vector with 1 as the first element.

  The coefficients in `D0` must define a stable discrete-time polynomial with roots within a unit disk. For example,

  ```
  D0 = [1 0.9 0.8];
  all(abs(roots(D0))<1)

  ans =

       1
  ```
- `F0` — Initial guess for the coefficients of the polynomial *F*(*q*), specified as a 1-by-(`nf+1`) vector with 1 as the first element.

  The coefficients in `F0` must define a stable discrete-time polynomial with roots within a unit disk. For example,

  ```
  F0 = [1 0.5 0.5];
  all(abs(roots(F0))<1)

  ans =

       1
  ```

Specifying as `[]`, uses the default value of `eps` for the polynomial coefficients.

Note: If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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 writable properties of `recursiveBJ` System object during object creation. For example, `obj = recursiveBJ([1 1 1 2`

`1],'EstimationMethod','Gradient')` creates a System object to estimate a Box-Jenkins polynomial model using the `'Gradient'` recursive estimation algorithm.

Properties

`recursiveBJ` System object properties consist of read-only and writable properties. The writable properties are tunable and nontunable properties. The nontunable properties cannot be changed when the object is locked, that is, after you use the `step` command.

Use `Name,Value` arguments to specify writable properties of `recursiveBJ` objects during object creation. After object creation, use dot notation to modify the tunable properties.

```
obj = recursiveBJ;
obj.ForgettingFactor = 0.99;
```

`B`

Estimated coefficients of polynomial $B(q)$, returned as a vector of real values specified in order of ascending powers of q^{-1}.

`B` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

`C`

Estimated coefficients of polynomial $C(q)$, returned as a vector of real values specified in order of ascending powers of q^{-1}.

`C` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

`D`

Estimated coefficients of polynomial $D(q)$, returned as a vector of real values specified in order of ascending powers of q^{-1}.

`D` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

F

Estimated coefficients of polynomial $F(q)$, returned as a vector of real values specified in order of ascending powers of q^{-1}.

`F` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

InitialB

Initial values for the coefficients of polynomial $B(q)$ of order `nb-1`, specified as a row vector of length `nb+nk`, with `nk` leading zeros. `nk` is the input-output delay. Specify the coefficients in order of ascending powers of q^{-1}.

If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

`InitialB` is a tunable property. You can change it when the object is in a locked state.

Default: `[0 eps]`

InitialC

Initial values for the coefficients of polynomial $C(q)$ of order `nc`, specified as a row vector of length `nc+1`, with 1 as the first element. Specify the coefficients in order of ascending powers of q^{-1}.

The coefficients in `InitialC` must define a stable discrete-time polynomial with roots within a unit circle. For example,

```
InitialC = [1 0.5 0.5];
all(abs(roots(InitialC))<1)
```

```
ans =

     1
```

If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

`InitialC` is a tunable property. You can change it when the object is in a locked state.

Default: `[1 eps]`

`InitialD`

Initial values for the coefficients of polynomial $D(q)$ of order `nd`, specified as a row vector of length `nd+1`, with 1 as the first element. Specify the coefficients in order of ascending powers of q^{-1}.

The coefficients in `InitialD` must define a stable discrete-time polynomial with roots within a unit circle. For example,

```
InitialD = [1 0.9 0.8];
all(abs(roots(InitialD))<1)
```

```
ans =

     1
```

If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

`InitialD` is a tunable property. You can change it when the object is in a locked state.

Default: `[1 eps]`

`InitialF`

Initial values for the coefficients of polynomial $F(q)$ of order `nf`, specified as a row vector of length `nf+1`, with 1 as the first element. Specify the coefficients in order of ascending powers of q^{-1}.

The coefficients in `InitialF` must define a stable discrete-time polynomial with roots within a unit circle. For example,

```
InitialF = [1 0.9 0.8];
all(abs(roots(InitialF))<1)
```

```
ans =

     1
```

If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

`InitialF` is a tunable property. You can change it when the object is in a locked state.

Default: `[1 eps]`

`ParameterCovariance`

Estimated covariance of the parameters, returned as an N-by-N symmetric positive-definite matrix. N is the number of parameters to be estimated.

Applicable when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'`.

`ParameterCovariance` is computed assuming that the residuals (difference between estimated and measured outputs) are white noise, and the variance of these residuals is 1.

`ParameterCovariance` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

`InitialParameterCovariance`

Covariance of the initial parameter estimates, specified as one of the following:

- Real positive scalar, a — Covariance matrix is an N-by-N diagonal matrix, with a as the diagonal elements. N is the number of parameters to be estimated.
- Vector of real positive scalars, $[a_1,...,a_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[a_1,...,a_N]$ as the diagonal elements.
- N-by-N symmetric positive-definite matrix.

`InitialParameterCovariance` represents the uncertainty in the initial parameter estimates. For large values of `InitialParameterCovariance`, less importance is placed on the initial parameter values and more on the measured data during beginning of estimation using `step`.

Use only when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'`.

`InitialParameterCovariance` is a tunable property. You can change it when the object is in a locked state.

Default: `10000`

`EstimationMethod`

Recursive estimation algorithm used for online estimation of model parameters, specified as one of the following values:

- `'ForgettingFactor'` — Algorithm used for parameter estimation
- `'KalmanFilter'` — Algorithm used for parameter estimation
- `'NormalizedGradient'` — Algorithm used for parameter estimation
- `'Gradient'` — Unnormalized gradient algorithm used for parameter estimation

Forgetting factor and Kalman filter algorithms are more computationally intensive than gradient and unnormalized gradient methods. However, they have better convergence properties. For information about these algorithms, see “Recursive Algorithms for Online Parameter Estimation”.

`EstimationMethod` is a nontunable property. You cannot change it during execution, that is after the object is locked using the `step` command. If you want to deploy code using MATLAB Coder, `EstimationMethod` can only be assigned once.

Default: `'ForgettingFactor'`

`ForgettingFactor`

Forgetting factor, λ, relevant for parameter estimation, specified as a scalar in the range (0,1].

Suppose that the system remains approximately constant over T_0 samples. You can choose λ such that:

$$T_0 = \frac{1}{1-\lambda}$$

- Setting $\lambda = 1$ corresponds to “no forgetting” and estimating constant coefficients.
- Setting $\lambda < 1$ implies that past measurements are less significant for parameter estimation and can be “forgotten”. Set $\lambda < 1$ to estimate time-varying coefficients.

Typical choices of λ are in the range `[0.98 0.995]`.

Use only when `EstimationMethod` is `'ForgettingFactor'`.

`ForgettingFactor` is a tunable property. You can change it when the object is in a locked state.

Default: `1`

`EnableAdapation`

Enable or disable parameter estimation, specified as one of the following:

- `true` or `1`— The `step` command estimates the parameter values for that time step and updates the parameter values.
- `false` or `0` — The `step` command does not update the parameters for that time step and instead outputs the last estimated value. You can use this option when your system enters a mode where the parameter values do not vary with time.

Note: If you set `EnableAdapation` to `false`, you must still execute the `step` command. Do not skip `step` to keep parameter values constant, because parameter estimation depends on current and past I/O measurements. `step` ensures past I/O data is stored, even when it does not update the parameters.

`EnableAdapation` is a tunable property. You can change it when the object is in a locked state.

Default: `true`

`DataType`

Floating point precision of parameters, specified as one of the following values:

- `'double'` — Double-precision floating point
- `'single'` — Single-precision floating point

Setting `DataType` to `'single'` saves memory, but leads to loss of precision. Specify `DataType` based on the precision required by the target processor where you will deploy generated code.

`DataType` is a nontunable property. It can only be set during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `'double'`

`ProcessNoiseCovariance`

Covariance matrix of parameter variations, specified as one of the following:

- Real nonnegative scalar, a — Covariance matrix is an N-by-N diagonal matrix, with a as the diagonal elements.
- Vector of real nonnegative scalars, $[a_1,...,a_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[a_1,...,a_N]$ as the diagonal elements.
- N-by-N symmetric positive semidefinite matrix.

N is the number of parameters to be estimated.

`ProcessNoiseCovariance` is applicable when `EstimationMethod` is `'KalmanFilter'`.

Kalman filter algorithm treats the parameters as states of a dynamic system and estimates these parameters using a Kalman filter. `ProcessNoiseCovariance` is the covariance of the process noise acting on these parameters. Zero values in the noise covariance matrix correspond to estimating constant coefficients. Values larger than 0 correspond to time-varying parameters. Use large values for rapidly changing parameters. However, the larger values result in noisier parameter estimates.

`ProcessNoiseCovariance` is a tunable property. You can change it when the object is in a locked state.

Default: `0.1`

`AdaptationGain`

Adaptation gain, γ, used in gradient recursive estimation algorithms, specified as a positive scalar.

`AdaptationGain` is applicable when `EstimationMethod` is `'Gradient'` or `'NormalizedGradient'`.

Specify a large value for `AdaptationGain` when your measurements have a high signal-to-noise ratio.

`AdaptationGain` is a tunable property. You can change it when the object is in a locked state.

Default: `1`

`NormalizationBias`

Bias in adaptation gain scaling used in the `'NormalizedGradient'` method, specified as a nonnegative scalar.

`NormalizationBias` is applicable when `EstimationMethod` is `'NormalizedGradient'`.

The normalized gradient algorithm divides the adaptation gain at each step by the square of the two-norm of the gradient vector. If the gradient is close to zero, this can cause jumps in the estimated parameters. `NormalizationBias` is the term introduced in the denominator to prevent these jumps. Increase `NormalizationBias` if you observe jumps in estimated parameters.

`NormalizationBias` is a tunable property. You can change it when the object is in a locked state.

Default: `eps`

Output Arguments

`obj` — System object for online parameter estimation of Box-Jenkins polynomial model

`recursiveBJ` System object

System object for online parameter estimation of Box-Jenkins polynomial model, returned as a `recursiveBJ` System object. This object is created using the specified model orders and properties. Use `step` command to estimate the coefficients of the Box-Jenkins model polynomials. You can then access the estimated coefficients and parameter covariance using dot notation. For example, type `obj.F` to view the estimated F polynomial coefficients.

More About

Box-Jenkins Polynomial Model Structure

The general Box-Jenkins model structure is:

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

where nu is the number of input channels.

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

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

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

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

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

Tips

- Starting in R2016b, instead of using the `step` command to update model parameter estimates, you can call the System object with input arguments, as if it were a function. For example, `[B,C,D,F,EstimatedOutput] = step(obj,y,u)` and `[B,C,D,F,EstimatedOutput] = obj(y,u)` perform equivalent operations.
- “What Is Online Estimation?”
- “Recursive Algorithms for Online Parameter Estimation”

See Also

`bj` | `clone` | `isLocked` | `Recursive Polynomial Model Estimator` | `recursiveAR` | `recursiveARMA` | `recursiveARMAX` | `recursiveARX` | `recursiveLS` | `recursiveOE` | `release` | `reset` | `step`

Introduced in R2015b

recursiveLS

Create System object for online parameter estimation using recursive least squares algorithm

Use `recursiveLS` command for parameter estimation with real-time data. If all data necessary for estimation is available at once, and you are estimating a time-invariant model, use `mldivide, \`.

Syntax

```
obj = recursiveLS
obj = recursiveLS(Np)
obj = recursiveLS(Np,theta0)
obj = recursiveLS(___,Name,Value)
```

Description

`obj = recursiveLS` creates a System object for online parameter estimation of a default single output system that is linear in estimated parameters. Such a system can be represented as:

$y(t) = H(t)\theta(t)+e(t)$.

Here, y is the output, θ are the parameters, H are the regressors, and e is the white-noise disturbance. The default system has one parameter with initial parameter value `1`.

After creating the object, use the `step` command to update model parameter estimates using recursive least squares algorithms and real-time data.

`obj = recursiveLS(Np)` also specifies the number of parameters to be estimated.

`obj = recursiveLS(Np,theta0)` also specifies the number of parameters and initial values of the parameters.

`obj = recursiveLS(___,Name,Value)` specifies additional attributes of the system and recursive estimation algorithm using one or more `Name,Value` pair arguments.

Object Description

`recursiveLS` creates a System object for online parameter estimation of a single output system that is linear in its parameters.

A System object is a specialized MATLAB object designed specifically for implementing and simulating dynamic systems with inputs that change over time. System objects use internal states to store past behavior, which is used in the next computational step.

After you create a System object, you use commands to process data or obtain information from or about the object. System objects use a minimum of two commands to process data — a constructor to create the object and the `step` command to update object parameters using real-time data. This separation of declaration from execution lets you create multiple, persistent, reusable objects, each with different settings.

You can use the following commands with the online estimation System objects in System Identification Toolbox:

Command	Description
`step`	Update model parameter estimates using recursive estimation algorithms and real-time data. `step` puts the object into a locked state. In a locked state, you cannot change any nontunable properties or input specifications, such as model order, data type, or estimation algorithm. During execution, you can only change tunable properties.
`release`	Unlock the System object. Use this command to enable setting of nontunable parameters.
`reset`	Reset the internal states of a locked System object to the initial values, and leave the object locked.
`clone`	Create another System object with the same object property values.

Command	Description
	Do not create additional objects using syntax `obj2 = obj`. Any changes made to the properties of the new object created this way (`obj2`) also change the properties of the original object (`obj`).
`isLocked`	Query locked status for input attributes and nontunable properties of the System object.

Use the `recursiveLS` command to create an online estimation System object. Then estimate the system parameters (`theta`) and output using the `step` command with regressors and incoming output data, `H` and `y`.

```
[theta,EstimatedOutput] = step(obj,y,H)
```

For `recursiveLS` object properties, see “Properties” on page 1-1208.

Examples

Create System Object for Online Estimation Using Recursive Least Squares Algorithm

```
obj = recursiveLS
```

```
obj = 

  recursiveLS with properties:

            NumberOfParameters: 1
                    Parameters: []
             InitialParameters: 1
           ParameterCovariance: []
    InitialParameterCovariance: 10000
              EstimationMethod: 'ForgettingFactor'
              ForgettingFactor: 1
              EnableAdaptation: true
                      DataType: 'double'
```

Estimate Parameters of System Using Recursive Least Squares Algorithm

The system has two parameters and is represented as:

$$y(t) = a_1 u(t) + a_2 u(t-1).$$

Here,

- u and y are the real-time input and output data, respectively.
- $u(t)$ and $u(t-1)$ are the regressors, `H`, of the system.
- a_1 and a_2 are the parameters, `theta`, of the system.

Create a System object for online estimation using recursive least squares algorithm.

```
obj = recursiveLS(2);
```

Load the estimation data. In this example, we are using a static data set for illustration.

```
load iddata3
input = z3.u;
output = z3.y;
```

Create a variable to store `u(t-1)`. This variable is updated at each time step.

```
oldInput = 0;
```

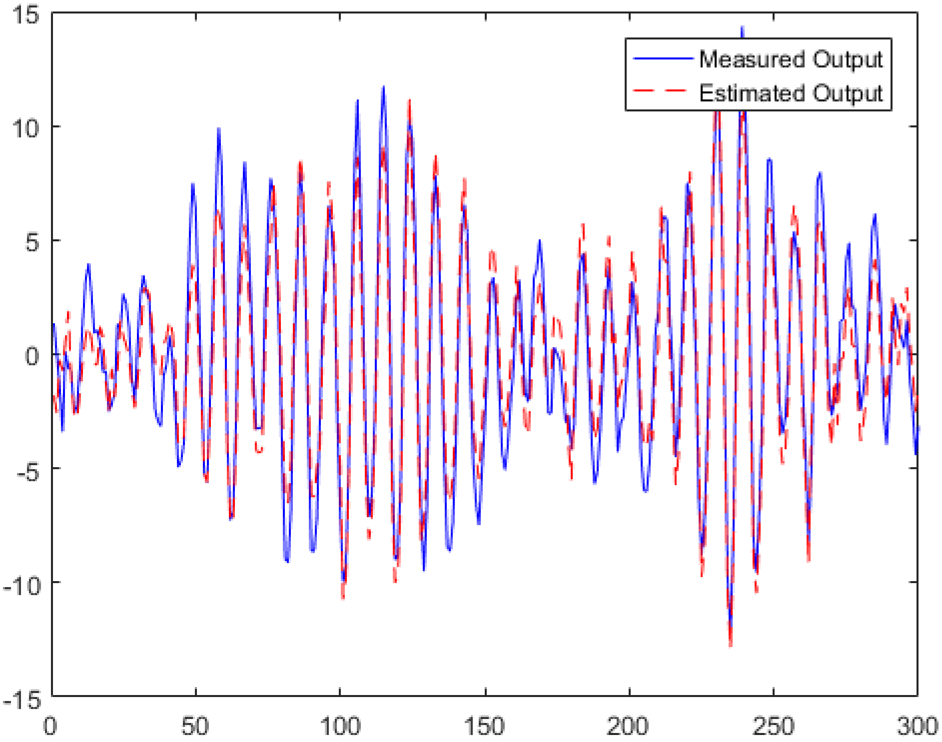
Estimate the parameters and output using `step` and input-output data.

```
for i = 1:numel(input)
    H = [input(i) oldInput];
    [theta, EstimatedOutput] = step(obj,output(i),H);
    estimatedOut(i)= EstimatedOutput;
    oldInput = input(i);
end
```

Plot the measured and estimated output data.

```
numSample = 1:numel(input);
```

```
plot(numSample,output,'b',numSample,estimatedOut,'r--');
legend('Measured Output','Estimated Output');
```



Specify Initial Parameters for Online Estimation Using Recursive Least Squares Algorithm

Create System object for online parameter estimation using recursive least squares algorithm of a system with two parameters and known initial parameter values.

```
obj = recursiveLS(2,[0.8 1],'InitialParameterCovariance',0.1);
```

`InitialParameterCovariance` represents the uncertainty in your guess for the initial parameters. Typically, the default `InitialParameterCovariance` (10000) is too large relative to the parameter values. This results in initial guesses being given less importance during estimation. If you have confidence in the initial parameter guesses, specify a smaller initial parameter covariance.

- “Perform Online Parameter Estimation at the Command Line”
- “Validate Online Parameter Estimation at the Command Line”
- “Line Fitting with Online Recursive Least Squares Estimation”

Input Arguments

`Np` — Number of parameters

positive integer

Number of parameters in the system, specified as a positive integer. All parameters have the initial value `1`.

`theta0` — Initial value of parameters

scalar | vector of real values

Initial value of parameters, specified as one of the following:

- Scalar — All the parameters have the same initial value.
- Vector of real values of length `Np`— The *i*th parameter has initial value `theta0(i)`.

Note: If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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 writable properties of `recursiveLS` System object during object creation. For example, `obj = recursiveLS(2,'EstimationMethod','Gradient')` creates a System object to estimate the system parameters using the `'Gradient'` recursive estimation algorithm.

Properties

`recursiveLS` System object properties consist of read-only and writable properties. The writable properties are tunable and nontunable properties. The nontunable properties cannot be changed when the object is locked, that is, after you use the `step` command.

Use `Name,Value` arguments to specify writable properties of `recursiveLS` objects during object creation. After object creation, use dot notation to modify the tunable properties.

```
obj = recursiveLS;
obj.ForgettingFactor = 0.99;
```

`NumberOfParameters`

Number of parameters to be estimated, returned as a positive integer.

`NumberOfParameters` is a read-only property. If `Np` is specified during object construction, `NumberOfParameters` takes the value assigned to `Np`.

Default: `1`

`Parameters`

Estimated parameters, returned as a column vector of real values.

`Parameters` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

`InitialParameters`

Initial value of parameters, specified as one of the following:

- Scalar — All the parameters have the same initial value.
- Vector of real values of length `Np`— The *i*th parameter has initial value `InitialParameters(i)`.

If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

`InitialParameters` is a tunable property. You can change `InitialParameters` when the object is in a locked state.

Default: `1`

`ParameterCovariance`

Estimated covariance of the parameters, returned as an N-by-N symmetric positive-definite matrix. N is the number of parameters to be estimated.

Applicable when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'`.

`ParameterCovariance` is computed assuming that the residuals (difference between estimated and measured outputs) are white noise, and the variance of these residuals is 1.

`ParameterCovariance` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

`InitialParameterCovariance`

Covariance of the initial parameter estimates, specified as one of the following:

- Real positive scalar, a — Covariance matrix is an N-by-N diagonal matrix, with a as the diagonal elements. N is the number of parameters to be estimated.
- Vector of real positive scalars, $[a_1,...,a_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[a_1,...,a_N]$ as the diagonal elements.
- N-by-N symmetric positive-definite matrix.

`InitialParameterCovariance` represents the uncertainty in the initial parameter estimates. For large values of `InitialParameterCovariance`, less importance is placed on the initial parameter values and more on the measured data during beginning of estimation using `step`.

Use only when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'`.

`InitialParameterCovariance` is a tunable property. You can change it when the object is in a locked state.

Default: `10000`

`EstimationMethod`

Recursive least squares estimation algorithm used for online estimation of model parameters, specified as one of the following values:

- `'ForgettingFactor'` — Algorithm used for parameter estimation
- `'KalmanFilter'` — Algorithm used for parameter estimation
- `'NormalizedGradient'` — Algorithm used for parameter estimation
- `'Gradient'` — Unnormalized gradient algorithm used for parameter estimation

Forgetting factor and Kalman filter algorithms are more computationally intensive than gradient and unnormalized gradient methods. However, they have better convergence properties. For information about these algorithms, see “Recursive Algorithms for Online Parameter Estimation”.

`EstimationMethod` is a nontunable property. You cannot change it during execution, that is after the object is locked using the `step` command. If you want to deploy code using MATLAB Coder, `EstimationMethod` can only be assigned once.

ForgettingFactor

Forgetting factor, λ, relevant for parameter estimation, specified as a scalar in the range (0,1].

Suppose that the system remains approximately constant over T_0 samples. You can choose λ such that:

$$T_0 = \frac{1}{1-\lambda}$$

- Setting $\lambda = 1$ corresponds to “no forgetting” and estimating constant coefficients.
- Setting $\lambda < 1$ implies that past measurements are less significant for parameter estimation and can be “forgotten”. Set $\lambda < 1$ to estimate time-varying coefficients.

Typical choices of λ are in the range `[0.98 0.995]`.

Use only when `EstimationMethod` is `'ForgettingFactor'`.

`ForgettingFactor` is a tunable property. You can change it when the object is in a locked state.

Default: 1

EnableAdapation

Enable or disable parameter estimation, specified as one of the following:

- `true` or `1`— The `step` command estimates the parameter values for that time step and updates the parameter values.
- `false` or `0` — The `step` command does not update the parameters for that time step and instead outputs the last estimated value. You can use this option when your system enters a mode where the parameter values do not vary with time.

Note: If you set `EnableAdapation` to `false`, you must still execute the `step` command. Do not skip `step` to keep parameter values constant, because parameter estimation depends on current and past I/O measurements. `step` ensures past I/O data is stored, even when it does not update the parameters.

`EnableAdapation` is a tunable property. You can change it when the object is in a locked state.

Default: `true`

DataType

Floating point precision of parameters, specified as one of the following values:

- `'double'` — Double-precision floating point
- `'single'` — Single-precision floating point

Setting `DataType` to `'single'` saves memory, but leads to loss of precision. Specify `DataType` based on the precision required by the target processor where you will deploy generated code.

`DataType` is a nontunable property. It can only be set during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `'double'`

ProcessNoiseCovariance

Covariance matrix of parameter variations, specified as one of the following:

- Real nonnegative scalar, a — Covariance matrix is an N-by-N diagonal matrix, with a as the diagonal elements.
- Vector of real nonnegative scalars, $[a_1,...,a_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[a_1,...,a_N]$ as the diagonal elements.

- N-by-N symmetric positive semidefinite matrix.

N is the number of parameters to be estimated.

`ProcessNoiseCovariance` is applicable when `EstimationMethod` is `'KalmanFilter'`.

Kalman filter algorithm treats the parameters as states of a dynamic system and estimates these parameters using a Kalman filter. `ProcessNoiseCovariance` is the covariance of the process noise acting on these parameters. Zero values in the noise covariance matrix correspond to estimating constant coefficients. Values larger than 0 correspond to time-varying parameters. Use large values for rapidly changing parameters. However, the larger values result in noisier parameter estimates.

`ProcessNoiseCovariance` is a tunable property. You can change it when the object is in a locked state.

Default: `0.1`

AdaptationGain

Adaptation gain, γ, used in gradient recursive estimation algorithms, specified as a positive scalar.

`AdaptationGain` is applicable when `EstimationMethod` is `'Gradient'` or `'NormalizedGradient'`.

Specify a large value for `AdaptationGain` when your measurements have a high signal-to-noise ratio.

`AdaptationGain` is a tunable property. You can change it when the object is in a locked state.

Default: `1`

NormalizationBias

Bias in adaptation gain scaling used in the `'NormalizedGradient'` method, specified as a nonnegative scalar.

`NormalizationBias` is applicable when `EstimationMethod` is `'NormalizedGradient'`.

The normalized gradient algorithm divides the adaptation gain at each step by the square of the two-norm of the gradient vector. If the gradient is close to zero, this can cause jumps in the estimated parameters. `NormalizationBias` is the term introduced in the denominator to prevent these jumps. Increase `NormalizationBias` if you observe jumps in estimated parameters.

`NormalizationBias` is a tunable property. You can change it when the object is in a locked state.

Default: `eps`

Output Arguments

`obj` — System object for online parameter estimation

`recursiveLS` System object

System object for online parameter estimation, returned as a `recursiveLS` System object. Use `step` command to estimate the parameters of the system. You can then access the estimated parameters and parameter covariance using dot notation. For example, type `obj.Parameters` to view the estimated parameters.

More About

Tips

- Starting in R2016b, instead of using the `step` command to update model parameter estimates, you can call the System object with input arguments, as if it were a function. For example, `[theta,EstimatedOutput] = step(obj,y,H)` and `[theta,EstimatedOutput] = obj(y,H)` perform equivalent operations.

- “What Is Online Estimation?”
- “Recursive Algorithms for Online Parameter Estimation”

See Also

`clone` | `isLocked` | `mldivide` | `Recursive Least Squares Estimator` | `recursiveAR` | `recursiveARMA` | `recursiveARMAX` | `recursiveARX` | `recursiveBJ` | `recursiveOE` | `release` | `reset` | `step`

Introduced in R2015b

recursiveOE

Create System object for online parameter estimation of Output-Error polynomial model

Use `recursiveOE` command for parameter estimation with real-time data. If all data necessary for estimation is available at once, and you are estimating a time-invariant model, use the offline estimation command, `oe`.

Syntax

```
obj = recursiveOE
obj = recursiveOE(Orders)
obj = recursiveOE(Orders,B0,F0)
obj = recursiveOE( ___ ,Name,Value)
```

Description

`obj = recursiveOE` creates a System object for online parameter estimation of a default single-input-single output (SISO) Output-Error model structure. The default model structure has polynomials of order 1 and initial polynomial coefficient values `eps`.

After creating the object, use the `step` command to update model parameter estimates using recursive estimation algorithms and real-time data.

`obj = recursiveOE(Orders)` specifies the polynomial orders of the Output-Error model to be estimated.

`obj = recursiveOE(Orders,B0,F0)` specifies the polynomial orders and initial values of the polynomial coefficients. Specify initial values to potentially avoid local minima during estimation. If the initial values are small compared to the default `InitialParameterCovariance` property value, and you have confidence in your initial values, also specify a smaller `InitialParameterCovariance`.

`obj = recursiveOE( ___ ,Name,Value)` specifies additional attributes of the Output-Error model structure and recursive estimation algorithm using one or more `Name,Value` pair arguments.

Object Description

`recursiveOE` creates a System object for online parameter estimation of SISO Output-Error polynomial models using a recursive estimation algorithm.

A System object is a specialized MATLAB object designed specifically for implementing and simulating dynamic systems with inputs that change over time. System objects use internal states to store past behavior, which is used in the next computational step.

After you create a System object, you use commands to process data or obtain information from or about the object. System objects use a minimum of two commands to process data — a constructor to create the object and the `step` command to update object parameters using real-time data. This separation of declaration from execution lets you create multiple, persistent, reusable objects, each with different settings.

You can use the following commands with the online estimation System objects in System Identification Toolbox:

Command	Description
`step`	Update model parameter estimates using recursive estimation algorithms and real-time data. `step` puts the object into a locked state. In a locked state, you cannot change any nontunable properties or input specifications, such as model order, data type, or estimation algorithm. During execution, you can only change tunable properties.
`release`	Unlock the System object. Use this command to enable setting of nontunable parameters.
`reset`	Reset the internal states of a locked System object to the initial values, and leave the object locked.
`clone`	Create another System object with the same object property values.

Command	Description
	Do not create additional objects using syntax `obj2 = obj`. Any changes made to the properties of the new object created this way (`obj2`) also change the properties of the original object (`obj`).
`isLocked`	Query locked status for input attributes and nontunable properties of the System object.

Use the `recursiveOE` command to create an online estimation System object. Then estimate the Output-Error polynomial model parameters (`B` and `F`) and output using the `step` command with incoming input and output data, `u` and `y`.

```
[B,F,EstimatedOutput] = step(obj,y,u)
```

For `recursiveOE` object properties, see “Properties” on page 1-1221.

Examples

Estimate Output-Error Polynomial Model Online

Create a System object for online parameter estimation of a Output-Error polynomial model using recursive estimation algorithms.

```
obj = recursiveOE;
```

The Output-Error model has a default structure with polynomials of order 1 and initial polynomial coefficient values, `eps`.

Load the estimation data. In this example, use a static data set for illustration.

```
load iddata1 z1;
output = z1.y;
input = z1.u;
```

Estimate Output-Error model parameters online using `step`.

```
for i = 1:numel(input)
[B,F,EstimatedOutput] = step(obj,output(i),input(i));
end
```

View the current estimated values of polynomial `F` coefficients.

```
obj.F
```

```
ans =

    1.0000   -0.7618
```

View the current covariance estimate of the parameters.

```
obj.ParameterCovariance
```

```
ans =

    0.0024    0.0002
    0.0002    0.0001
```

View the current estimated output.

```
EstimatedOutput
```

```
EstimatedOutput =

   -4.1866
```

Create System Object for Output-Error Model With Known Orders and Delays

Specify Output-Error polynomial model orders and delays.

```
nb = 1;
nf = 2;
nk = 1;
```

Create a System object for online estimation of Output-Error polynomial model with the specified orders and delays.

```
obj = recursiveOE([nb nf nk]);
```

Create System Object for Output-Error Model With Known Initial Parameters

Specify Output-Error polynomial model orders and delays.

```
nb = 1;
nf = 2;
nk = 1;
```

Create a System object for online estimation of Output-Error model with known initial polynomial coefficients.

```
B0 = [0 1];
F0 = [1 0.7 0.8];
obj = recursiveOE([nb nf nk],B0,F0);
```

Specify the initial parameter covariance.

```
obj.InitialParameterCovariance = 0.1;
```

`InitialParameterCovariance` represents the uncertainty in your guess for the initial parameters. Typically, the default `InitialParameterCovariance` (10000) is too large relative to the parameter values. This results in initial guesses being given less importance during estimation. If you have confidence in the initial parameter guesses, specify a smaller initial parameter covariance.

Specify Estimation Method for Online Estimation of Output-Error Model

Create a System object that uses the unnormalized gradient algorithm for online parameter estimation of an Output-Error model.

```
obj = recursiveOE([1 2 1],'EstimationMethod','Gradient');
```

- “Perform Online Parameter Estimation at the Command Line”
- “Validate Online Parameter Estimation at the Command Line”

Input Arguments

`Orders` — Model orders and delays

1-by-3 vector of integers

Model orders and delays of a Output-Error polynomial model , specified as a 1-by-3 vector of integers, `[nb nf nk]`.

- `nb` — Order of the polynomial $B(q) + 1$, specified as a positive integer.
- `nf` — Order of the polynomial $F(q)$, specified as a nonnegative integer.

- `nk` — Input-output delay, specified as a positive integer. `nk` is number of input samples that occur before the input affects the output. `nk` is expressed as fixed leading zeros of the *B* polynomial.

`B0,F0` — Initial value of polynomial coefficients

row vectors of real values | `[]`

Initial value of polynomial coefficients, specified as row vectors of real values with elements in order of ascending powers of q^{-1}.

- `B0` — Initial guess for the coefficients of the polynomial $B(q)$, specified as a 1-by-(`nb+nk`) vector with `nk` leading zeros.
- `F0` — Initial guess for the coefficients of the polynomial $F(q)$, specified as a 1-by-(`nf+1`) vector with 1 as the first element.

  The coefficients in `F0` must define a stable discrete-time polynomial with roots within a unit disk. For example,

  ```
  F0 = [1 0.5 0.5];
  all(abs(roots(F0))<1)
  ```

  ```
  ans =

       1
  ```

Specifying as `[]`, uses the default value of `eps` for the polynomial coefficients.

Note: If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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 writable properties of `recursiveOE` System object during object creation. For example, `obj = recursiveOE([1 2`

`1],'EstimationMethod','Gradient')` creates a System object to estimate a Output-Error polynomial model using the `'Gradient'` recursive estimation algorithm.

Properties

`recursiveOE` System object properties consist of read-only and writable properties. The writable properties are tunable and nontunable properties. The nontunable properties cannot be changed when the object is locked, that is, after you use the `step` command.

Use `Name,Value` arguments to specify writable properties of `recursiveOE` objects during object creation. After object creation, use dot notation to modify the tunable properties.

```
obj = recursiveOE;
obj.ForgettingFactor = 0.99;
```

B

Estimated coefficients of polynomial $B(q)$, returned as a vector of real values specified in order of ascending powers of q^{-1}.

`B` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

F

Estimated coefficients of polynomial $F(q)$, returned as a vector of real values specified in order of ascending powers of q^{-1}.

`F` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

InitialB

Initial values for the coefficients of polynomial $B(q)$ of order `nb-1`, specified as a row vector of length `nb+nk`, with `nk` leading zeros. `nk` is the input-output delay. Specify the coefficients in order of ascending powers of q^{-1}.

If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

`InitialB` is a tunable property. You can change it when the object is in a locked state.

Default: `[0 eps]`

InitialF

Initial values for the coefficients of polynomial *F*(*q*) of order `nf`, specified as a row vector of length `nf+1`, with 1 as the first element. Specify the coefficients in order of ascending powers of q^{-1}.

The coefficients in `InitialF` must define a stable discrete-time polynomial with roots within a unit circle. For example,

```
InitialF = [1 0.9 0.8];
all(abs(roots(InitialF))<1)
```

```
ans =

     1
```

If the initial guesses are much smaller than the default `InitialParameterCovariance`, 10000, the initial guesses are given less importance during estimation. In that case, specify a smaller initial parameter covariance.

`InitialF` is a tunable property. You can change it when the object is in a locked state.

Default: `[1 eps]`

ParameterCovariance

Estimated covariance of the parameters, returned as an *N*-by-*N* symmetric positive-definite matrix. *N* is the number of parameters to be estimated.

Applicable when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'`.

`ParameterCovariance` is computed assuming that the residuals (difference between estimated and measured outputs) are white noise, and the variance of these residuals is 1.

`ParameterCovariance` is a read-only property and is initially empty after you create the object. It is populated after you use the `step` command for online parameter estimation.

`InitialParameterCovariance`

Covariance of the initial parameter estimates, specified as one of the following:

- Real positive scalar, a — Covariance matrix is an N-by-N diagonal matrix, with a as the diagonal elements. N is the number of parameters to be estimated.
- Vector of real positive scalars, $[a_1,...,a_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[a_1,...,a_N]$ as the diagonal elements.
- N-by-N symmetric positive-definite matrix.

`InitialParameterCovariance` represents the uncertainty in the initial parameter estimates. For large values of `InitialParameterCovariance`, less importance is placed on the initial parameter values and more on the measured data during beginning of estimation using `step`.

Use only when `EstimationMethod` is `'ForgettingFactor'` or `'KalmanFilter'`.

`InitialParameterCovariance` is a tunable property. You can change it when the object is in a locked state.

Default: `10000`

`EstimationMethod`

Recursive estimation algorithm used for online estimation of model parameters, specified as one of the following values:

- `'ForgettingFactor'` — Algorithm used for parameter estimation
- `'KalmanFilter'` — Algorithm used for parameter estimation
- `'NormalizedGradient'` — Algorithm used for parameter estimation
- `'Gradient'` — Unnormalized gradient algorithm used for parameter estimation

Forgetting factor and Kalman filter algorithms are more computationally intensive than gradient and unnormalized gradient methods. However, they have better convergence properties. For information about these algorithms, see “Recursive Algorithms for Online Parameter Estimation”.

`EstimationMethod` is a nontunable property. You cannot change it during execution, that is after the object is locked using the `step` command. If you want to deploy code using MATLAB Coder, `EstimationMethod` can only be assigned once.

Default: `'ForgettingFactor'`

`ForgettingFactor`

Forgetting factor, λ, relevant for parameter estimation, specified as a scalar in the range (0,1].

Suppose that the system remains approximately constant over T_0 samples. You can choose λ such that:

$$T_0 = \frac{1}{1-\lambda}$$

- Setting $\lambda = 1$ corresponds to “no forgetting” and estimating constant coefficients.
- Setting $\lambda < 1$ implies that past measurements are less significant for parameter estimation and can be “forgotten”. Set $\lambda < 1$ to estimate time-varying coefficients.

Typical choices of λ are in the range `[0.98 0.995]`.

Use only when `EstimationMethod` is `'ForgettingFactor'`.

`ForgettingFactor` is a tunable property. You can change it when the object is in a locked state.

Default: `1`

`EnableAdapation`

Enable or disable parameter estimation, specified as one of the following:

- `true` or `1`— The `step` command estimates the parameter values for that time step and updates the parameter values.
- `false` or `0` — The `step` command does not update the parameters for that time step and instead outputs the last estimated value. You can use this option when your system enters a mode where the parameter values do not vary with time.

Note: If you set `EnableAdapation` to `false`, you must still execute the `step` command. Do not skip `step` to keep parameter values constant, because parameter estimation depends on current and past I/O measurements. `step` ensures past I/O data is stored, even when it does not update the parameters.

`EnableAdapation` is a tunable property. You can change it when the object is in a locked state.

Default: `true`

`DataType`

Floating point precision of parameters, specified as one of the following values:

- `'double'` — Double-precision floating point
- `'single'` — Single-precision floating point

Setting `DataType` to `'single'` saves memory, but leads to loss of precision. Specify `DataType` based on the precision required by the target processor where you will deploy generated code.

`DataType` is a nontunable property. It can only be set during object construction using `Name,Value` arguments and cannot be changed afterward.

Default: `'double'`

`ProcessNoiseCovariance`

Covariance matrix of parameter variations, specified as one of the following:

- Real nonnegative scalar, a — Covariance matrix is an N-by-N diagonal matrix, with a as the diagonal elements.
- Vector of real nonnegative scalars, $[a_1,...,a_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[a_1,...,a_N]$ as the diagonal elements.
- N-by-N symmetric positive semidefinite matrix.

N is the number of parameters to be estimated.

`ProcessNoiseCovariance` is applicable when `EstimationMethod` is `'KalmanFilter'`.

Kalman filter algorithm treats the parameters as states of a dynamic system and estimates these parameters using a Kalman filter. `ProcessNoiseCovariance` is the covariance of the process noise acting on these parameters. Zero values in the noise covariance matrix correspond to estimating constant coefficients. Values larger than 0 correspond to time-varying parameters. Use large values for rapidly changing parameters. However, the larger values result in noisier parameter estimates.

`ProcessNoiseCovariance` is a tunable property. You can change it when the object is in a locked state.

Default: `0.1`

`AdaptationGain`

Adaptation gain, γ, used in gradient recursive estimation algorithms, specified as a positive scalar.

`AdaptationGain` is applicable when `EstimationMethod` is `'Gradient'` or `'NormalizedGradient'`.

Specify a large value for `AdaptationGain` when your measurements have a high signal-to-noise ratio.

`AdaptationGain` is a tunable property. You can change it when the object is in a locked state.

Default: `1`

`NormalizationBias`

Bias in adaptation gain scaling used in the `'NormalizedGradient'` method, specified as a nonnegative scalar.

`NormalizationBias` is applicable when `EstimationMethod` is `'NormalizedGradient'`.

The normalized gradient algorithm divides the adaptation gain at each step by the square of the two-norm of the gradient vector. If the gradient is close to zero, this can cause jumps in the estimated parameters. `NormalizationBias` is the term introduced in the denominator to prevent these jumps. Increase `NormalizationBias` if you observe jumps in estimated parameters.

`NormalizationBias` is a tunable property. You can change it when the object is in a locked state.

Default: `eps`

Output Arguments

`obj` — System object for online parameter estimation of Output-Error polynomial model

`recursiveOE` System object

System object for online parameter estimation of SISO Output-Error polynomial model, returned as a `recursiveOE` System object. This object is created using the specified model orders and properties. Use `step` command to estimate the coefficients of the Output-Error model polynomials. You can then access the estimated coefficients and parameter covariance using dot notation. For example, type `obj.B` to view the estimated *B* polynomial coefficients.

More About

Output-Error Model Structure

The general Output-Error model structure is:

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

The orders of the Output-Error model are:

$$\begin{aligned} nb&: \quad B(q) = b_1 + b_2 q^{-1} + ... + b_{nb} q^{-nb+1} \\ nf&: \quad F(q) = 1 + f_1 q^{-1} + ... + f_{nf} q^{-nf} \end{aligned}$$

Tips

- Starting in R2016b, instead of using the `step` command to update model parameter estimates, you can call the System object with input arguments, as if it were a function. For example, `[B,F,EstimatedOutput] = step(obj,y,u)` and `[B,F,EstimatedOutput] = obj(y,u)` perform equivalent operations.
- “What Is Online Estimation?”

- "Recursive Algorithms for Online Parameter Estimation"

See Also

`clone` | `isLocked` | `oe` | `Recursive Polynomial Model Estimator` | `recursiveAR` | `recursiveARMA` | `recursiveARMAX` | `recursiveARX` | `recursiveBJ` | `recursiveLS` | `release` | `reset` | `step`

Introduced in R2015b

release

Unlock online parameter estimation System object

Syntax

```
release(obj)
```

Description

`release(obj)` unlocks the online parameter estimation System object, `obj`. Use `release` to change nontunable properties of the object.

Note: You can use `release` on a System object in code generated using MATLAB Coder, but once you release its resources, you cannot use that System object again.

Examples

Unlock Online Estimation System Object

Create a System object™ for online estimation of an ARMAX model with default properties.

```
obj = recursiveARMAX;
```

Estimate model parameters online using `step` and input-output data.

```
[A,B,C,EstimatedOutput] = step(obj,1,1);
```

`step` puts the object in a locked state.

```
L = isLocked(obj)
```

```
L =

  logical
```

```
   1
```

Unlock the object.

```
release(obj)
```

Check the locked status of the object.

```
L = isLocked(obj)
```

```
L =

  logical

   0
```

- “Perform Online Parameter Estimation at the Command Line”
- “Validate Online Parameter Estimation at the Command Line”

Input Arguments

`obj` — System object for online parameter estimation

`recursiveAR` object | `recursiveARMA` object | `recursiveARX` object | `recursiveARMAX` object | `recursiveOE` object | `recursiveBJ` object | `recursiveLS` object

System object for online parameter estimation, created using one of the following commands:

- `recursiveAR`
- `recursiveARMA`
- `recursiveARX`
- `recursiveARMAX`
- `recursiveOE`
- `recursiveBJ`
- `recursiveLS`

More About

- “What Is Online Estimation?”

See Also

`clone` | `isLocked` | `recursiveAR` | `recursiveARMA` | `recursiveARMAX` | `recursiveARX` | `recursiveBJ` | `recursiveLS` | `recursiveOE` | `reset` | `step`

Introduced in R2015b

repsys

Replicate and tile models

Syntax

```
rsys = repsys(sys,[M N])
rsys = repsys(sys,N)
rsys = repsys(sys,[M N S1,...,Sk])
```

Description

`rsys = repsys(sys,[M N])` replicates the model `sys` into an `M`-by-`N` tiling pattern. The resulting model `rsys` has `size(sys,1)*M` outputs and `size(sys,2)*N` inputs.

`rsys = repsys(sys,N)` creates an N-by-N tiling.

`rsys = repsys(sys,[M N S1,...,Sk])` replicates and tiles `sys` along both I/O and array dimensions to produce a model array. The indices `S` specify the array dimensions. The size of the array is `[size(sys,1)*M, size(sys,2)*N, size(sys,3)*S1, ...]`.

Input Arguments

`sys`

Model to replicate.

`M`

Number of replications of `sys` along the output dimension.

`N`

Number of replications of `sys` along the input dimension.

S

Numbers of replications of `sys` along array dimensions.

Output Arguments

rsys

Model having `size(sys,1)*M` outputs and `size(sys,2)*N` inputs.

If you provide array dimensions `S1,...,Sk`, `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.
```

More About

Tips

`rsys = repsys(sys,N)` produces the same result as `rsys = repsys(sys,[N N])`. To produce a diagonal tiling, use `rsys = sys*eye(N)`.

See Also

`append`

Introduced in R2010b

resample

Resample time-domain data by decimation or interpolation (requires Signal Processing Toolbox software)

Syntax

```
resample(data,P,Q)
resample(data,P,Q,order)
```

Description

`resample(data,P,Q)` resamples data such that the data is interpolated by a factor `P` and then decimated by a factor `Q`. `resample(z,1,Q)` results in decimation by a factor `Q`.

`resample(data,P,Q,order)` filters the data by applying a filter of specified `order` before interpolation and decimation.

Input Arguments

`data`

Name of time-domain `iddata` object. Can be input-output or time-series data.

Data must be sampled at equal time intervals.

`P`, `Q`

Integers that specify the resampling factor, such that the new sample time is `Q/P` times the original one.

`(Q/P)>1` results in decimation and `(Q/P)<1` results in interpolation.

`order`

Order of the filters applied before interpolation and decimation.

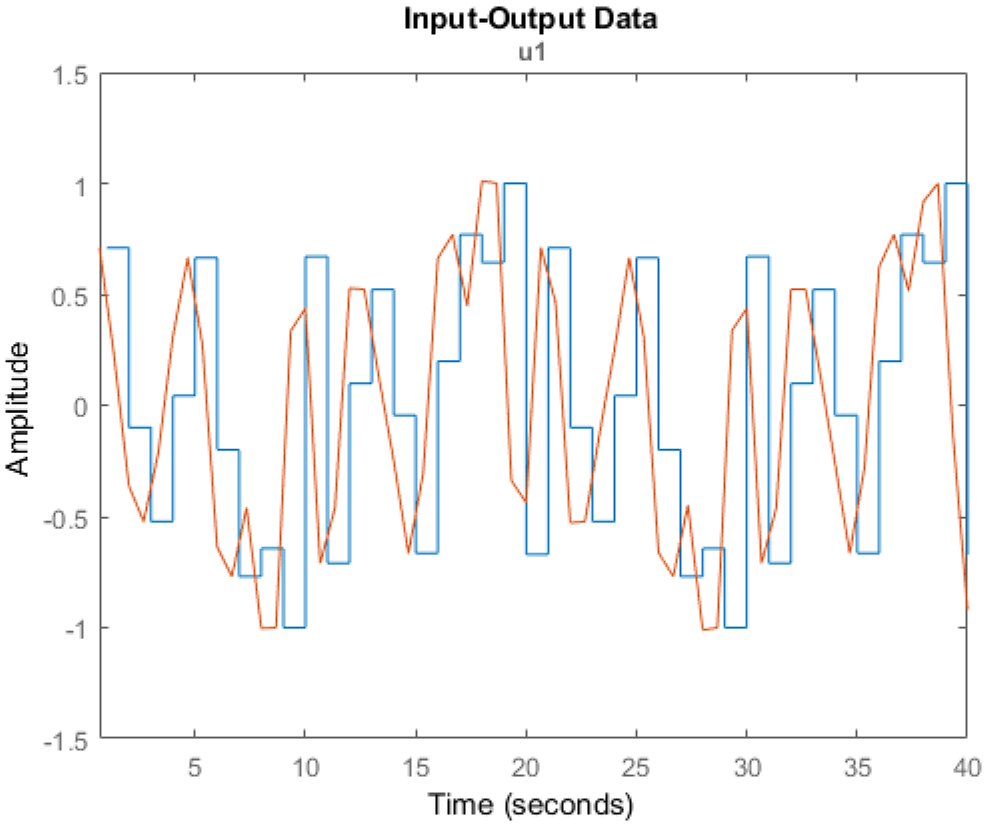
Default: `10`

Examples

Resample Time-Domain Data

Increase the sampling rate of data by a factor of 1.5 and compare the resampled and the original data signals.

```
u = idinput([20 1 2],'sine',[],[],[5 10 1]);
u = iddata([],u,1);
plot(u)
ur = resample(u,3,2);
plot(u,ur)
```



More About

Algorithms

If you have installed the Signal Processing Toolbox software, `resample` calls the Signal Processing Toolbox `resample` function. The algorithm takes into account the intersample characteristics of the input signal, as described by `data.InterSample`.

See Also

`idresamp`

Introduced before R2006a

reset

Reset online parameter estimation System object

Syntax

```
reset(obj)
```

Description

`reset(obj)` resets the states of a locked online parameter estimation System object, `obj`, to initial values and leaves the object locked. The states of the object are the estimated parameters and parameter covariance. Use `reset` if you are not satisfied with the estimation or if your system changes modes.

Examples

Reset Online Estimation System Object

Create a System object for online estimation of an Output-Error model.

```
obj = recursiveOE('InitialB',[0 0.5],'InitialF',[1 0.8],...
    'InitialParameterCovariance',0.1);
```

Load the estimation data. For this example, use a static data set for illustration.

```
load iddata1 z1;
output = z1.y;
input = z1.u;
```

Estimate model parameters online using `step`.

```
for i = 1:numel(input)
  [B,F,EstimatedOutput] = step(obj,output(i),input(i));
end
```

View the object properties.

```
obj
```

```
obj =

  recursiveOE with properties:

                           B: [0 2.0014]
                           F: [1 -0.7639]
                    InitialB: [0 0.5000]
                    InitialF: [1 0.8000]
         ParameterCovariance: [2×2 double]
  InitialParameterCovariance: [2×2 double]
            EstimationMethod: 'ForgettingFactor'
            ForgettingFactor: 1
            EnableAdaptation: true
                    DataType: 'double'
```

Reset the System object.

```
reset(obj)
```

The estimated parameters, `B` and `F`, and parameter covariance, `ParameterCovariance` are reset to the initial values.

```
obj
```

```
obj =

  recursiveOE with properties:

                           B: [0 0.5000]
                           F: [1 0.8000]
                    InitialB: [0 0.5000]
                    InitialF: [1 0.8000]
         ParameterCovariance: [2×2 double]
  InitialParameterCovariance: [2×2 double]
            EstimationMethod: 'ForgettingFactor'
            ForgettingFactor: 1
            EnableAdaptation: true
                    DataType: 'double'
```

- “Perform Online Parameter Estimation at the Command Line”
- “Validate Online Parameter Estimation at the Command Line”

Input Arguments

`obj` — System object for online parameter estimation

`recursiveAR` object | `recursiveARMA` object | `recursiveARX` object | `recursiveARMAX` object | `recursiveOE` object | `recursiveBJ` object | `recursiveLS` object

System object for online parameter estimation, created using one of the following commands:

Online Estimation System Object	Estimated Parameters
`recursiveAR`	`A` — Reset to `InitialA`
`recursiveARMA`	`A` — Reset to `InitialA` `C` — Reset to `InitialC`
`recursiveARX`	`A` — Reset to `InitialA` `B` — Reset to `InitialB`

`recursiveARMAX`	`A` — Reset to `InitialA`
	`B` — Reset to `InitialB`
	`C` — Reset to `InitialC`

`recursiveOE`	`B` — Reset to `InitialB` `F` — Reset to `InitialF`

`recursiveBJ`	`B` — Reset to `InitialB`
	`C` — Reset to `InitialC`
	`D` — Reset to `InitialD`
	`F` — Reset to `InitialF`

`recursiveLS`	`Parameters` — Reset to `InitialParameters`

When `EstimationMethod` property of `obj` is `'ForgettingFactor'` or `'KalmanFilter'`, the `ParameterCovariance` property of `obj` is reset to the value of `InitialParameterCovariance`.

More About

- “What Is Online Estimation?”

See Also

`clone` | `isLocked` | `recursiveAR` | `recursiveARMA` | `recursiveARMAX` | `recursiveARX` | `recursiveBJ` | `recursiveLS` | `recursiveOE` | `release` | `step`

Introduced in R2015b

reshape

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`

Introduced before R2006a

resid

Compute and test residuals

Syntax

```
resid(Data,sys)
resid(Data,sys,Linespec)
resid(Data,sys1,...,sysn)
resid(Data,sys1,Linespec1,...,sysn,Linespecn)

resid(___,Options)

resid(___,Type)

[E,R] = resid(Data,sys)
```

Description

`resid(Data,sys)` computes the 1-step-ahead prediction errors (residuals) for an identified model, `sys`, and plots residual-input dynamics as one of the following, depending on the data in`Data`:

- For time-domain data, `resid` plots the auto-=correlation of the residuals and the cross-correlation of the residuals with the input signals. The correlations are generated for lags -25 to 25. To specify a different maximum lag value, use `residOptions`. The 99% confidence region marking statistically insignificant correlations displays as a shaded region around the X-axis.
- For frequency-domain data, `resid` plots a bode plot of the frequency response from the input signals to the residuals. The 99% confidence region marking statistically insignificant response is shown as a region around the X-axis.

To change display options, right-click the plot to access the context menu. For more details about the menu, see “Tips” on page 1-1256.

`resid(Data,sys,Linespec)` sets the line style, marker symbol, and color.

`resid(Data,sys1,...,sysn)` computes and plots the residual of multiple identified models `sys1`,...,`sysn`.

`resid(Data,sys1,Linespec1,...,sysn,Linespecn)` sets the line style, marker symbol, and color for each system.

`resid( ___ ,Options)` specifies additional residual calculation options. Use `Options` with any of the previous syntaxes.

`resid( ___ ,Type)` specifies the plot type. Use `Type` with any of the previous syntaxes.

`[E,R] = resid(Data,sys)` returns the calculated residuals, `E`, and residual correlations, `R`. No plot is generated.

Examples

Plot Model Residual Correlations

Load time-domain data.

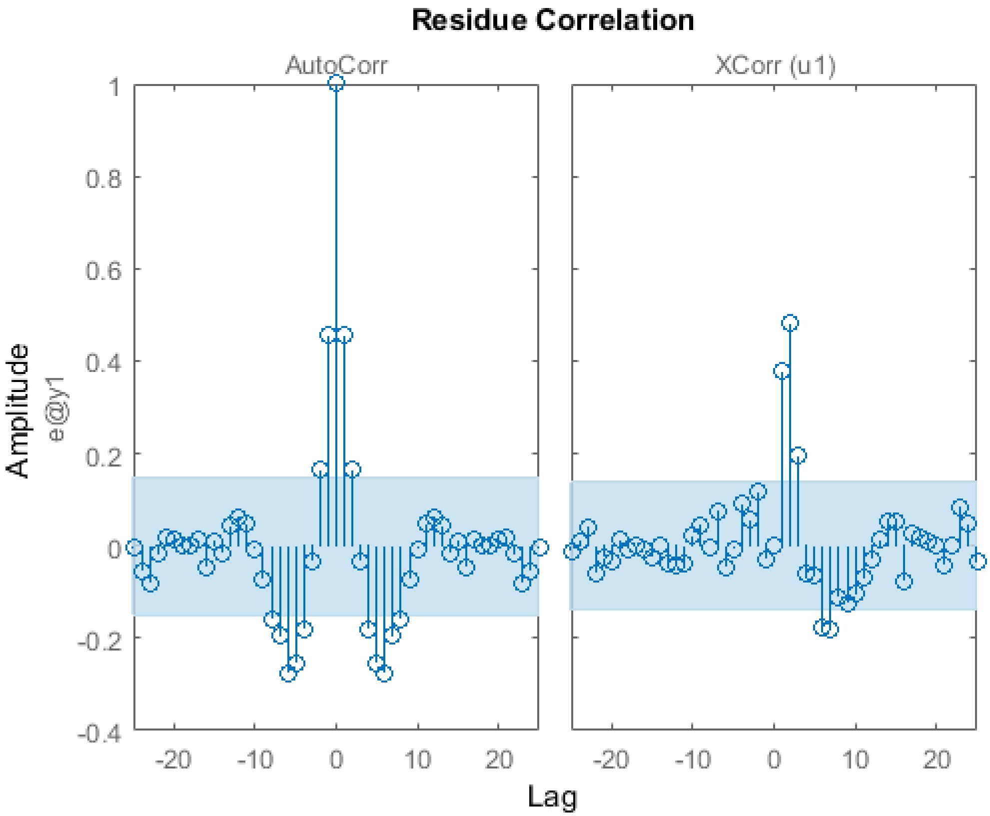
```
load iddata1
data = z1;
```

Estimate an ARX model.

```
sys = arx(data,[1 1 0]);
```

Plot the autocorrelation of the residuals and cross-correlation between the residuals and the inputs.

```
resid(data,sys)
```



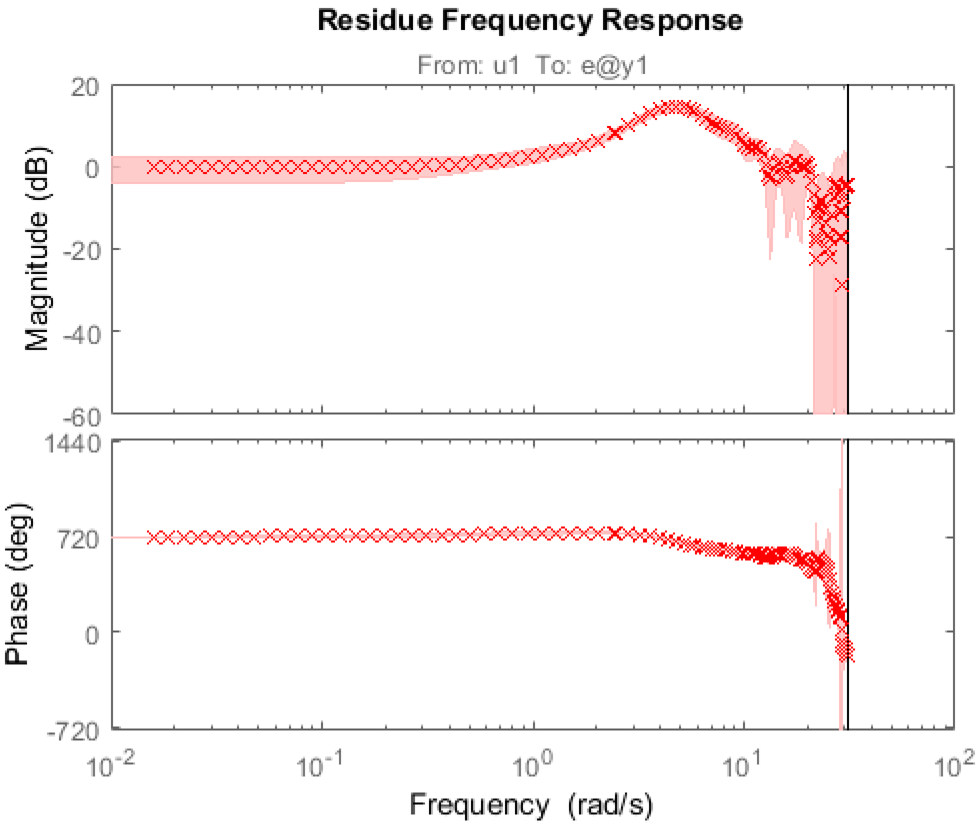
The correlations are calculated until the default maximum lag, 25. The 99% confidence region marking statistically insignificant correlations displays as a shaded region around the X-axis.

Convert data to frequency domain.

```
data2 = fft(data);
```

Compute the residuals for identified model, `sys`, and the frequency-domain data. Plot the residual response using red crosses.

```
resid(data2,sys,'rx')
```



For frequency-domain data, `resid` plots the Bode plot showing frequency response from the input to the residuals.

Compare the Residuals for Multiple Identified Models

Load time-domain data.

```
load iddata1
```

Estimate an ARX model.

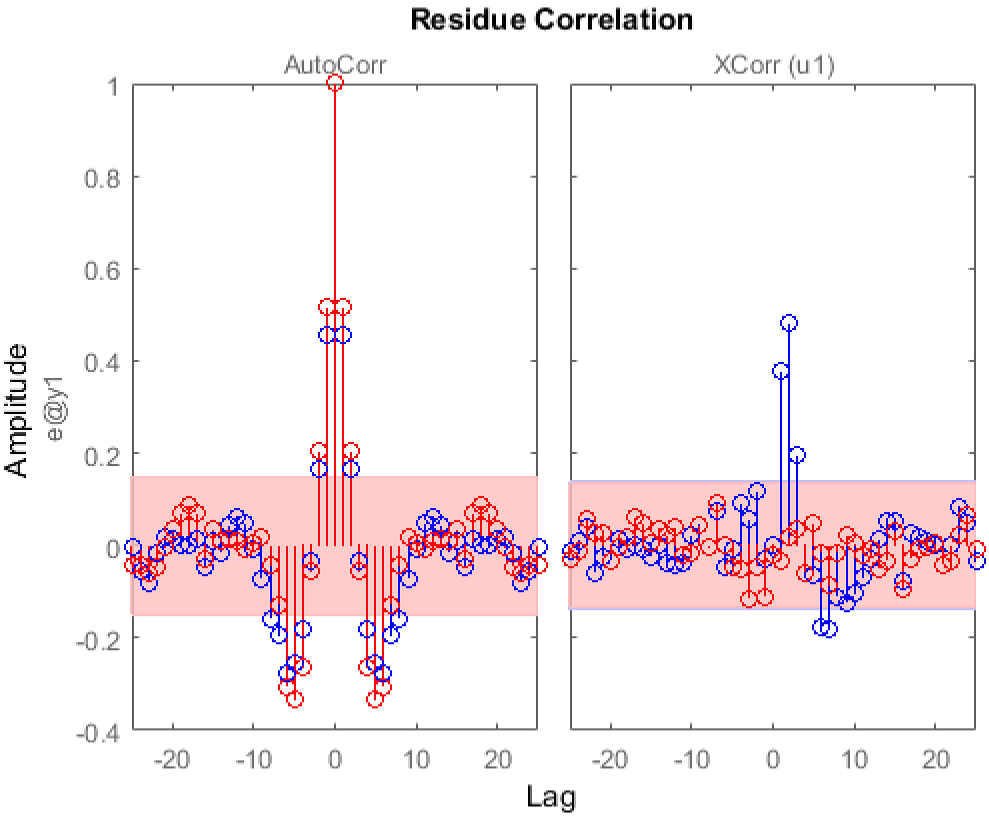
```
sys1 = arx(z1,[1 1 0]);
```

Estimate a transfer function model.

```
sys2 = tfest(z1,2);
```

Plot the correlations of the residuals.

```
resid(z1,sys1,'b',sys2,'r')
```



The cross-correlation between residuals of `sys2` and the inputs lie in the 99% confidence band for all lags.

Specify Maximum Lag for Residual Impulse Response Calculations

Load time-domain data.

```
load iddata1
```

Estimate an ARX model.

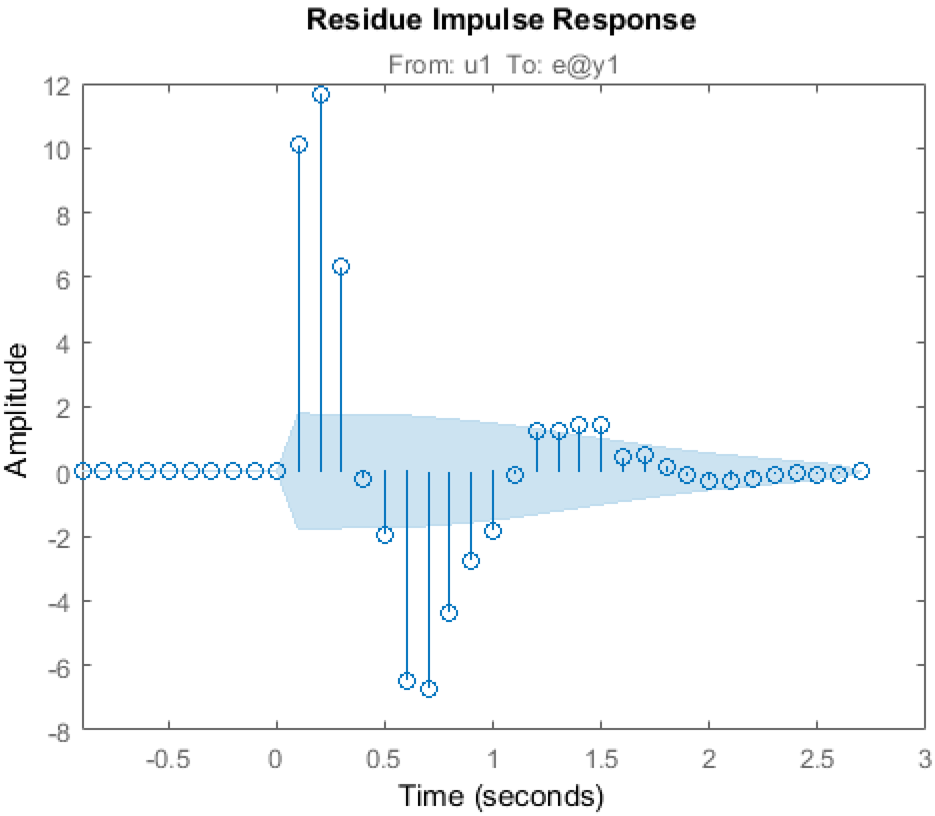
```
sys = arx(z1,[1 1 0]);
```

Specify the maximum lag for residual correlation calculations.

```
opt = residOptions('MaxLag',35);
```

Plot the impulse response from the input to the residuals.

```
resid(z1,sys,opt,'ir')
```



Calculate Residuals for a MIMO System

Load time-domain data.

```
load iddata7
```

The data is a two-input, single-output dataset.

Estimate an ARX model.

```
sys = tfest(z7,2);
```

Calculate the residuals and their autocorrelations and cross-correlations with inputs.

```
[E,R] = resid(z7,sys);
```

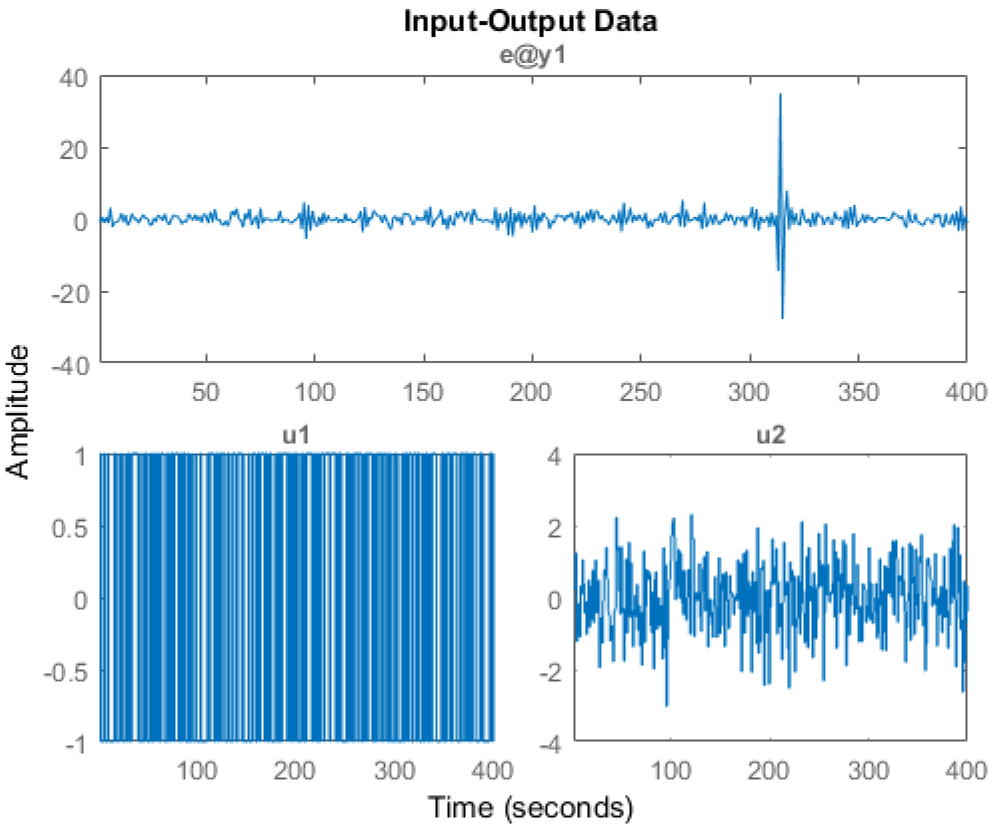
`R` is a 26-by-3-by-3 matrix of correlations. For example,

- `R(:,1,1)` is the autocorrelation of the residuals until lag 25.
- `R(:,1,2)` is the cross-correlation of the residuals with the first input,until lag 25.

`E` is an iddata object with the residuals as output data and the inputs of validation data (z7) as input data. You can use `E` to identify error models and analyze the error dynamics.

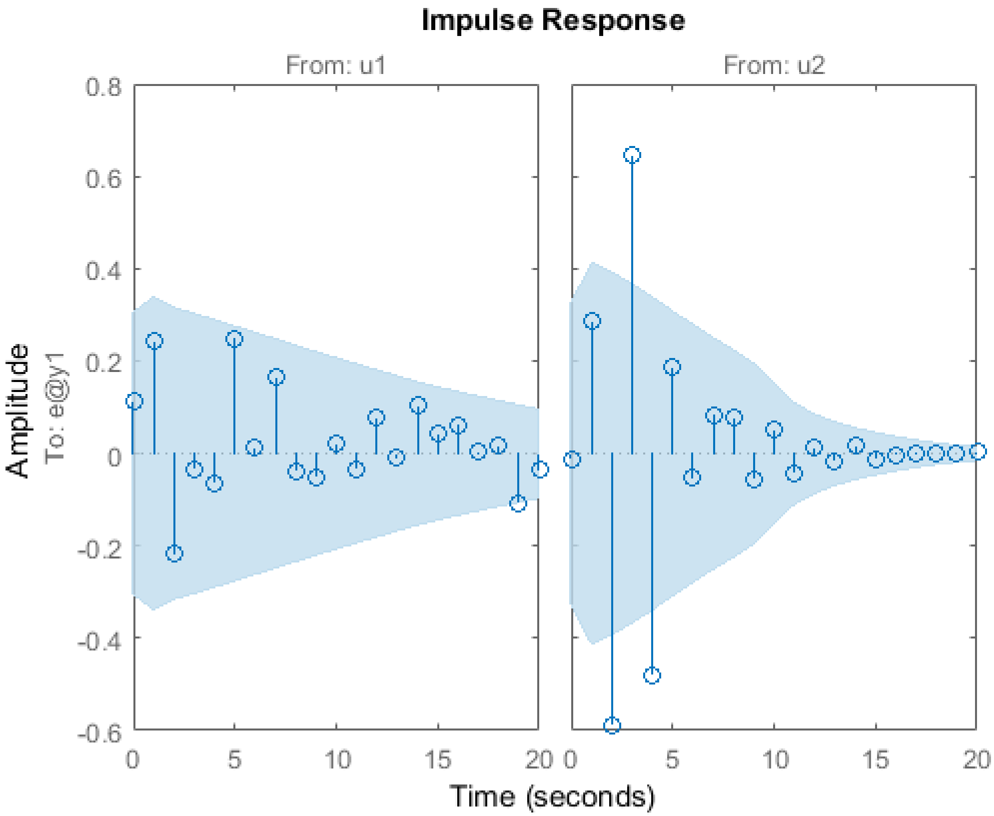
Plot the error.

```
plot(E)
```



Estimate impulse response between inputs and residuals. Plot them with a 3 standard deviation confidence region.

```
I = impulseest(E);
showConfidence(impulseplot(I,20),3)
```



Input Arguments

`Data` — Validation data

`iddata` object

Validation input-output data, specified as an `iddata` object. `Data` can have multiple input-output channels. When `sys` is linear, `Data` is time-domain or frequency-domain. When `sys` is nonlinear, `Data` is time-domain.

`sys` — System used for computing residuals

identified linear or nonlinear model

System used for computing residuals, specified as an identified linear or nonlinear model.

Example: `idpoly`

`Linespec` — Line style, marker symbol, and color

character vector

Line style, marker symbol, and color, specified as a character vector. For more information, see `plot` in the MATLAB documentation. When `Type` is specified as `'corr'`, only the line style is used.

Example: `'Linespec','kx'`

`Options` — Residual analysis options

`residOptions` option set

Residual analysis options, specified as an `residOptions` option set.

`Type` — Plot type

`'corr'` | `'ir'` | `'fr'`

Plot type, specified as one of the following values:

- `'corr'` — Plots the autocorrelation of the residuals, `e`, and the cross-correlation of the residuals with the input signals, `u`. The correlations are generated for lags -25 to 25. Use `residOptions` to specify a different maximum lag value. The 99% confidence region marking statistically insignificant correlations is also shown as a shaded region around the X-axis. The computation of the confidence region is done assuming `e` to be white and independent of `u`.

  `'corr'` is default for time-domain data. This plot type is not available for frequency-domain data.
- `'ir'` — Plots the impulse response up to lag 25 of a system from the input to the residuals. The `impulseest` command first estimates the impulse response model with `e` as output data and `u` as inputs. Then `impulseest` calculates the impulse response of the estimated model. The 99% confidence region marking statistically insignificant response displays as a shaded region. A low magnitude indicates a reliable model.

  This plot type is not available for frequency-domain data.
- `'fr'` — The frequency response from the input to the residuals (based on a high-order FIR model) is shown as a Bode plot. The 99% confidence region marking

statistically insignificant response displays as a shaded region. A low magnitude in the frequency range of interest indicates a reliable model.

`'fr'` is default for frequency-domain data.

Output Arguments

`E` — Model residuals

`iddata` object

Model residuals, returned as an `iddata` object. The residuals are stored in `E.OutputData`, and the inputs are stored in `E.InputData`. Use `E` to build models that describe the dynamics from the inputs to the residuals. The dynamics are negligible if `sys` is a reliable identified model.

`R` — Correlations of the residuals

matrix of doubles | `[]`

Correlations of the residuals, returned as one of the following:

- Matrix of doubles — For time-domain-data

  `R` is a matrix of size M+1-by-(ny+nu)-by-(ny+nu). Where, M is the maximum lag specified in `Options`, ny is the number of outputs, and nu is the number of inputs. The default value of M is 25.

  At each lag k (`k = 0:M`), `R(k,i,j)` is the expectation value, `<Z(t,i).Z(t+k-1,j)>`. Here, `Z = [E.OutputData,E.InputData]`.

  For example, for a two-output, single-input model, `Z = [e1,e2,u1]`. Where, $e1$ is the residual of the first output, $e2$ is the residual of the second output, and $u1$ is the input. `R` is a `26`-by-`3`-by-`3` matrix, where:

  - `R(5,1,2) = <e1(t).e2(t+4)>` is the cross-correlation at lag 4 between $e1$ and $e2$.
  - `R(5,1,3) = <e1(t).u1(t+4)>` is the cross-correlation at lag 4 between $e1$ and $u1$.
  - `R(5,1,1)`, `R(5,2,2)`, `R(5,3,3)` are the autocorrelations at lag 4 for $e1$, $e2$, and $u1$, respectively.

- [ ] — For frequency-domain data

More About

Tips

- Right-clicking the plot opens the context menu, where you can access the following options:
    - **Systems** — Select systems to view the residual correlation or response plots. By default, all systems are plotted.
    - **Show Confidence Region** — View the 99% confidence region marking statistically insignificant correlations. Applicable only for the correlation plots.
    - **Data Experiment** — For multi-experiment data only. Toggle between data from different experiments.
    - **Characteristics** — View data characteristics. Not applicable for correlation plots.
        - **Peak Response** — View peak response of the data.
        - **Confidence Region** — View the 99% confidence region marking statistically insignificant response.
    - **Show** — Applicable only for frequency-response plots.
        - **Magnitude** — View magnitude of frequency response.
        - **Phase** — View phase of frequency response.
    - **I/O Grouping** — For datasets containing more than one input or output channel. Select grouping of input and output channels on the plot. Not applicable for correlation plots.
        - **None** — Plot input-output channels in their own separate axes.
        - **All** — Group all input channels together and all output channels together.
    - **I/O Selector** — For datasets containing more than one input or output channel. Select a subset of the input and output channels to plot. By default, all output channels are plotted.
    - **Grid** — Add grids to the plot.
    - **Normalize** — Normalize the y-scale of all data in the plot. Not applicable for frequency-response data.

  - **Full View** — Return to full view. By default, the plot is scaled to full view.
  - **Initial Condition** — Specify handling of initial conditions.

    Specify as one of the following:

    - **Estimate** — Treat the initial conditions as estimation parameters.
    - **Zero** — Set all initial conditions to zero.
    - **Absorb delays and estimate** — Absorb nonzero delays into the model coefficients and treat the initial conditions as estimation parameters. Use this option for discrete-time models only.
  - **Properties** — Open the Property Editor dialog box to customize plot attributes.

- "What Is Residual Analysis?"

References

[1] Ljung, L. *System Identification: Theory for the User*. Upper Saddle River, NJ: Prentice-Hall PTR, 1999, Section 16.6.

See Also

`compare` | `predict` | `residOptions` | `sim` | `simsd`

Introduced before R2006a

residOptions

Option set for `resid`

Syntax

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

Description

`opt = residOptions` creates the default option set for `resid`. Use dot notation to customize the option set, if needed.

`opt = residOptions(Name,Value)` creates an option set with options specified by one or more `Name,Value` pair arguments. The options that you do not specify retain their default value.

Examples

Create and Modify Default Option Set for Residual Analysis

Create a default option set for `resid`.

```
opt = residOptions;
```

Specify the maximum lag for residual correlation calculations.

```
opt.MaxLag = 35;
```

Specify Options for Residual Analysis

Create an option set for `resid` that specifies initial condition as zero.

```
opt = residOptions('InitialCondition','z');
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example: `residOptions('InitialCondition','e')`

`'MaxLag'` — Maximum positive lag
`25` (default) | positive integer

Maximum positive lag for residual correlation and impulse response calculations, specified as the comma-separated pair consisting of `'MaxLag'` and a positive integer.

`'InitialCondition'` — Handling of initial conditions
`'e'` (default) | `'z'` | `'d'` | vector or matrix | structure | `idpar` object `x0obj` | `'fixed'` | `'model'`

Handling of initial conditions, specified as the comma-separated pair consisting of `'InitialCondition'` and 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.

  Use this option for linear identified models only.
- `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 only for state-space models (`idss`, `idgrey`) and nonlinear models (`idnlarx`, `idnlhw`, `idnlgrey`).
- `io` — Structure with the following fields:
  - `Input`

  - `Output`

  Use the `Input` and `Output` fields to specify the input and output history for a time interval that starts before the start time of the data used by `resid`. If the data used by `resid` 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 *Ne* elements, where *Ne* is the number of experiments.

- `x0obj` — Specification object created using `idpar`. Use `x0obj` to impose constraints on the initial states by fixing their value or specifying minimum and maximum bounds.

  Use this object only for discrete-time state-space models (`idss`, `idgrey`) and nonlinear grey-box models `idnlgrey`.

- `'fixed'` — `sys.InitialStates` determines the values of the initial states, but all the states are considered fixed for estimation.

  Use this option only for `idnlgrey` models.

- `'model'` — `sys.InitialStates` determines the values of the initial states, which states to estimate, and their minimum and maximum values.

  Use this option only for `idnlgrey` models.

`'InputOffset'` — Removal of offset from time-domain input data during estimation

[ ] (default) | vector of positive integers | matrix

Removal of offset from time-domain input data during estimation, specified as the comma-separated pair consisting of `'InputOffset'` and one of the following:

- A column vector of positive integers of length *Nu*, where *Nu* is the number of inputs.
- `[]` — Indicates no offset.
- *Nu*-by-*Ne* matrix — For multi-experiment data, specify `InputOffset` as an *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.

`'OutputOffset'` — Removal of offset from time-domain output data during estimation

[ ] (default) | vector | matrix

Removal of offset from time-domain output data during estimation, specified as the comma-separated pair consisting of `'OutputOffset'` and one of the following:

- A column vector of length *Ny*, where *Ny* is the number of outputs.
- `[]` — Indicates no offset.
- *Ny*-by-*Ne* matrix — For multi-experiment 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.

`'OutputWeight'` — Weight of output for initial condition estimation

`[]` (default) | `'noise'` | matrix

Weight of output for initial condition estimation, specified as the comma-separated pair consisting of `'OutputWeight'` and one of the following:

- `[]` — No weighting is used. This option is the same as using `eye(Ny)` for the output weight. *Ny* is the number of outputs.
- `'noise'` — Inverse of the noise variance stored with the model.
- Matrix of doubles — A positive semidefinite matrix of dimension *Ny*-by-*Ny*. *Ny* is the number of outputs.

Output Arguments

`opt` — Option set for `resid`

`residOptions` option set

Option set for `resid`, returned as an `residOptions` option set.

See Also

`resid`

Introduced in R2016a

retrend

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

Retrend Simulated Model Output

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 a data object with sample time of 0.08 seconds.

```
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);
```

More About

- "Handling Offsets and Trends in Data"

See Also

`getTrend` | `detrend` | `TrendInfo`

Introduced in R2009a

roe

(To be removed) Estimate recursively output-error models (IIR-filters)

Note: `roe` will be removed in a future release. Use `recursiveOE` instead.

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

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

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,

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

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

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,...,bnb,f1,...,fnf]
```

`yhat` is the predicted value of the output, according to the current model; that is, row `k` of `yhat` contains the predicted value of `y(k)` based on all past data.

The actual algorithm is selected with the two arguments `adg` and `adm`. These are described under `rarx`.

The input argument `th0` contains the initial value of the parameters, a row vector consistent with the rows of `thm`. The default value of `th0` is all zeros.

The arguments `P0` and `P` are the initial and final values, respectively, of the scaled covariance matrix of the parameters. The default value of `P0` is 10^4 times the unit matrix. The arguments `phi0`, `psi0`, `phi`, and `psi` contain initial and final values of the data vector and the gradient vector, respectively. The sizes of these depend on the chosen model orders. The normal choice of `phi0` and `psi0` is to use the outputs from a previous call to `roe` with the same model orders. (This call could be a dummy call with default input arguments.) The default values of `phi0` and `psi0` are all zeros.

Note that the function requires that the delay `nk` be larger than `0`. If you want `nk = 0`, shift the input sequence appropriately and use `nk = 1`.

More About

Algorithms

The general recursive prediction error algorithm (11.44) of Ljung (1999) is implemented. See also “Recursive Algorithms for Online Parameter Estimation”.

- “Recursive Algorithms for Online Parameter Estimation”

See Also

`nkshift` | `recursiveOE` | `rpem` | `rplr`

Introduced before R2006a

rpem

Estimate general input-output models using recursive prediction-error minimization method

`rpem` is not compatible with MATLAB Coder or MATLAB Compiler™. For the special cases of ARX, AR, ARMA, ARMAX, Box-Jenkins, and Output-Error models, use `recursiveARX`, `recursiveAR`, `recursiveARMA`, `recursiveARMAX`, `recursiveBJ`, and `recursiveOE`, respectively.

Syntax

```
thm = rpem(z,nn,adm,adg)
```

```
[thm,yhat,P,phi,psi] = rpem(z,nn,adm,adg,th0,P0,phi0,psi0)
```

Description

The parameters of the general linear model structure

$$A(q)y(t) = \frac{B_1(q)}{F_1(q)}u_1(t-nk_1) + \ldots + \frac{B_{nu}(q)}{F_{nu}(q)}u_{nu}(t-nk_{nu}) + \frac{C(q)}{D(q)}e(t)$$

are estimated using a recursive prediction error method.

The input-output data is contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. (In the multiple-input case, `u` contains one column for each input.) `nn` is given as

```
nn = [na nb nc nd nf nk]
```

where `na`, `nb`, `nc`, `nd`, and `nf` are the orders of the model, and `nk` is the delay. For multiple-input systems, `nb`, `nf`, and `nk` are row vectors giving the orders and delays of each input. See "What Are Polynomial Models?" for an exact definition of the orders.

The estimated parameters are returned in the matrix `thm`. The `k`th row of `thm` contains the parameters associated with time `k`; that is, they are based on the data in the rows up to and including row `k` in `z`. Each row of `thm` contains the estimated parameters in the following order.

```
thm(k,:) = [a1,a2,...,ana,b1,...,bnb,...
  c1,...,cnc,d1,...,dnd,f1,...,fnf]
```

For multiple-input systems, the *B* part in the above expression is repeated for each input before the *C* part begins, and the *F* part is also repeated for each input. This is the same ordering as in `m.par`.

`yhat` is the predicted value of the output, according to the current model; that is, row `k` of `yhat` contains the predicted value of `y(k)` based on all past data.

The actual algorithm is selected with the two arguments `adg` and `adm`:

- `adm = 'ff'` and `adg` = `lam` specify the *forgetting factor* algorithm with the forgetting factor λ=`lam`. This algorithm is also known as recursive least squares (RLS). In this case, the matrix `P` has the following interpretation: R_2`/2 * P` is approximately equal to the covariance matrix of the estimated parameters.R_2 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.

The input argument `th0` contains the initial value of the parameters, a row vector consistent with the rows of `thm`. The default value of `th0` is all zeros.

The arguments `P0` and `P` are the initial and final values, respectively, of the scaled covariance matrix of the parameters. The default value of `P0` is 10^4 times the 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`.

Examples

Estimate Model Parameters Using Recursive Prediction-Error Minimization

Specify the order and delays of a polynomial model structure.

```
na = 2;
nb = 1;
nc = 1;
nd = 1;
nf = 0;
nk = 1;
```

Load the estimation data.

```
load iddata1 z1
```

Estimate the parameters using forgetting factor algorithm with forgetting factor 0.99.

```
EstimatedParameters = rpem(z1,[na nb nc nd nf nk],'ff',0.99);
```

Get the last set of estimated parameters.

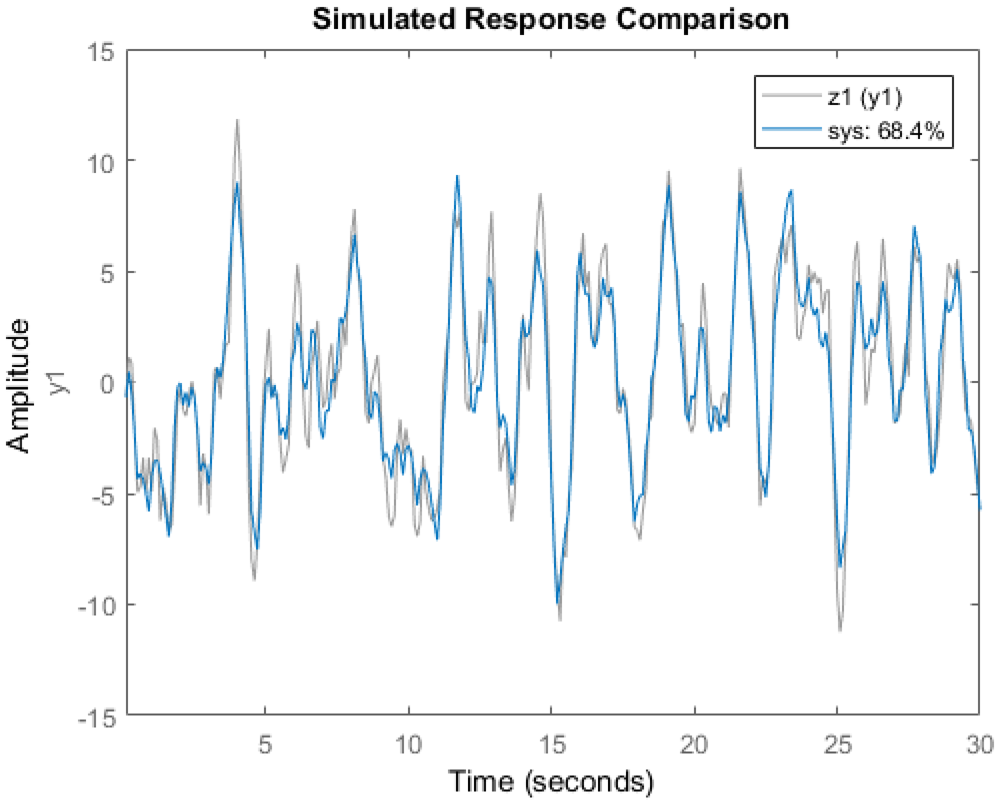
```
p = EstimatedParameters(end,:);
```

Construct a polynomial model with the estimated parameters.

```
sys = idpoly([1 p(1:na)],... % A polynomial
    [zeros(1,nk) p(na+1:na+nb)],... % B polynomial
    [1 p(na+nb+1:na+nb+nc)],... % C polynomial
    [1 p(na+nb+nc+1:na+nb+nc+nd)]); % D polynomial
sys.Ts = z1.Ts;
```

Compare the estimated output with measured data.

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



More About

Algorithms

The general recursive prediction error algorithm (11.44) of Ljung (1999) is implemented. See also "Recursive Algorithms for Online Parameter Estimation".

- "What Is Online Estimation?"
- "Recursive Algorithms for Online Parameter Estimation"

See Also

`nkshift` | `recursiveAR` | `recursiveARMA` | `recursiveARMAX` | `recursiveARX` | `recursiveBJ` | `recursiveOE` | `rplr`

Introduced before R2006a

rplr

Estimate general input-output models using recursive pseudolinear regression method

`rplr` is not compatible with MATLAB Coder or MATLAB Compiler.

Syntax

```
thm = rplr(z,nn,adm,adg)
```

```
[thm,yhat,P,phi] = rplr(z,nn,adm,adg,th0,P0,phi0)
```

Description

This is a direct alternative to `rpem` and has essentially the same syntax. See `rpem` for an explanation of the input and output arguments.

`rplr` differs from `rpem` in that it uses another gradient approximation. See Section 11.5 in Ljung (1999). Several of the special cases are well-known algorithms.

When applied to ARMAX models, (`nn = [na nb nc 0 0 nk]`), `rplr` gives the extended least squares (ELS) method. When applied to the output-error structure (`nn = [0 nb 0 0 nf nk]`), the method is known as HARF in the `adm = 'ff'` case and SHARF in the `adm = 'ng'` case.

`rplr` can also be applied to the time-series case in which an ARMA model is estimated with:

```
z = y
nn = [na nc]
```

Examples

Estimate Output-Error Model Parameters Using Recursive Pseudolinear Regression

Specify the order and delays of an Output-Error model structure.

```
na = 0;
nb = 2;
nc = 0;
nd = 0;
nf = 2;
nk = 1;
```

Load the estimation data.

```
load iddata1 z1
```

Estimate the parameters using forgetting factor algorithm, with forgetting factor 0.99.

```
EstimatedParameters = rplr(z1,[na nb nc nd nf nk],'ff',0.99);
```

More About

- “What Is Online Estimation?”
- “Recursive Algorithms for Online Parameter Estimation”

References

For more information about HARF and SHARF, see Chapter 11 in Ljung (1999).

See Also

`nkshift` | `recursiveAR` | `recursiveARMA` | `recursiveARMAX` | `recursiveARX` | `recursiveBJ` | `recursiveOE` | `rpem`

Introduced before R2006a

rsample

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`.

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

Create Random Samples of Estimated Model

Estimate a third-order, discrete-time, state-space model.

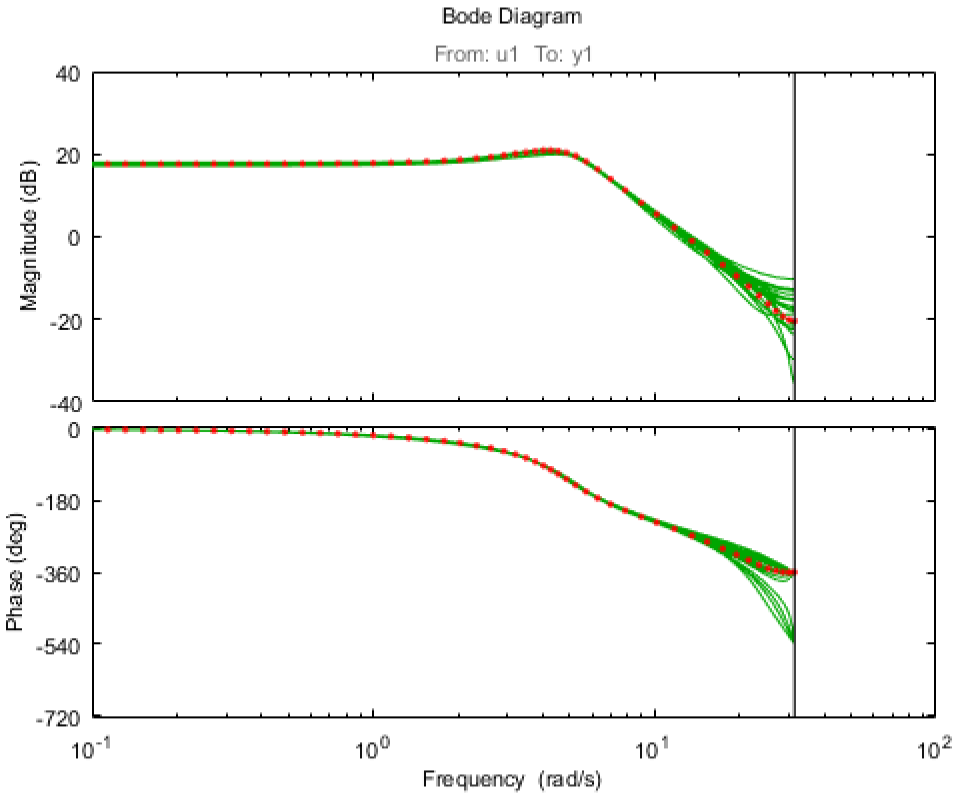
```
load iddata2 z2;
sys = n4sid(z2,3);
```

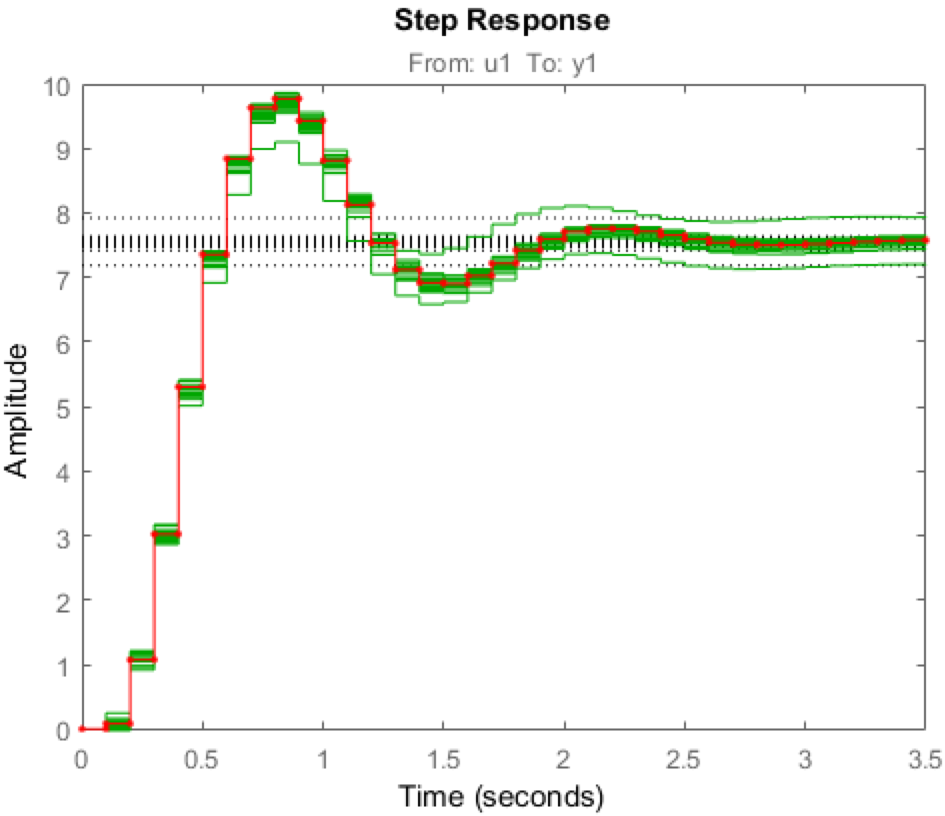
Randomly sample the estimated model.

```
N = 20;
sys_array = rsample(sys,N);
```

Analyze the uncertainty in time (step) and frequency (Bode) responses.

```
opt = bodeoptions;
opt.PhaseMatching = 'on';
figure;
bodeplot(sys_array,'g',sys,'r.',opt)
figure;
stepplot(sys_array,'g',sys,'r.-')
```


Bode Diagram
From: u1 To: y1
Magnitude (dB)
40
20
0
-20
-40
Phase (deg)
0
-180
-360
-540
-720
10^{-1}
10^{0}
10^{1}
10^{2}
Frequency (rad/s)



Specify Standard Deviation Level for Parameter Perturbation

Estimate a third-order, discrete-time, state-space model.

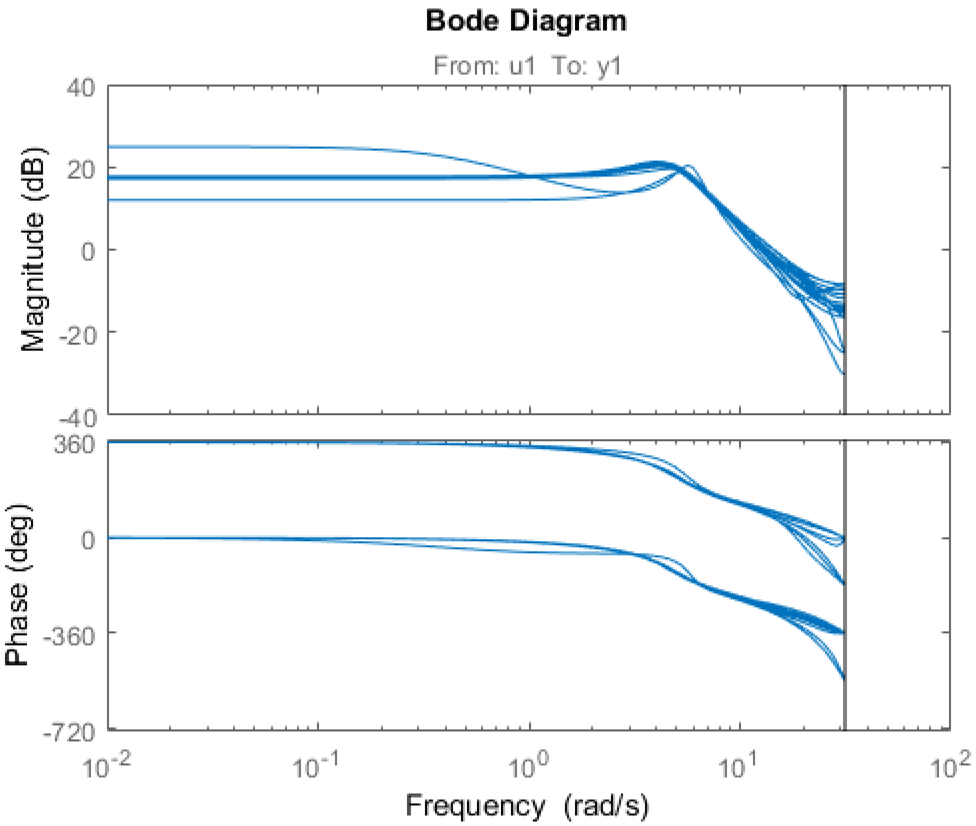
```
load iddata2 z2;
sys = n4sid(z2,3);
```

Randomly sample the estimated model. Specify the standard deviation level for perturbing the model parameters.

```
N = 20;
sd = 2;
sys_array = rsample(sys,N,sd);
```

Analyze the model uncertainty.

```
figure;
bode(sys_array);
```



Compare Frequency Response Confidence Regions for Sampled ARMAX Model

Estimate an ARMAX model.

```
load iddata1 z1
sys = armax(z1,[2 2 2 1]);
```

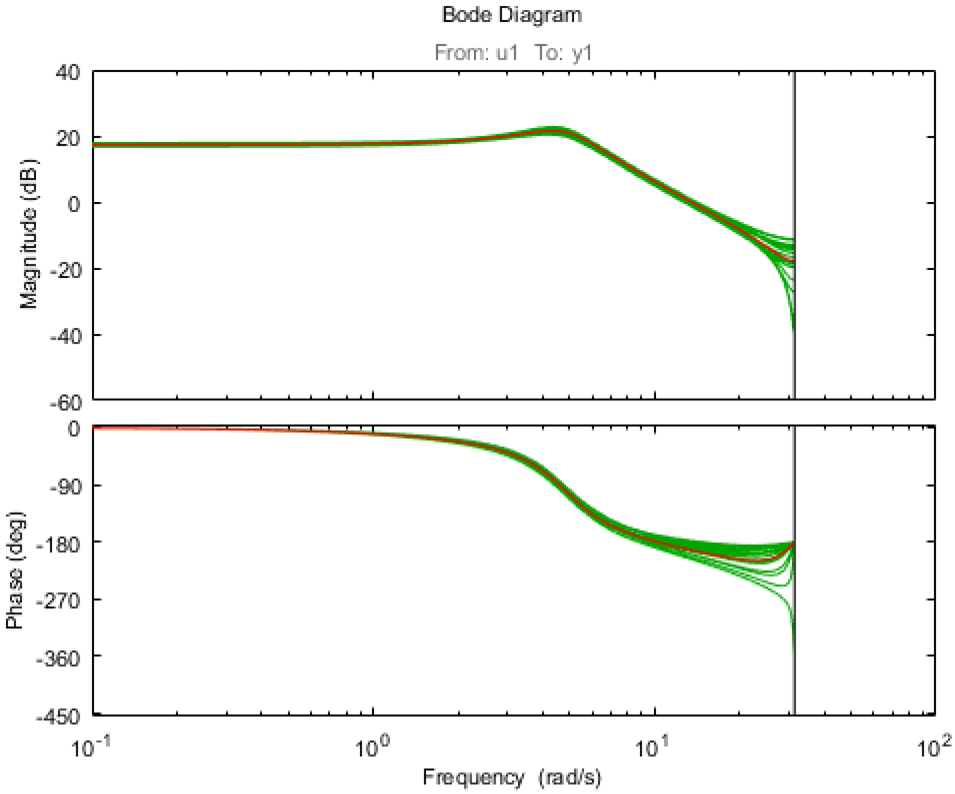
Randomly sample the ARMAX model. Perturb the model parameters up to 2 standard deviations.

```
N = 20;
```

```
sd = 2;
sys_array = rsample(sys,N,sd);
```

Compare the frequency response confidence region corresponding to 2 standard deviations (asymptotic estimate) with the model array response.

```
opt = bodeoptions; opt.PhaseMatching = 'on';
opt.ConfidenceRegionNumberSD = 2;
bodeplot(sys_array,'g',sys,'r',opt)
```



To view the confidence region, right click the plot, and choose **Characteristics** > **Confidence Region**.

More About

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`.

See Also

`bode` | `init` | `iopzmap` | `noise2meas` | `noisecnv` | `simsd` | `step` | `translatecov`

Introduced in R2012a

saturation

Create a saturation nonlinearity estimator object

Syntax

```
NL = saturation
NL = saturation('LinearInterval',[a,b])
```

Description

`NL = saturation` creates a default saturation nonlinearity estimator object for estimating Hammerstein-Wiener models. The linear interval is set to `[NaN NaN]`. The initial value of the linear interval is determined from the estimation data range during estimation using `nlhw`. Use dot notation to customize the object properties, if needed.

`NL = saturation('LinearInterval',[a,b])` creates a saturation nonlinearity estimator object initialized with linear interval, `[a,b]`.

Alternatively, use `NL = saturation([a,b])`.

Object Description

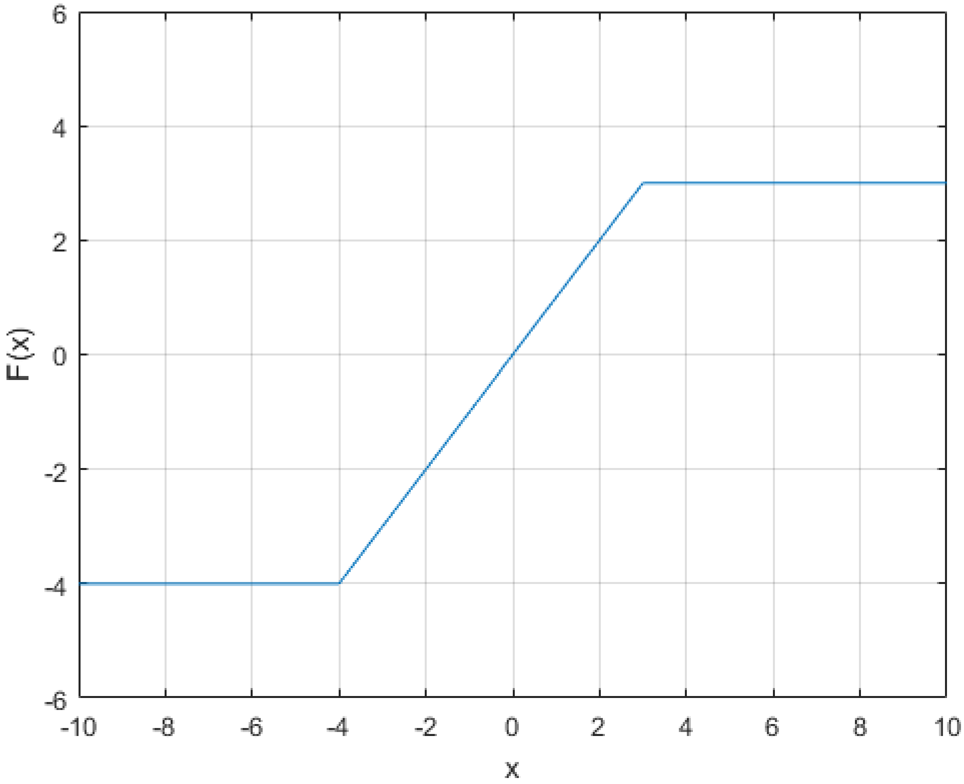
`saturation` is an object that stores the saturation nonlinearity estimator for estimating Hammerstein-Wiener models.

Use `saturation` to define a nonlinear function $y = F(x,\theta)$, where y and x are scalars, and θ represents the parameters a and b that define the linear interval, `[a,b]`.

The saturation nonlinearity function has the following characteristics:

$$\begin{array}{ll} a \le x < b & F(x) = x \\ a > x & F(x) = a \\ b \le x & F(x) = b \end{array}$$

For example, in the following plot, the linear interval is `[-4,3]`.



The value `F(x)` is computed by `evaluate(NL,x)`, where `NL` is the `saturation` object.

For `saturation` object properties, see “Properties” on page 1-1286.

Examples

Create a Default Saturation Nonlinearity Estimator

```
NL = saturation;
```

Specify the linear interval.

```
NL.LinearInterval = [-4,5];
```

Estimate a Hammerstein Model with Saturation

Load data.

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

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 MIMO Hammerstein-Wiener Model

Load the estimation data.

```
load motorizedcamera;
```

Create an `iddata` object.

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

`z` is an `iddata` object with 6 inputs and 2 outputs.

Specify the model orders and delays.

```
Orders = [ones(2,6),ones(2,6),ones(2,6)];
```

Specify the same nonlinearity estimator for each input channel.

```
InputNL = saturation;
```

Specify different nonlinearity estimators for each output channel.

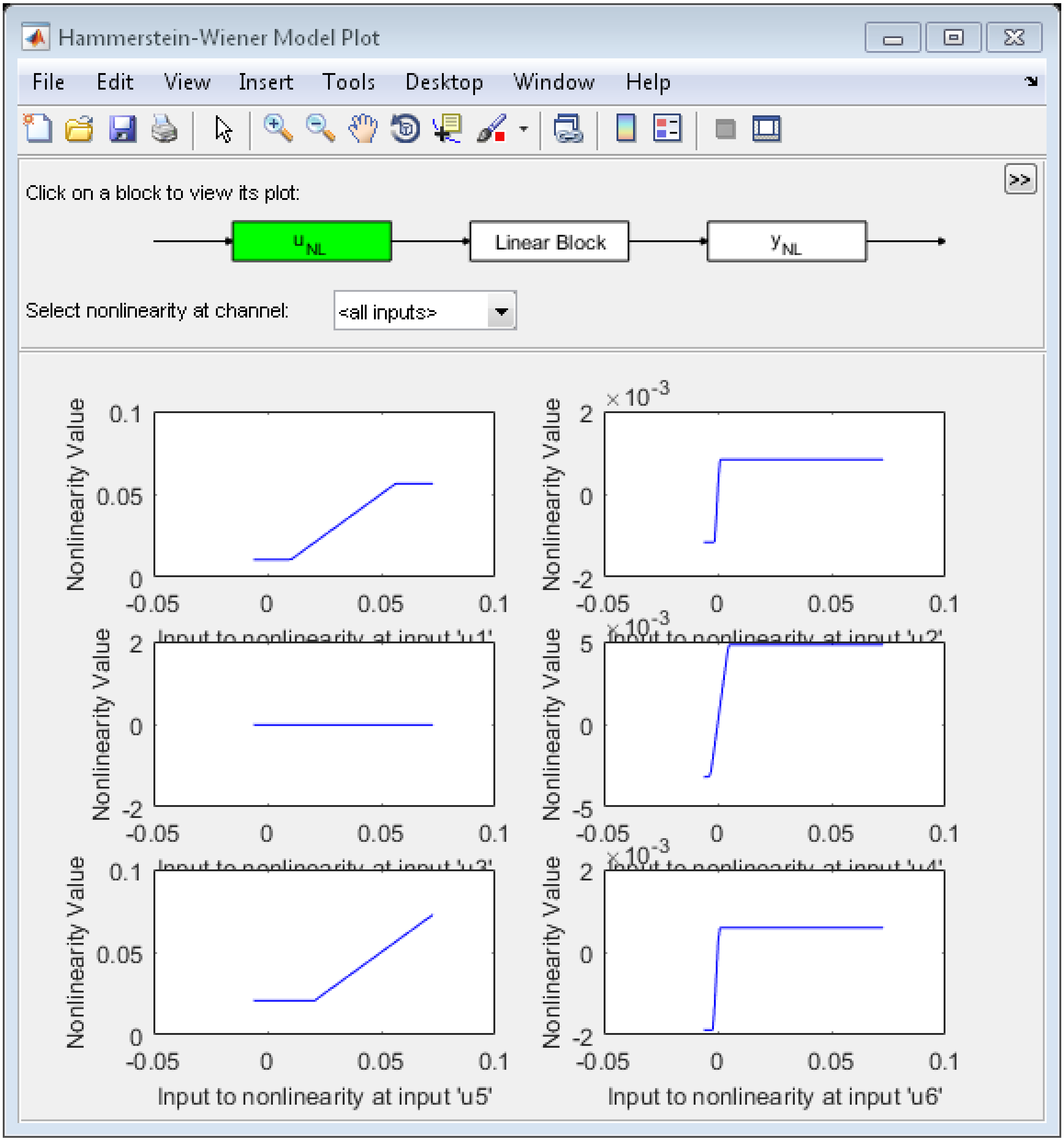
```
 OutputNL = [deadzone,wavenet];
```

Estimate the Hammerstein-Wiener model.

```
sys = nlhw(z,Orders,InputNL,OutputNL);
```

To see the shape of the estimated input and output nonlinearities, plot the nonlinearities.

```
plot(sys)
```



Click on a block to view its plot:

Select nonlinearity at channel: <all inputs>

Click on the input and output nonlinearity blocks on the top of the plot to see the nonlinearites.

Input Arguments

`[a,b]` — Linear interval

`[NaN NaN]` (default) | 2–element row vector

Linear interval of the saturation, specified as a 2–element row vector of doubles.

The saturation nonlinearity is initialized at the interval `[a,b]`. The interval values are adjusted to the estimation data by `nlhw`. To remove the lower limit, set `a` to `-Inf`. The lower limit is not adjusted during estimation. To remove the upper limit, set `b` to `Inf`. The upper limit is not adjusted during estimation.

When the interval is `[NaN NaN]`, the initial value of the linear interval is determined from the estimation data range during estimation using `nlhw`.

Example: `[-2 1]`

Properties

`LinearInterval`

Linear interval of the saturation, specified as a 2–element row vector of doubles.

Default: `[NaN NaN]`

Output Arguments

`NL` — Saturation nonlinearity estimator object

`saturation` object

Saturation nonlinearity estimator object, returned as a `saturation` object.

See Also

`nlhw`

Introduced in R2007a

segment

Segment data and estimate models for each segment

`segment` is not compatible with MATLAB Coder or MATLAB Compiler.

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;
```

The output argument `segm` is a matrix, where the `k`th row contains the parameters corresponding to time `k`. This is analogous to output estimates returned by the `recursiveARX` and `recursiveARMAX` estimators. The output argument `thm` of `segment` contains the corresponding model parameters that have not yet been segmented. Each row of `thm` contains the parameter estimates at the corresponding time instant. These estimates are formed by weighting together the parameters of `M` (default: 5) different time-varying models, with the participating models changing at every time step. Consider `segment` as an alternative to the online estimation commands when you are not interested in continuously tracking the changes in parameters of a single model, but need to detect abrupt changes in the system dynamics.

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.

`ll` is the guaranteed life of each of the models. That is, any created candidate model is not abolished until after at least `ll` time steps. The default is `ll = 1`. `Mu` is a forgetting parameter that is used in the scheme that estimates `R2`. The default is `0.97`.

The most critical parameter for you to choose is `R2`. It is usually more robust to have a reasonable guess of `R2` than to estimate it. Typically, you need to try different values of `R2` and evaluate the results. (See the example below.) `sqrt(R2)` corresponds to a change in the value *y*(*t*) that is normal, giving no indication that the system or the input might have changed.

Examples

Divide Sinusoid into Segments with Constant Levels

Create a sinusoid for the simulated model output.

```
y = sin([1:50]/3)';
```

Specify the input signal to be constant at `1`.

```
u = ones(size(y));
```

Specify the estimated noise variance for the model.

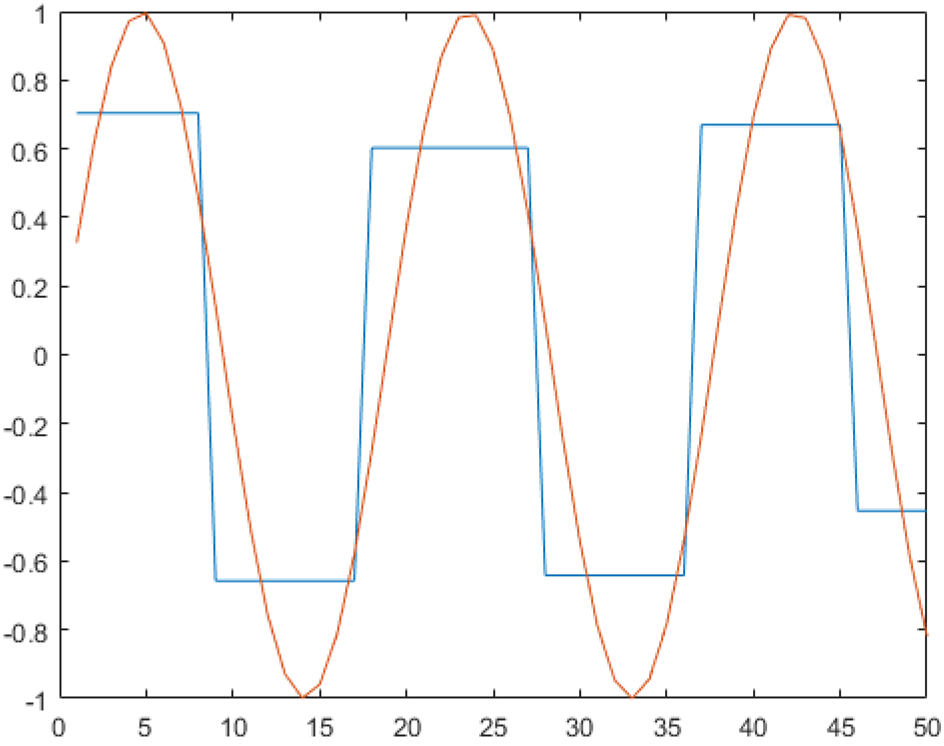
```
R2 = 0.1;
```

Segment the signal and estimate an ARX model for each segment. Use the simple model $y(t) = b_1u(t-1)$, where $b1$ is the model parameter describing the piecewise constant level of the estimated output, $y(t)$.

```
segm = segment([y,u],[0 1 1],R2);
```

Examine the result.

```
plot([segm,y])
```

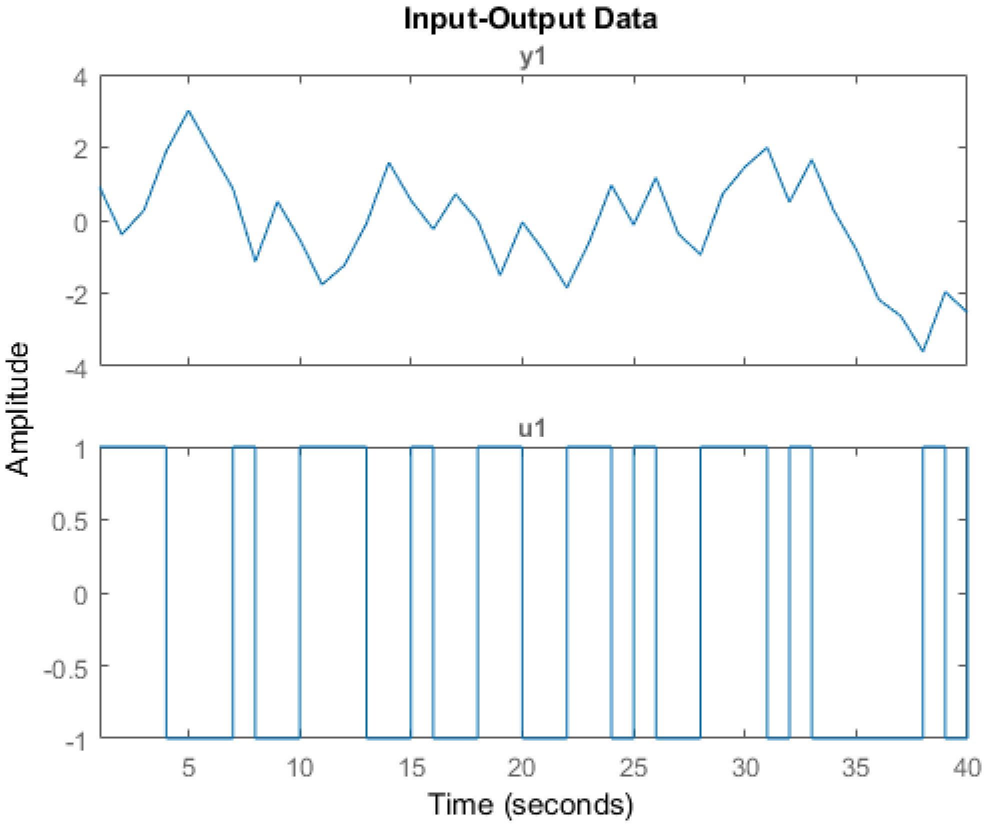


Vary the value of R2 to change the estimated noise variance. Decreasing R2 increases the number of segments produced for this model.

Model Abrupt Change In Time Delay Using Segmentation

Load and plot the estimation data.

```
load iddemo6m.mat z
z = iddata(z(:,1),z(:,2));
plot(z)
```



This data contains a change in time delay from `2` to `1`, which is difficult to detect by examining the data.

Specify the model orders to estimate an ARX model of the form:

$$y(t) + ay(t-1) = b_1u(t-1) + b_2u(t-2)$$

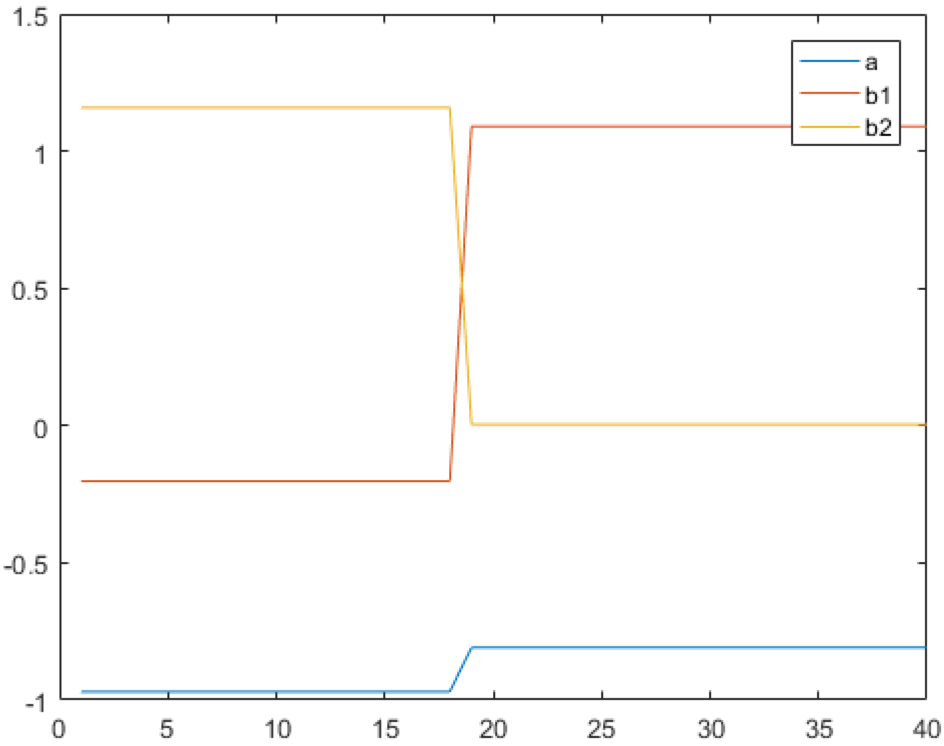
```
nn = [1 2 1];
```

Segment the data and estimate ARX models for each segment. Specify an estimated noise variance of `0.1`.

```
seg = segment(z,nn,0.1);
```

Examine the parameters of the segmented model.

```
plot(seg)
legend('a','b1','b2');
```



The data has been divided into two segments, as indicated by the change in model parameters around sample number 19. The increase in `b1`, along with a corresponding decrease in `b2`, shows the change in model delay.

- "Detect Abrupt System Changes Using Identification Techniques"

More About

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 `ll` 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.

- “Data Segmentation”

Introduced before R2006a

selstruc

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

Generate Model-Order Combinations and Estimate ARX Model Using IV Method

Create estimation and validation data sets

```
load iddata1;
ze = z1(1:150);
zv = z1(151:300);
```

Generate model-order combinations for estimation, specifying ranges for model orders and delays.

```
NN = struc(1:3,1:2,2:4);
```

Estimate ARX models using the instrumental variable method, and compute the loss function for each model order combination.

```
V = ivstruc(ze,zv,NN);
```

Select the model order with the best fit to the validation data.

```
order = selstruc(V,0);
```

Estimate an ARX model of selected order.

```
M = iv4(ze,order);
```

Generate Model-Order Combinations and Estimate Multi-Input ARX Model

Create estimation and validation data sets.

```
load co2data;
Ts = 0.5; % Sample time 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.

```
NN = struc(2:4,2:5,[1 4],1:4,0);
```

Estimate an ARX model for each model order combination.

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

Select the model order with the best fit to the validation data.

```
order = selstruc(V,0);
```

Estimate an ARX model of selected order.

```
M = arx(ze,order);
```

Introduced before R2006a

set

Set or modify model properties

Syntax

```
set(sys,'Property',Value)
set(sys,'Property1',Value1,'Property2',Value2,...)
sysnew = set(___)
set(sys,'Property')
```

Description

`set` is used to set or modify the properties of a dynamic system model using property name/property value pairs.

`set(sys,'Property',Value)` assigns the value `Value` to the property of the model `sys`. `'Property'` 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. This syntax is equivalent to `sys.Property = Value`.

`set(sys,'Property1',Value1,'Property2',Value2,...)` sets multiple property values with a single statement. Each property name/property value pair updates one particular property.

`sysnew = set(___)` returns the modified dynamic system model, and can be used with any of the previous syntaxes.

`set(sys,'Property')` displays help for the property specified by `'Property'`.

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
```

More About

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:

$$\bar{h}\left(z^{-1}\right) = \frac{1+2z^{-1}}{1+3z^{-1}+4z^{-2}} = zh\left(z\right)$$

Note Because the resulting transfer functions are different, make sure to use the convention consistent with your choice of variable.

See Also

`idfrd` | `idtf` | `idproc` | `idnlarx` | `idnlgrey` | `get` | `frd` | `ss` | `tf` | `zpk` | `idss` | `idgrey` | `idpoly` | `idnlhw`

Introduced before R2006a

setcov

Set parameter covariance data in identified model

Syntax

```
sys = setcov(sys0,cov)
```

Description

`sys = setcov(sys0,cov)` modifies the parameter covariance of `sys0` to the value specified by `cov`.

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.

Input Arguments

`sys0`

Identified model.

`cov`

Parameter covariance matrix.

`cov` is one of the following:

- an *np*-by-*np* semi-positive definite symmetric matrix, where *np* is equal to the number of parameters of `sys0`.
- a structure with the following fields that describe the parameter covariance in a factored form:
  - `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 `sys0`.

```
cov(Free,Free) = T*inv(R'*R)*T'.
```

Default:

Output Arguments

`sys`

Identified model.

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

Set Raw Covariance Data for Identified Model

Create a transfer function model for the following system:

$$sys0 = \frac{4}{s^2 + 2s + 1}$$

```
sys0 = idtf(4,[1 2 1]);
np = nparams(sys0);
```

`sys0` contains `np` model parameters.

Specify the covariance values for the denominator parameters only.

```
cov = zeros(np);
den_index = 2:3;
cov(den_index,den_index) = diag([0.04 0.001]);
```

`cov` is a covariance matrix with nonzero entries for the denominator parameters.

Set the covariance for `sys0`.

```
sys = setcov(sys0,cov);
```

See Also

`getcov` | `rsample` | `setpvec` | `sim`

Introduced in R2012a

setinit

Set initial states of `idnlgrey` model object

Syntax

```
model = setinit(model,Property,Values)
```

Description

`model = setinit(model,Property,Values)` sets the `values` of the `Property` field of the `InitialStates` model property. `Property` can be `'Name'`, `'Unit'`, `'Value'`, `'Minimum'`, `'Maximum'`, and `'Fixed'`.

Input Arguments

`model`

Name of the `idnlgrey` model object.

`Property`

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

`Values`

Values of the specified property `Property`. `Values` are an `Nx`-by-1 cell array of values, where `Nx` is the number of states.

See Also

`getinit` | `getpar` | `idnlgrey` | `setpar`

Introduced in R2007a

setoptions

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
```

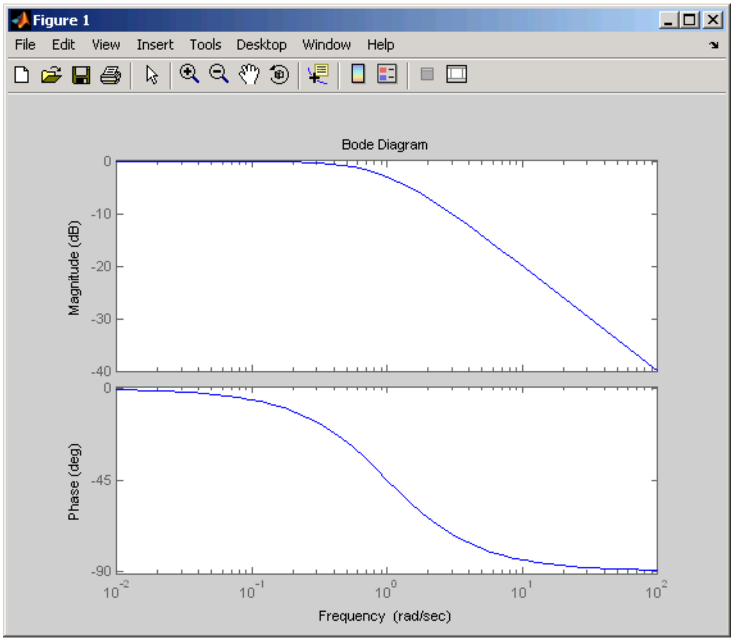
For a list of the properties and values available for each plot type, see “Properties and Values Reference”.

`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

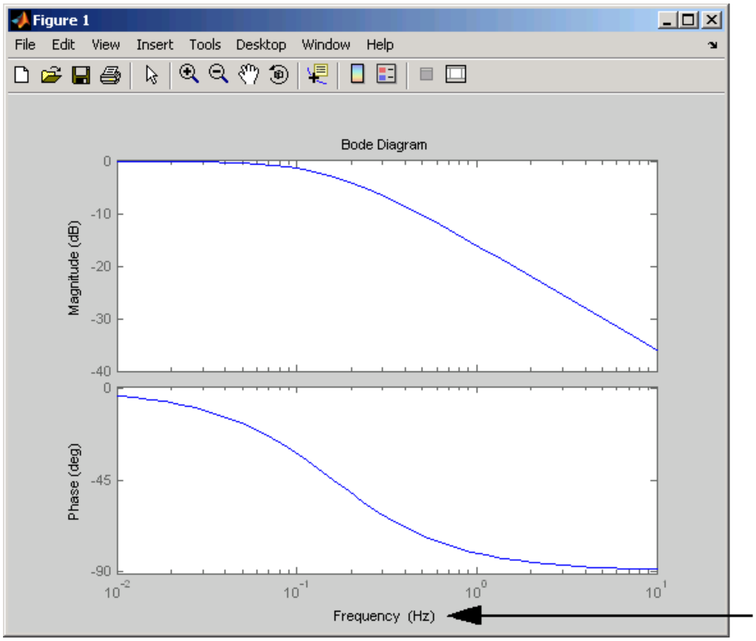
To change frequency units, first create a Bode plot.

```
sys=tf(1,[1 1]);
h=bodeplot(sys)     % Create a Bode plot with plot handle h.
```



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.
```



The frequency units are now Hz.

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`

Introduced in R2012a

idParametric/setpar

Set attributes such as values and bounds of linear model parameters

Syntax

```
sys1 = setpar(sys,'value',value)
sys1 = setpar(sys,'free',free)
sys1 = setpar(sys,'bounds',bounds)
sys1 = setpar(sys,'label',label)
```

Description

`sys1 = setpar(sys,'value',value)` sets the parameter values of the model `sys`. For model arrays, use `setpar` separately on each model in the array.

`sys1 = setpar(sys,'free',free)` sets the free or fixed status of the parameters.

`sys1 = setpar(sys,'bounds',bounds)` sets the minimum and maximum bounds on the parameters.

`sys1 = setpar(sys,'label',label)` sets the labels for the parameters.

Examples

Assign Model Parameter Values

Estimate an ARMAX model.

```
load iddata8;
init_data = z8(1:100);
na = 1;
nb = [1 1 1];
nc = 1;
nk = [0 0 0];
sys = armax(init_data,[na nb nc nk]);
```

Set the parameter values.

```
sys = setpar(sys,'value',[0.5 0.1 0.3 0.02 0.5]');
```

To view the values, type `val = getpar(sys,'value')`.

Fix or Free Model Parameters

Construct a process model.

```
m = idproc('P2DUZI');
m.Kp = 1;
m.Tw = 100;
m.Zeta = .3;
m.Tz = 10;
m.Td = 0.4;
```

Set the free status of the parameters.

```
m = setpar(m,'free',[1 1 1 1 0]);
```

Here, you set `Tz` to be a fixed parameter.

To check the free status of `Tz`, type `m.Structure.Tz`.

Set Minimum and Maximum Bounds on Parameters

Estimate an ARMAX model.

```
load iddata8;
init_data = z8(1:100);
na = 1;
nb = [1 1 1];
nc = 1;
nk = [0 0 0];
sys = armax(init_data,[na nb nc nk]);
```

Set the minimum and maximum bounds for the parameters. Each row represents the bounds for a single parameter. The first value in each row specifies the minimum bound and the second value specifies the maximum bound.

```
sys = setpar(sys,'bounds',[0 1; 1 1.5; 0 2; 0.5 1; 0 1]);
```

Assign Default Labels to Parameters

Estimate an ARMAX model.

```
load iddata8;
init_data = z8(1:100);
na = 1;
nb = [1 1 1];
nc = 1;
nk = [0 0 0];
sys = armax(init_data,[na nb nc nk]);
```

Assign default labels to model parameters.

```
sys = setpar(sys,'label','default');
```

View the default labels.

```
getpar(sys,'label')
```

```
ans =

  5×1 cell array

    'A1(1)'
    'B0(1)'
    'B0(2)'
    'B0(3)'
    'C1'
```

Input Arguments

`sys` — Identified linear model

`idss` | `idproc` | `idgrey` | `idtf` | `idpoly`

Identified linear model, specified as an `idss`, `idproc`, `idgrey`, `idtf`, or `idpoly` model object.

`value` — Parameter values

vector of doubles

Parameter values, specified as a double vector of length `nparams(sys)`.

`free` — Free or fixed status of parameters

vector of logical values

Free or fixed status of parameters, specified as a logical vector of length `nparams(sys)`.

`bounds` — Minimum and maximum bounds on parameters
matrix of doubles

Minimum and maximum bounds on parameters, specified as a double matrix of size `nparams(sys)`-by-2. The first column specifies the minimum bound and the second column the maximum bound.

`label` — Parameter labels
cell array of character vectors | `'default'`

Parameter labels, specified as a cell array of character vectors. The cell array is of length `nparams(sys)`. For example, `{'a1','a3'}`, if `nparams(sys)` is two.

Use `'default'` to assign default labels, `A1, A2..., B1,B2,...`, to the parameters.

Output Arguments

`sys1` — Model with specified values of parameter attributes
`idss` | `idproc` | `idgrey` | `idtf` | `idpoly`

Model with specified values of parameter attributes. The model `sys` you specify as the input to `setpar` gets updated with the specified parameter attribute values.

See Also

`getpar` | `setcov` | `setpvec`

Introduced in R2013b

setpar

Set initial parameter values of `idnlgrey` model object

Syntax

```
setpar(model,property,values)
```

Input Arguments

`model`

Name of the `idnlgrey` model object.

`property`

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

Default: `'Value'`.

`values`

Values of the specified property `Property`. `values` are an `Np`-by-1 cell array of values, where `Np` is the number of parameters.

Description

`setpar(model,property,values)` sets the model parameter values in the `property` field of the `Parameters` model property. `property` can be `'Name'`, `'Unit'`, `'Value'`, `'Minimum'`, and `'Maximum'`.

See Also

`getinit` | `getpar` | `idnlgrey` | `setinit`

Introduced in R2007a

setPolyFormat

Specify format for *B* and *F* polynomials of multi-input polynomial model

Syntax

```
modelOut = setPolyFormat(modelIn,'double')
modelOut = setPolyFormat(modelIn,'cell')
```

Description

`modelOut = setPolyFormat(modelIn,'double')` converts the `B` and `F` polynomials of a multi-input polynomial model, `modelIn`, to double matrices.

By default, the `B` and `F` polynomials of an `idpoly` model are cell arrays. For MATLAB scripts written before R2012a, convert the cell arrays to double matrices for backward compatibility using this syntax. For example:

```
model = arx(data,[3 2 2 1 1]);
model = setPolyFormat(model,'double');
```

`modelOut = setPolyFormat(modelIn,'cell')` converts the `B` and `F` polynomials of `modelIn` to cell arrays.

MATLAB data files saved before R2012a store `idpoly` models with their `B` and `F` polynomials represented as double matrices. If these models were previously set to operate in backward-compatibility mode, they are not converted to use cell arrays when loaded. Convert these models to use cell arrays using this syntax. For example:

```
load polyData.mat model;
model = setPolyFormat(model,'cell');
```

Examples

Convert *B* and *F* Polynomials of a Multi-Input ARX Model to Double Matrices

Load estimation data.

```
load iddata8;
```

Estimate the model.

```
m1 = arx(z8,[3 [2 2 1] [1 1 1]]);
```

Convert the `b` and `f` polynomials to use double matrices.

```
m2 = setPolyFormat(m1,'double');
```

Extract pole and zero information from the model using matrix syntax.

```
Poles1 = roots(m2.F(1,:));
Zeros1 = roots(m2.B(1,:));
```

- “Extracting Numerical Model Data”

Input Arguments

`modelIn` — Polynomial model

`idpoly` object

Polynomial model, specified as an `idpoly` object. The `B` and `F` polynomials of `modelIn` are either:

- Cell arrays with N_u elements, where N_u is the number of model inputs, with each element containing a double vector. This configuration is the default.
- Double matrices with N_u rows. This configuration applies to backward-compatible `idpoly` models stored in MATLAB data files before R2012a.

Note: `setPolyFormat` only supports multi-input, single-output models. Specifying `modelIn` as a:

- Multi-output model generates an error.
- Single-input, single-output model has no effect. The `B` and `F` polynomials remain as double vectors.

Output Arguments

`modelOut` — Polynomial model

`idpoly` object

Polynomial model, returned as an `idpoly` object.

To access the `b` and `f` polynomials of `modelOut`, use:

- Matrix syntax after using `modelOut = setPolyFormat(modelIn,'double')`. For example:

  ```
  modelOut.B(1,:);
  ```
- Cell array syntax after using `modelOut = setPolyFormat(modelIn,'cell')`. For example:

  ```
  modelOut.B{1};
  ```

After using `modelOut = setPolyFormat(modelIn,'cell')`, you can resave the converted model in cell array format. For example:

```
save polyNew.mat modelOut;
```

More About

Tips

- To verify the current format of the `B` and `F` polynomials for a given `idpoly` model, enter:

  ```
  class(model.B)
  ```

  If the model uses double matrices, the displayed result is:

  ```
  ans =

  double
  ```

  Otherwise, for cell arrays, the result is:

  ```
  ans =

  cell
  ```

See Also

`get` | `idpoly` | `polydata` | `set` | `tfdata`

Introduced in R2010a

setpvec

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`.

Default:

`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 Parameter Values of Transfer Function Model

Construct a transfer function model.

```
sys0 = idtf(1,[1 2]);
```

Define a parameter vector and use it to set the model parameters. The second parameter is set to `NaN`, indicating that its value is unknown.

```
par = [1;NaN;0];
sys = setpvec(sys0,par);
```

The constructed model, `sys`, can be used to initialize a model estimation.

Modify Free Parameter Values of Transfer Function Model

Construct a transfer function model.

```
sys0 = idtf([1 0],[1 2 0]);
```

Set the first three parameters of `sys0` as free parameters.

```
sys0 = setpar(sys0,'free',[1 1 1 0 0]);
```

Define a parameter vector and use it to set the free model parameters.

```
par = [1;2;1];
sys = setpvec(sys0,par,'free');
```

See Also

`getpvec` | `nparams` | `setcov`

Introduced in R2012a

sgrid

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

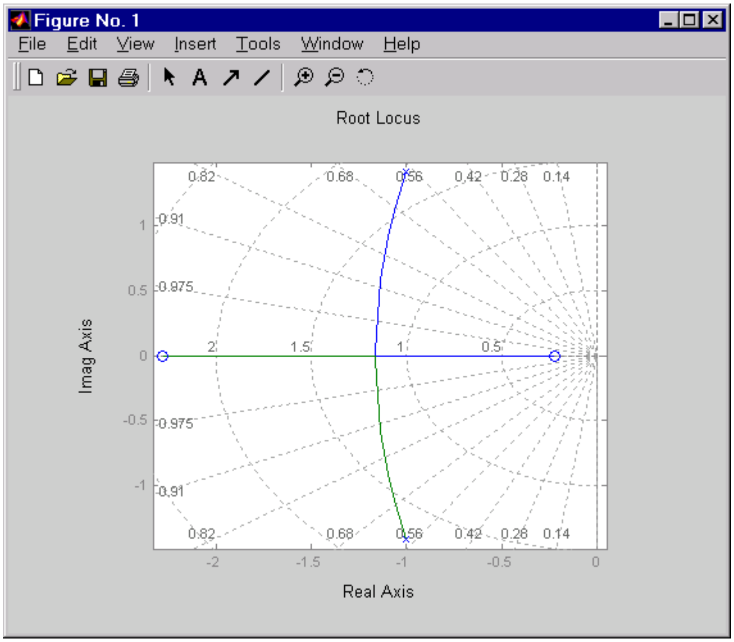
Plot *s*-plane grid lines on the root locus for the following system.

$$H(s) = \frac{2s^2 + 5s + 1}{s^2 + 2s + 3}$$

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`

Introduced in R2012a

showConfidence

Display confidence regions on response plots for identified models

Syntax

```
showConfidence(plot_handle)
showConfidence(plot_handle,sd)
```

Description

`showConfidence(plot_handle)` displays the confidence region on the response plot, with handle `plot_handle`, for an identified model.

`showConfidence(plot_handle,sd)` displays the confidence region for `sd` 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`, `impulseplot`, `nyquistplot`, or `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')`

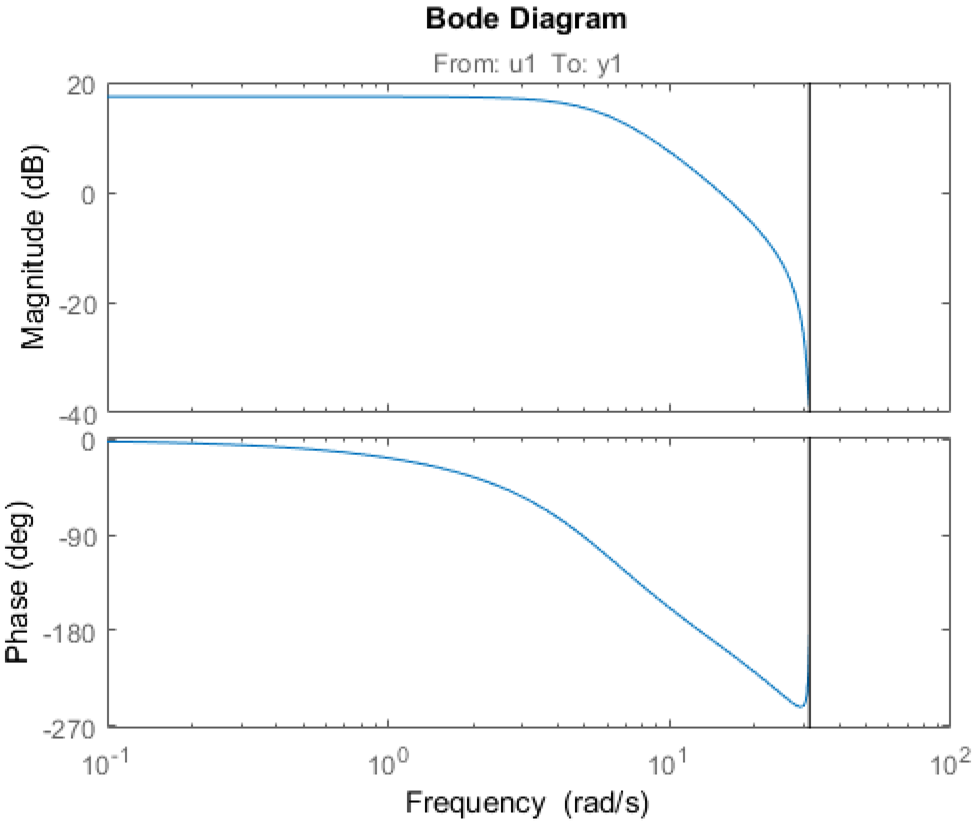
Examples

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.

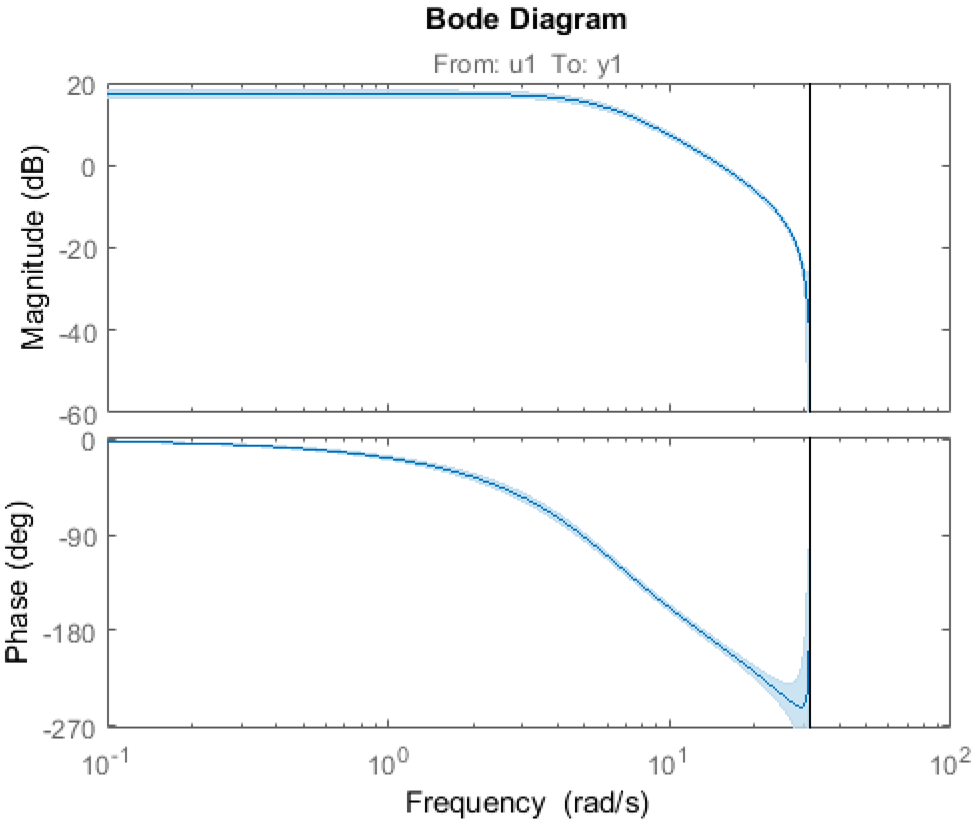
```
load iddata1 z1
sys = arx(z1, [2 2 1]);
h = bodeplot(sys);
```



`z1` is an `iddata` object that contains time domain system response data. `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);
```



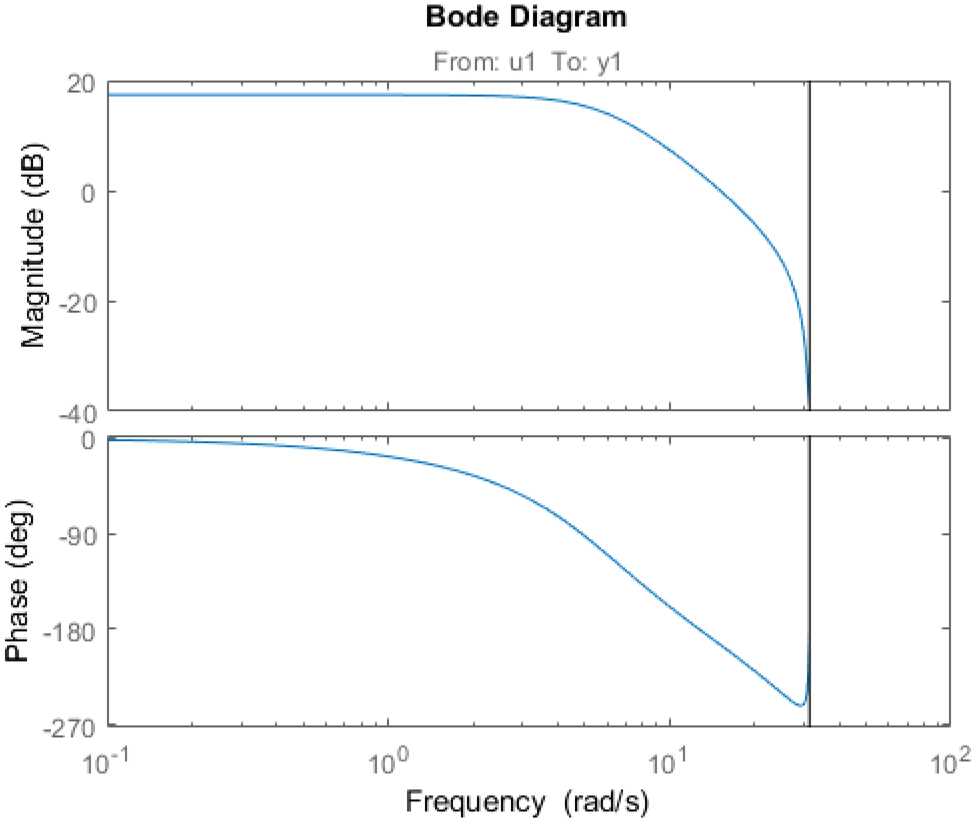
This plot depicts the confidence region for 1 standard deviation.

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);
```

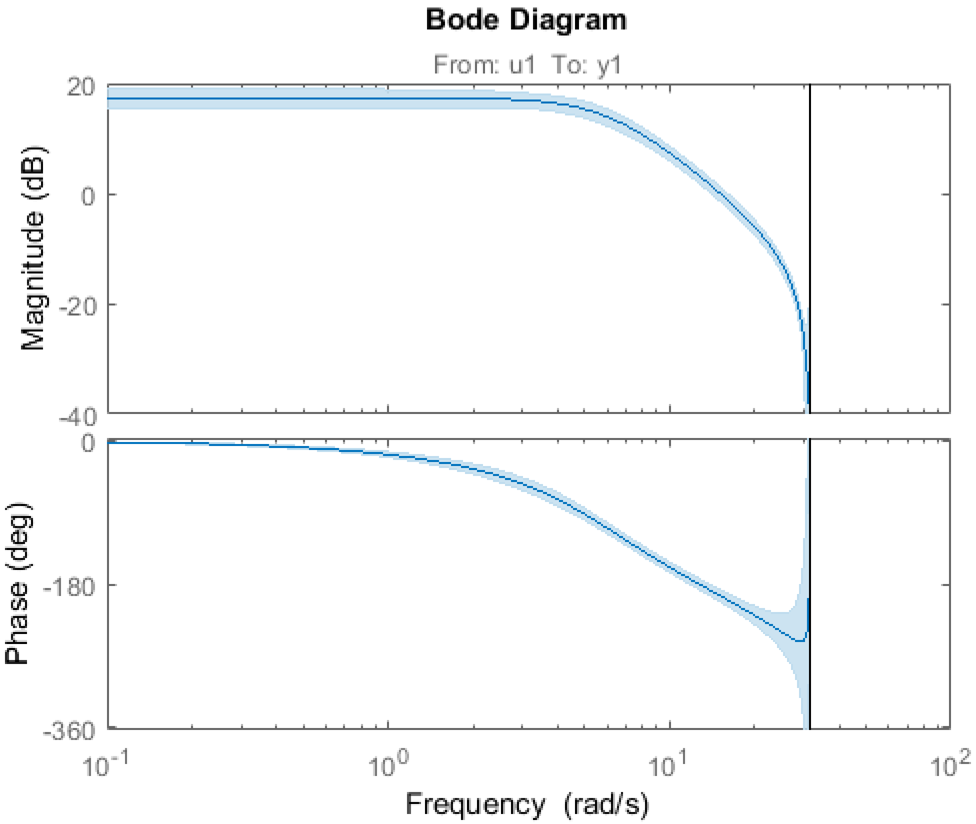


`z1` is an `iddata` object that contains time domain system response data. `sys` is an `idpoly` model containing the identified polynomial model. `h` is the plot handle for the bode response plot of `sys`.

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 can interactively turn on the confidence region display on a response plot. Right-click the response plot, and select **Characteristics** > **Confidence Region**.

See Also

`bodeplot` | `impulseplot` | `iopzplot` | `nyquistplot` | `stepplot`

Introduced in R2012a

sigmoidnet

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 1-1328.

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

sigmoidnet Properties

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

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

```
% List all property values
get(s)
```

```
% Get value of NumberOfUnits property
s.NumberOfUnits
```

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

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

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

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

Examples

Use `sigmoidnet` to specify the nonlinear estimator in nonlinear ARX and Hammerstein-Wiener models. For example:

```
m=nlarx(Data,Orders,sigmoidnet('num',5));
```

More About

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:

$$\begin{aligned} F(x) = (x-r)PL + a_1 f\big((x-r)Qb_1 + c_1\big) + \ldots \\ + a_n f\big((x-r)Qb_n + c_n\big) + d \end{aligned}$$

where f is the sigmoid function, given by the following equation:

$$f(z) = \frac{1}{e^{-z}+1}.$$

P and Q are `m`-*by*-`p` and `m`-*by*-`q` projection matrices. The projection matrices P and Q are determined by principal component analysis of estimation data. Usually, `p=m`. If the components of x in the estimation data are linearly dependent, then `p<m`. The number of columns of Q, `q`, corresponds to the number of components of `x` used in the sigmoid function.

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

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

d, a, and c are scalars.

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

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

Algorithms

`sigmoidnet` uses an iterative search technique for estimating parameters.

See Also

`nlarx` | `nlhw`

Introduced in R2007a

sim

Simulate response of identified model

Syntax

```
y = sim(sys,udata)
y = sim(sys,udata,opt)

[y,y_sd] = sim( ___ )
[y,y_sd,x] = sim( ___ )
[y,y_sd,x,x_sd] = sim( ___ )

sim( ___ )
```

Description

`y = sim(sys,udata)` returns the simulated response of an identified model using the input data, `udata`. By default, zero initial conditions are used for all model types except `idnlgrey`, in which case the initial conditions stored internally in the model are used.

`y = sim(sys,udata,opt)` uses the option set, `opt`, to configure the simulation option, including the specification of initial conditions.

`[y,y_sd] = sim( ___ )` returns the estimated standard deviation, `y_sd`, of the simulated response.

`[y,y_sd,x] = sim( ___ )` returns the state trajectory, `x`, for state-space models.

`[y,y_sd,x,x_sd] = sim( ___ )` returns the standard deviation of the state trajectory, `x_sd`, for state-space models.

`sim( ___ )` plots the simulated response of the identified model.

Examples

Simulate State-Space Model Using Input Data

Load the estimation data.

```
load iddata2 z2
```

Estimate a third-order state-space model.

```
sys = ssest(z2,3);
```

Simulate the identified model using the input channels from the estimation data.

```
y = sim(sys,z2);
```

Add Noise to Simulated Model Response

Load the data, and obtain the identified model.

```
load iddata2 z2
sys = n4sid(z2,3);
```

`sys` is a third-order state-sapce model estimated using a subspace method.

Create a simulation option set to add noise to the simulated model response.

```
opt1 = simOptions('AddNoise',true);
```

Simulate the model.

```
y = sim(sys,z2,opt1);
```

Default Gaussian white noise is filtered by the noise transfer function of the model and added to the simulated model response.

You can also add your own noise signal, `e`, using the `NoiseData` option.

```
e = randn(length(z2.u),1);
opt2 = simOptions('AddNoise',true,'NoiseData',e);
```

Simulate the model.

```
y = sim(sys,z2,opt2);
```

Simulate Model Using Initial Conditions Obtained During Estimation

Load data.

```
load iddata1 z1
```

Specify the estimation option to estimate the initial state.

```
estimOpt = ssestOptions('InitialState','estimate');
```

Estimate a state-space model, and return the value of the estimated initial state.

```
[sys,x0] = ssest(z1,2,estimOpt);
```

Specify initial conditions for simulation

```
simOpt = simOptions('InitialCondition',x0);
```

Simulate the model, and obtain the model response and standard deviation.

```
[y,y_sd] = sim(sys,z1,simOpt);
```

Estimate Standard Deviation and State Trajectory for State-Space Models

Load estimation data, and estimate a state-space model.

```
load iddata1 z1
sys = ssest(z1,2);
```

Return the standard deviation and state trajectory.

```
[y,y_sd,x] = sim(sys,z1);
```

Estimate State Trajectory and Standard Deviations of Simulated Response

Load estimation data, and estimate a state-space model.

```
load iddata1 z1
sys = ssest(z1,2);
```

Create a simulation option set, and specify the initial states.

```
opt = simOptions('InitialCondition',[1;2]);
```

Specify the covariance of the initial states.

```
opt.X0Covariance = [0.1 0; 0 0.1];
```

Calculate the standard deviations of simulated response, `y_sd`, and state trajectory, `x_sd`.

```
[y,y_sd,x,x_sd] = sim(sys,z1,opt);
```

Plot Simulated Model Response

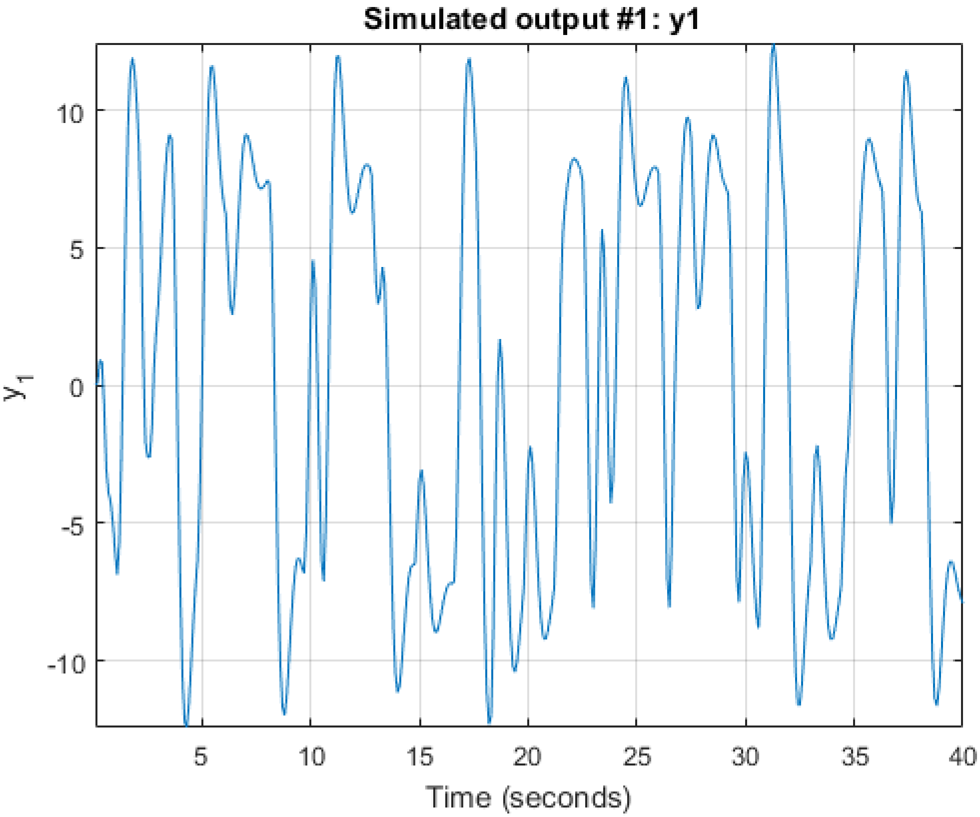
Obtain the identified model.

```
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`.

Simulate the model.

```
sim(sys,z2)
```



Simulate Nonlinear ARX Model

Simulate a single-input single-output nonlinear ARX model around a known equilibrium point, with an input level of `1` and output level of `10`.

Load the sample data.

```
load iddata2
```

Estimate a nonlinear ARX model from the data.

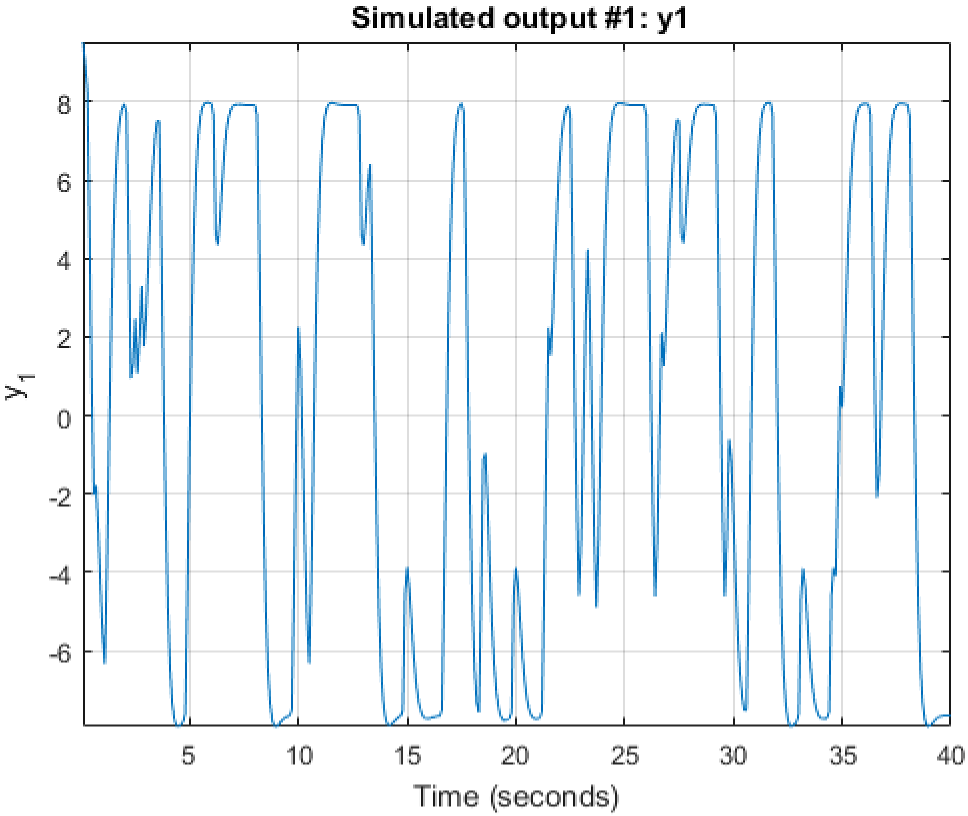
```
M = nlarx(z2,[2 2 1],'treepartition');
```

Estimate current states of model based on past data.

```
x0 = data2state(M,struct('Input',1,'Output',10));
```

Simulate the model using the initial states returned by `data2state`.

```
opt = simOptions('InitialCondition',x0);
sim(M,z2,opt)
```



Continue from End of Previous Simulation

Continue the simulation of a nonlinear ARX model from the end of a previous simulation run.

Estimate a nonlinear ARX model from data.

```
load iddata2
M = nlarx(z2,[2 2 1],'treepartition');
```

Simulate the model using the first half of the input data `z2`. Start the simulation from zero initial states.

```
u1 = z2(1:200,[]);
```

```
opt1 = simOptions('InitialCondition','zero');
ys1 = sim(M,u1,opt1);
```

Start another simulation using the second half of the input data `z2`. Use the same states of the model from the end of the first simulation.

```
u2 = z2(201:end,[]);
```

To set the initial states for the second simulation correctly, package input `u1` and output `ys1` from the first simulation into one `iddata` object. Pass this data as initial conditions for the next simulation.

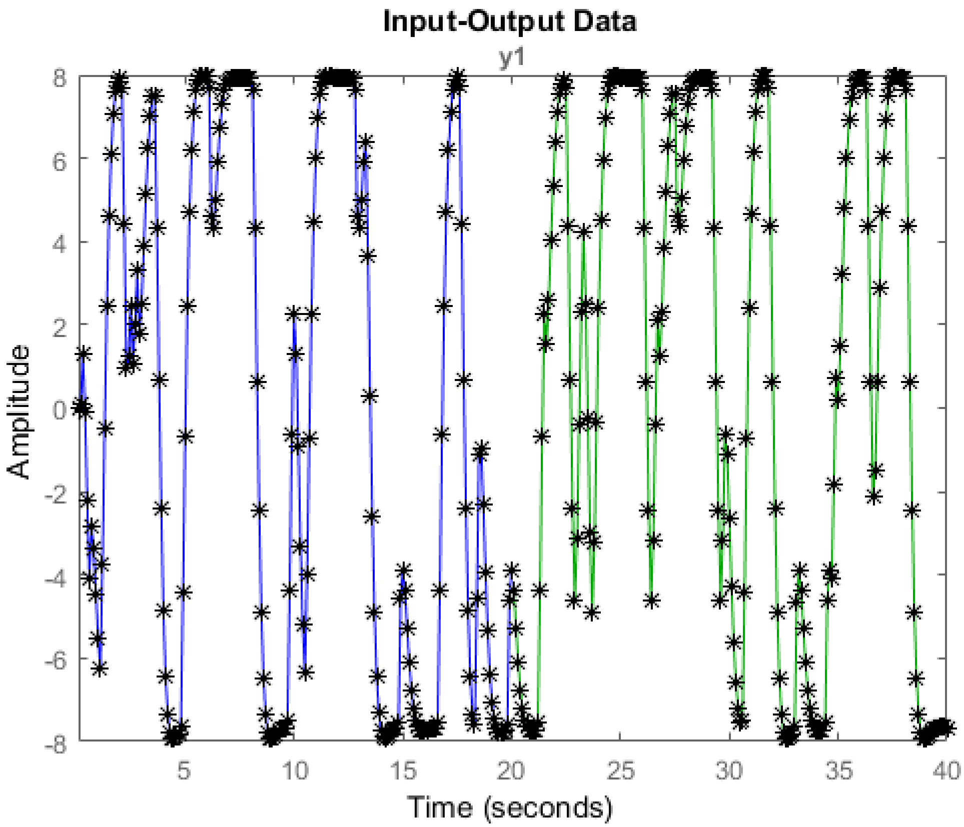
```
firstSimData = [ys1,u1];
opt2 = simOptions('InitialCondition',firstSimData);
ys2 = sim(M,u2,opt2);
```

Verify the two simulations by comparing to a complete simulation using all the input data `z2`. First, extract the whole set of input data.

```
uTotal = z2(:,[]);
opt3 = simOptions('InitialCondition','zero');
ysTotal = sim(M,uTotal,opt3);
```

Plot the three responses `ys1`, `ys2` and `ysTotal.` `ys1` should be equal to first half of `ysTotal`. `ys2` should be equal to the second half of `ysTotal`.

```
plot(ys1,'b',ys2,'g',ysTotal,'k*')
```



The plot shows that the three responses `ys1`, `ys2`, and `ysTotal` overlap as expected.

Match Model Response to Output Data

Estimate initial states of model `M` such that, the response best matches the output in data set `z2`.

Load the sample data.

```
load iddata2;
```

Estimate a nonlinear ARX model from the data.

```
M = nlarx(z2,[4 3 2],wavenet('NumberOfUnits',20));
```

Estimate the initial states of `M` to best fit `z2.y` in the simulated response.

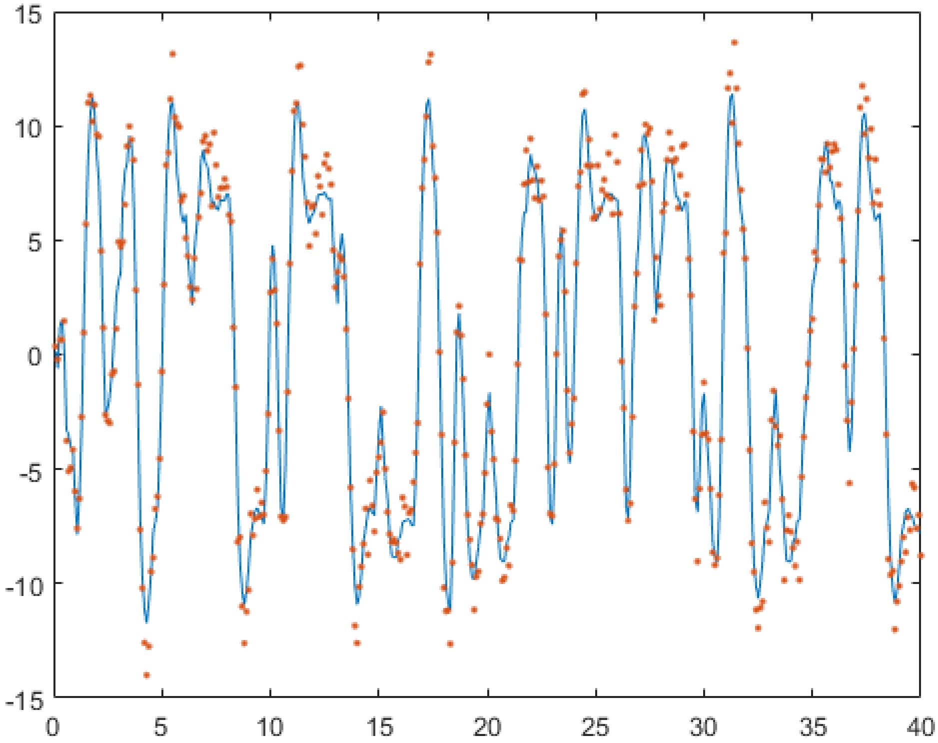
```
x0 = findstates(M,z2,Inf);
```

Simulate the model.

```
opt = simOptions('InitialCondition',x0);
ysim = sim(M,z2.u,opt);
```

Compare the simulated model output `ysim` with the output signal in `z2`.

```
time = z2.SamplingInstants;
plot(time,ysim,time,z2.y,'.')
```



Simulate Model Near Steady State with Known Input and Unknown Output

Start simulation of a model near steady state, where the input is known to be `1`, but the output is unknown.

Load the sample data.

```
load iddata2
```

Estimate a nonlinear ARX model from the data.

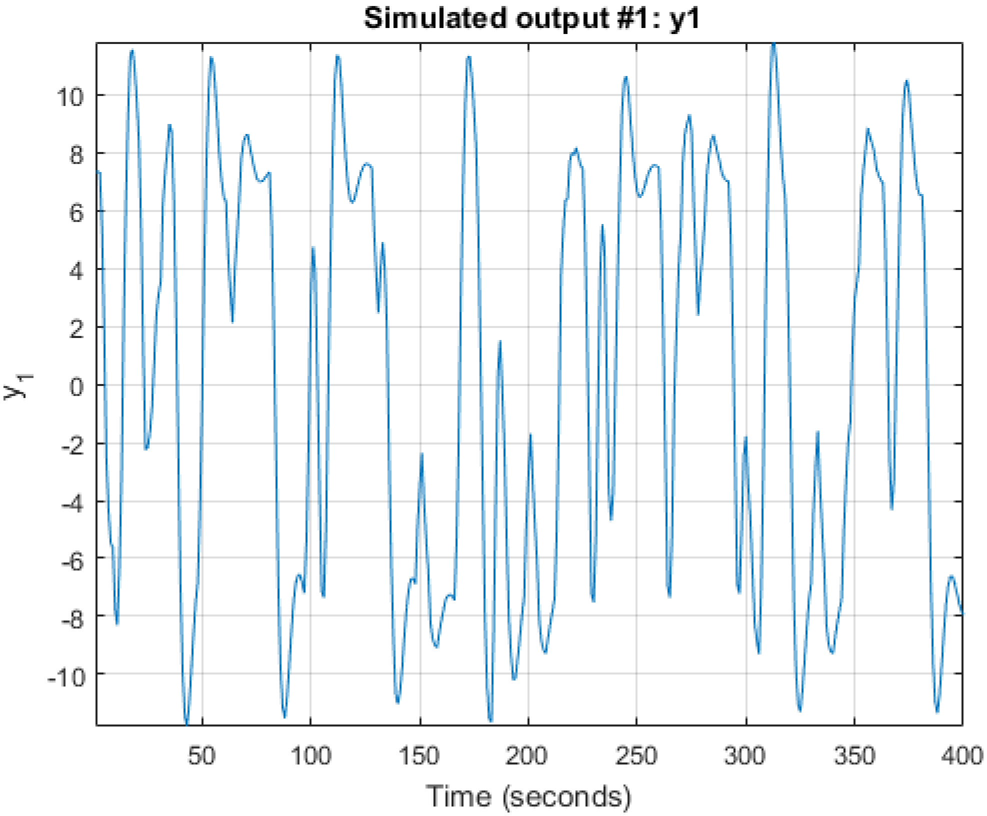
```
M = nlarx(z2,[4 3 2],'wavenet');
```

Determine equilibrium state values for input `1` and unknown target output.

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

Simulate the model using initial states `x0`.

```
opt = simOptions('InitialCondition',x0);
sim(M,z2.u,opt)
```



Simulate Hammerstein-Wiener Model at Steady-State Operating Point

Load the sample data.

```
load iddata2
```

Create a Hammerstein-Wiener model.

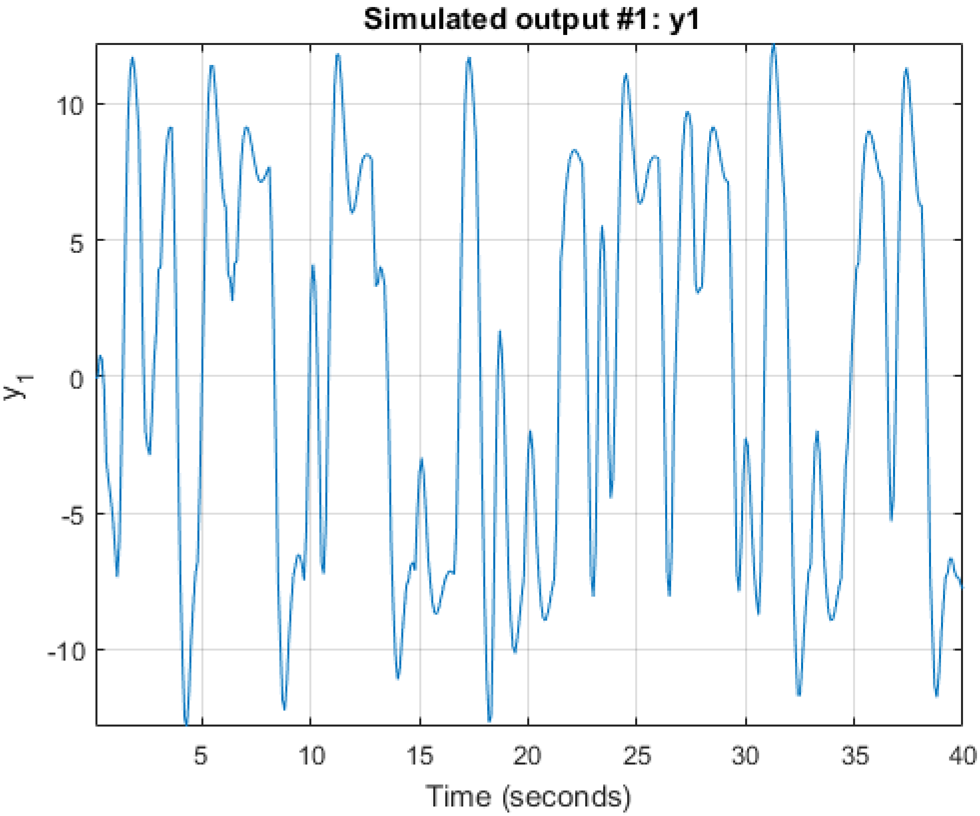
```
M = nlhw(z2,[4 3 2],[],'pwlinear');
```

Compute steady-state operating point values corresponding to an input level of 1 and an unknown output level.

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

Simulate the model using the estimated initial states.

```
opt = simOptions('InitialCondition',x0);
sim(M,z2.u)
```



Simulate Time-Series Model

Load time-series data, and estimate an AR model using the least-squares approach.

```
load iddata9 z9
sys = ar(z9,6,'ls');
```

For time-series data,, specify the desired simulation length, $N = 200$ using an N-by-`0` input data set.

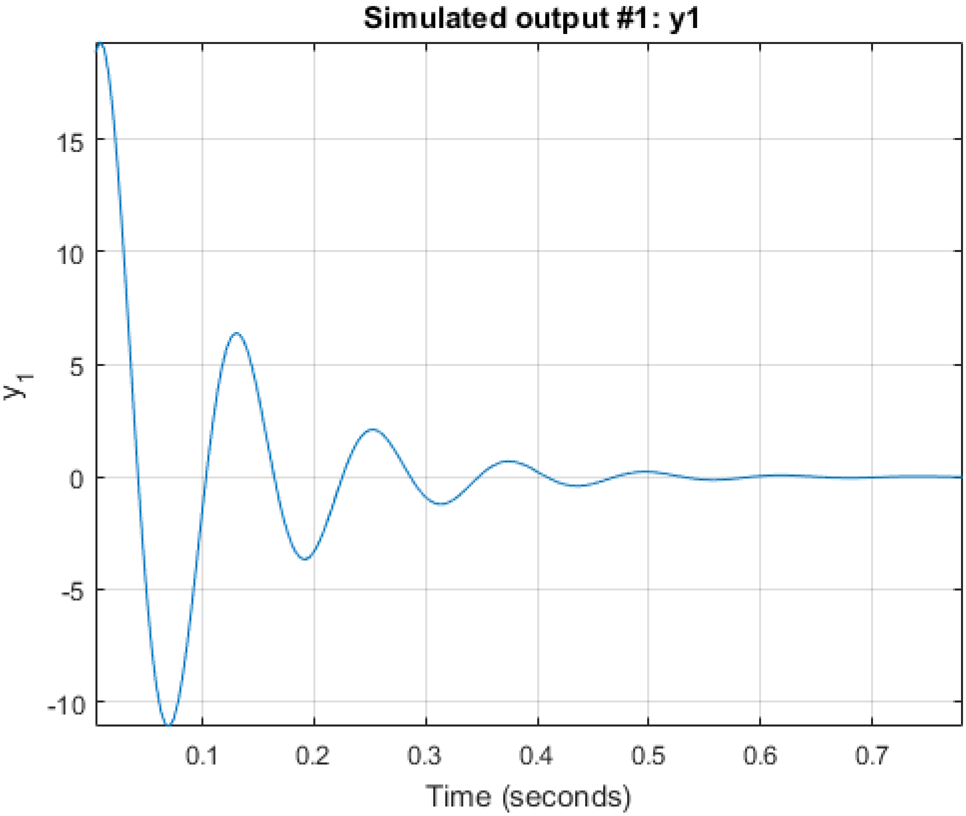
```
data = iddata([],zeros(200,0),z9.Ts);
```

Set the initial conditions to use the initial samples of the time series as historical output samples.

```
IC = struct('Input',[],'Output',z9.y(1:6));
opt = simOptions('InitialCondition',IC);
```

Simulate the model.

```
sim(sys,data,opt)
```



- “Simulate a Continuous-Time State-Space Model”
- “Simulate Model Output with Noise”

Input Arguments

`sys` — Identified model

identified linear model | identified nonlinear model

Identified model, specified as one of the following model objects:

	Model Type	Model Object
Identified Linear Model	Polynomial model	`idpoly`
	Process model	`idproc`
	State-space model	`idss`
	Transfer function model	`idtf`
	Linear grey-box model	`idgrey`
Identified Nonlinear Model	Nonlinear ARX model	`idnlarx`
	Nonlinear Hammerstein-Wiener model	`idnlhw`
	Nonlinear grey-box model	`idnlgrey`

`udata` — Simulation input data

`iddata` object | matrix

Simulation input data, specified as an `iddata` object or a matrix. `sim` uses the input channels from this object as the simulation inputs. For time-domain simulation of discrete-time systems, you can also specify `udata` as a matrix with columns that correspond to each input channel.

If `sys` is a linear model, you can use either time-domain or frequency-domain data. If `sys` is a nonlinear model, you can only use time-domain data.

If `sys` is a time-series model, that is a model with no inputs, specify `udata` as an *Ns*-by-0 signal, where *Ns* is the wanted number of simulation output samples. For example, to simulate `100` output samples, specify `udata` as follows.

```
udata = iddata([],zeros(100,0),Ts);
```

If you do not have data from an experiment, use `idinput` to generate signals with various characteristics.

`opt` — Simulation options

`simOptions` option set

Simulation options, specified as a `simOptions` option set for setting the following options:

- Initial conditions
- Input/output offsets
- Additive noise

Output Arguments

`y` — Simulated response

`iddata` object

Simulated response for `sys`, returned as an `iddata` object.

If `udata` represents time-domain data, then `y` is the simulated response for the time vector corresponding to `udata`.

If `udata` represents frequency-domain data, $U(\omega)$, then `y` contains the Fourier transform of the corresponding sampled time-domain output signal. This signal is the product of the frequency response of `sys`, $G(\omega)$, and $U(\omega)$.

For multi-experiment data, `y` is a corresponding multi-experiment `iddata` object.

`y_sd` — Estimated standard deviation

`iddata` object | `[]`

Estimated standard deviation of the simulated response for linear models or nonlinear grey-box models, returned as an `iddata` object.

`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

matrix | `[]`

Estimated state trajectory for state-space models, returned as an *Ns*-by-*Nx* matrix, where *Ns* is the number of samples and *Nx* is the number of states.

`x` is only relevant if `sys` is a state-space model (`idss`, `idgrey`, or `idnlgrey`). If `sys` is not a state-space model, `x` is returned as `[]`.

`x_sd` — Estimated standard deviation of state trajectory
matrix | `[]`

Estimated standard deviation of state trajectory for state-space models, returned as an *Ns*-by-*Nx* matrix, where *Ns* is the number of samples and *Nx* is the number of states.

`x_sd` is only relevant if `sys` is a state-space model (`idss`, `idgrey`, or `idnlgrey`). If `sys` is not a state-space model, `x_sd` is returned as `[]`.

Alternatives

- Use `simsd` for a Monte-Carlo method of computing the standard deviation of the response.
- `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 preceding operations, use `lsim`.

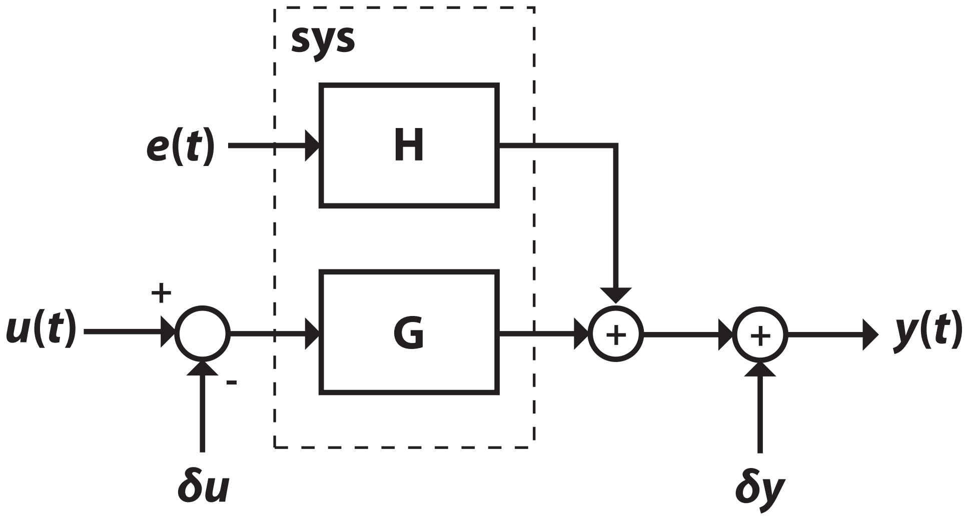
More About

Tips

- When the initial conditions of the estimated model and the system that measured the validation data set are different, the simulated and measured responses may also differ, especially at the beginning of the response. To minimize this difference, estimate the initial state values using `findstates` and use the estimated values to set the `InitialCondition` option using `simOptions`. For an example, see “Match Model Response to Output Data” on page 1-1339.

Algorithms

Simulation means computing the model response using input data and initial conditions. `sim` simulates the following system:



Here,

- *u*(*t*) is the simulation input data, `udata`.
- *y*(*t*) is the simulated output response.
- *G* is the transfer function from the input to the output and is defined in `sys`. The simulation initial conditions, as specified using `simOptions`, set the initial state of *G*.
- *e*(*t*) is an optional noise signal. Add noise to your simulation by creating a `simOptions` option set, and setting the `AddNoise` option to `true`. Additionally, you can change the default noise signal by specifying the `NoiseData` option.
- *H* is the noise transfer function and is defined in `sys`.

- δu is an optional input offset subtracted from the input signal, $u(t)$, before the input is used to simulate the model. Specify an input offset by setting the `InputOffset` option using `simOptions`.
- δy is an optional output offset added to the output response, $y(t)$, after simulation. Specify an output offset by setting the `OutputOffset` option using `simOptions`.

For more information on specifying simulation initial conditions, input and output offsets, and noise signal data, see `simOptions`. For multiexperiment data, you can specify these options separately for each experiment.

- “Simulating and Predicting Model Output”
- “Why Simulate or Predict Model Output?”

See Also

`compare` | `findstates` | `forecast` | `idinput` | `lsim` | `predict` | `simOptions` | `simsd` | `step`

Introduced before R2006a

simOptions

Option set for `sim`

Syntax

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

Description

`opt = simOptions` creates the default option set for `sim`.

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

Examples

Create Default Option Set for Model Simulation

```
opt = simOptions;
```

Specify Options for Model Simulation

Create an option set for `sim` specifying the following options.

- Zero initial conditions
- Input offset of 5 for the second input of a two-input model

```
opt = simOptions('InitialCondition','z','InputOffset',[0; 5]);
```

Add Noise to Simulation Output

Create noise data for a simulation with `500` input data samples and two outputs.

```
noiseData = randn(500,2);
```

Create a default option set.

```
opt = simOptions;
```

Modify the option set to add the noise data.

```
opt.AddNoise = true;
opt.NoiseData = noiseData;
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example: `'AddNoise',true','InputOffset',[5;0]` adds default Gaussian white noise to the response model and specifies an input offset of `5` for the first of two model inputs.

`'InitialCondition'` — Simulation initial conditions

`'z'` (default) | column vector | matrix | structure | structure array | `'model'`

Simulation initial conditions, specified as one of the following:

- `'z'` — Zero initial conditions.
- Numerical column vector of initial states with length equal to the model order.

  For multi-experiment data, specify a matrix with *Ne* columns, where *Ne* is the number of experiments, to configure the initial conditions separately for each experiment. Otherwise, use a column vector to specify the same initial conditions for all experiments.

  Use this option for state-space models (`idss` and `idgrey`) only.
- Structure with the following fields, which contain the historical input and output values for a time interval immediately before the start time of the data used in the simulation:

Field	Description
`Input`	Input history, specified as a matrix with *Nu* columns, where *Nu* is the number of input channels. For time-series models, use `[]`.

Field	Description
`Output`	Output history, specified as a matrix with *Ny* columns, where *Ny* is the number of output channels.

For each field, if the respective signals are constant before the start time of the simulation, use a row vector to specify the constant values.

For multi-experiment data, you can configure the initial conditions separately for each experiment by specifying `InitialCondition` as a structure array with *Ne* elements. Otherwise, use a single structure to specify the same initial conditions for all experiments.

- `'model'` — Initial conditions as specified in the `InitialStates` property of the model. Use this option for `idnlgrey` models only. This option corresponds to the default initial conditions for `idnlgrey` models when no `simOptions` option set is used.

`'X0Covariance'` — Covariance of initial states vector

[ ] (default) | matrix

Covariance of initial states vector, specified as one of the following:

- Positive definite matrix of size *Nx*-by-*Nx*, where *Nx* is the model order.

  For multi-experiment data, specify as an *Nx*-by-*Nx*-by-*Ne* matrix, where *Ne* is the number of experiments.
- `[]` — No uncertainty in the initial states.

Use this option only for state-space models (`idss` and `idgrey`) when `'InitialCondition'` is specified as a column vector. Use this option to account for initial condition uncertainty when computing the standard deviation of the simulated response of a model.

`'InputOffset'` — Input signal offset

[ ] (default) | column vector | matrix

Input signal offset, specified as a column vector of length *Nu*. Use `[]` if there are no input offsets. Each element of `InputOffset` is subtracted from the corresponding input data before the input is used to simulate the model.

For multiexperiment data, specify `InputOffset` as:

- An *Nu*-by-*Ne* matrix to set offsets separately for each experiment.
- A column vector of length *Nu* to apply the same offset for all experiments.

`'OutputOffset'` — Output signal offset

`[]` (default) | column vector | matrix

Output signal offset, specified as a column vector of length *Ny*. Use `[]` if there are no output offsets. Each element of `OutputOffset` is added to the corresponding simulated output response of the model.

For multiexperiment data, specify `OutputOffset` as:

- An *Ny*-by-*Ne* matrix to set offsets separately for each experiment.
- A column vector of length *Ny* to apply the same offset for all experiments.

`'AddNoise'` — Noise addition toggle

`false` (default) | `true`

Noise addition toggle, specified as a logical value indicating whether to add noise to the response model.

`'NoiseData'` — Noise signal data

`[]` (default) | matrix | cell array of matrices

Noise signal data specified as one of the following:

- `[]` — Default Gaussian white noise.
- Matrix with N_s rows and N_y columns, where N_s is the number of input data samples, and N_y is the number of outputs. Each matrix entry is scaled according to `NoiseVariance` property of the simulated model and added to the corresponding output data point. To set `NoiseData` at a level that is consistent with the model, use white noise with zero mean and a unit covariance matrix.
- Cell array of N_e matrices, where N_e is the number of experiments for multiexperiment data. Use a cell array to set the `NoiseData` separately for each experiment, otherwise set the same noise signal for all experiments using a matrix.

`NoiseData` is the noise signal, $e(t)$, for the model

$$y(t) = Gu(t) + He(t).$$

Here,G is the transfer function from the input, $u(t)$, to the output, $y(t)$, and H is the noise transfer function.

`NoiseData` is used for simulation only when `AddNoise` is true.

Output Arguments

`opt` — Option set for `sim` command

`simOptions` option set

Option set for `sim` command, returned as a `simOptions` option set.

See Also

`sim`

Introduced in R2012a

simsd

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.

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, created using `simsdOptions`, that specifies options including:

- Signal offsets
- Initial condition handling
- Additive noise

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.

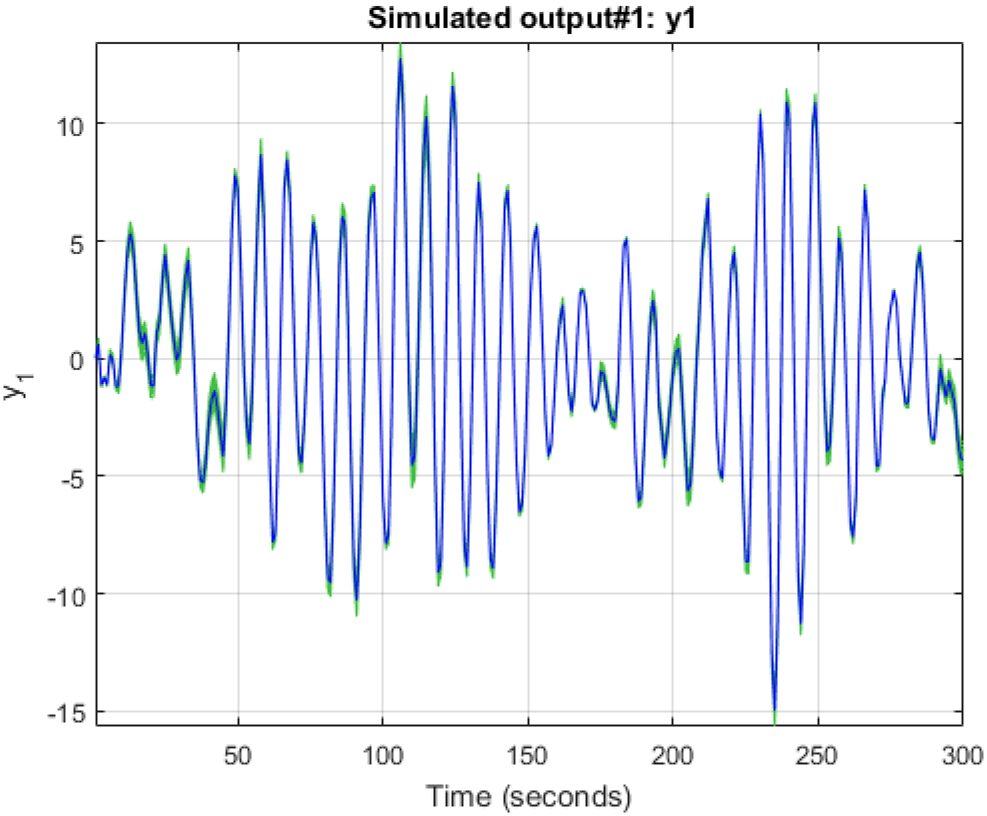
Obtain an identified model.

```
load iddata3
sys = ssest(z3,2,'form','canon');
```

`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)
```



More About

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`.

See Also

`getcov` | `rsample` | `showConfidence` | `sim` | `simsdOptions`

Introduced before R2006a

simsdOptions

Option set for `simsd`

Syntax

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

Description

`opt = simsdOptions` creates the default option set for `simsd`.

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

Examples

Create Default Option Set for Uncertain Model Simulation

```
opt = simsdOptions;
```

Specify Options for Uncertain Model Simulation

Create an option set for `simsd` specifying the following options.

- Zero initial conditions
- Input offset of 5 for the second input of a two-input model

```
opt = simsdOptions('InitialCondition','z','InputOffset',[0; 5]);
```

Add Noise to Uncertain Simulation Output

Create noise data for a simulation with 500 input data samples and two outputs.

```
noiseData = randn(500,2);
```

Create a default option set.

```
opt = simsdOptions;
```

Modify the option set to add the noise data.

```
opt.AddNoise = true;
opt.NoiseData = noiseData;
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example: `'AddNoise',true','InputOffset',[5;0]` adds default Gaussian white noise to the response model and specifies an input offset of `5` for the first of two model inputs.

`'InitialCondition'` — Simulation initial conditions

`'z'` (default) | column vector | matrix | structure | structure array

Simulation initial conditions, specified as one of the following:

- `'z'` — Zero initial conditions.
- Numerical column vector of initial states with length equal to the model order.

  For multi-experiment data, specify a matrix with *Ne* columns, where *Ne* is the number of experiments, to configure the initial conditions separately for each experiment. Otherwise, use a column vector to specify the same initial conditions for all experiments.

  Use this option for state-space models (`idss` and `idgrey`) only.
- Structure with the following fields, which contain the historical input and output values for a time interval immediately before the start time of the data used in the simulation:

Field	Description
`Input`	Input history, specified as a matrix with *Nu* columns, where *Nu* is the number of input channels. For time-series models, use `[]`.

Field	Description
`Output`	Output history, specified as a matrix with *Ny* columns, where *Ny* is the number of output channels.

For each field, if the respective signals are constant before the start time of the simulation, use a row vector to specify the constant values.

For multi-experiment data, you can configure the initial conditions separately for each experiment by specifying `InitialCondition` as a structure array with *Ne* elements. Otherwise, use a single structure to specify the same initial conditions for all experiments.

`'InputOffset'` — Input signal offset

`[]` (default) | column vector | matrix

Input signal offset, specified as a column vector of length *Nu*. Use `[]` if there are no input offsets. Each element of `InputOffset` is subtracted from the corresponding input data before the input is used to simulate the model.

For multiexperiment data, specify `InputOffset` as:

- An *Nu*-by-*Ne* matrix to set offsets separately for each experiment.
- A column vector of length *Nu* to apply the same offset for all experiments.

`'OutputOffset'` — Output signal offset

`[]` (default) | column vector | matrix

Output signal offset, specified as a column vector of length *Ny*. Use `[]` if there are no output offsets. Each element of `OutputOffset` is added to the corresponding simulated output response of the model.

For multiexperiment data, specify `OutputOffset` as:

- An *Ny*-by-*Ne* matrix to set offsets separately for each experiment.
- A column vector of length *Ny* to apply the same offset for all experiments.

`'AddNoise'` — Noise addition toggle

`false` (default) | `true`

Noise addition toggle, specified as a logical value indicating whether to add noise to the response model.

`'NoiseData'` — Noise signal data

[ ] (default) | matrix | cell array of matrices

Noise signal data specified as one of the following:

- `[]` — Default Gaussian white noise.
- Matrix with N_s rows and N_y columns, where N_s is the number of input data samples, and N_y is the number of outputs. Each matrix entry is scaled according to `NoiseVariance` property of the simulated model and added to the corresponding output data point. To set `NoiseData` at a level that is consistent with the model, use white noise with zero mean and a unit covariance matrix.
- Cell array of N_e matrices, where N_e is the number of experiments for multiexperiment data. Use a cell array to set the `NoiseData` separately for each experiment, otherwise set the same noise signal for all experiments using a matrix.

`NoiseData` is the noise signal, $e(t)$, for the model

$$y(t) = Gu(t) + He(t).$$

Here,G is the transfer function from the input, $u(t)$, to the output, $y(t)$, and H is the noise transfer function.

`NoiseData` is used for simulation only when `AddNoise` is true.

Output Arguments

`opt` — Option set for `simsd` command

`simsdOptions` option set

Option set for `simsd` command, returned as a `simsdOptions` option set.

See Also

`simsd`

Introduced in R2012a

size

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

`nparams` | `isempty` | `issiso` | `ndims`

Introduced before R2006a

spa

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.

Information about the estimation results and options used is stored in the model's `Report` property. `Report` has the following fields:

- `Status` — Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.
- `Method` — Estimation command used.
- `WindowSize` — Size of the Hann window.
- `DataUsed` — Attributes of the data used for estimation. Structure with the following fields:
  - `Name` — Name of the data set.
  - `Type` — Data type.
  - `Length` — Number of data samples.
  - `Ts` — Sample time.
  - `InterSample` — Input intersample behavior.
  - `InputOffset` — Offset removed from time-domain input data during estimation.
  - `OutputOffset` — Offset removed from time-domain output data during estimation.

`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.

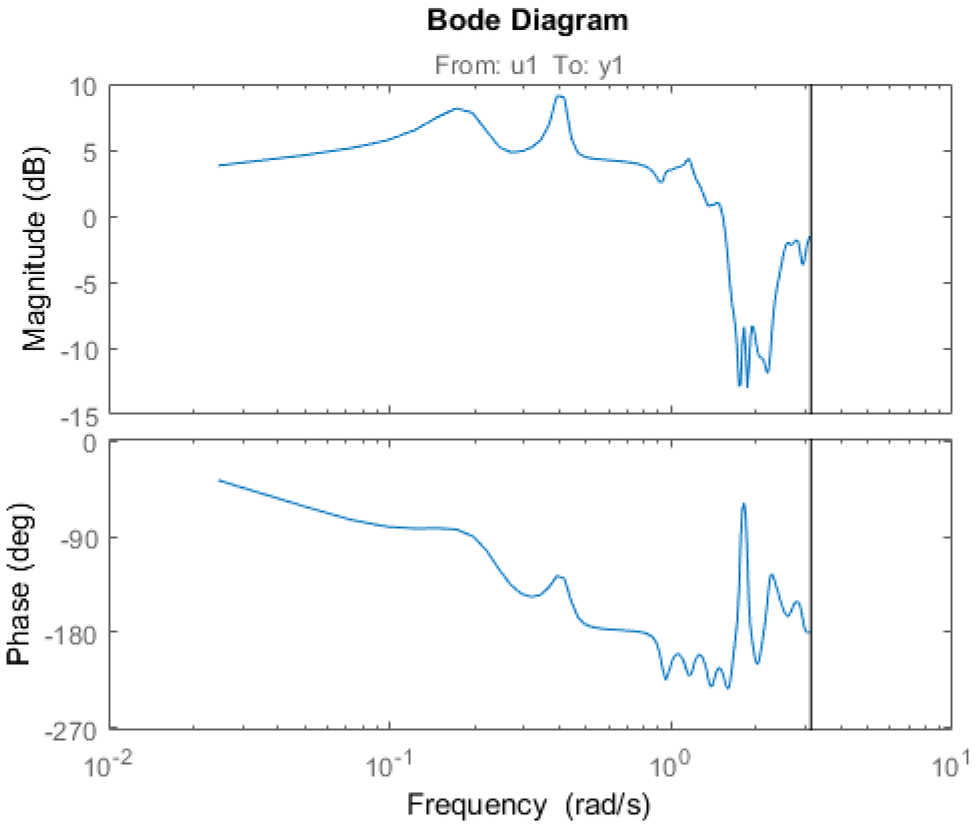
`G = spa(data,winSize,freq,MaxSize)` can improve computational performance using `MaxSize` to split the input-output data such that each segment contains fewer than `MaxSize` elements. `MaxSize` is a positive integer.

Examples

Estimate Frequency Response

Estimate frequency response with fixed resolution at 128 equally spaced, logarithmic frequency values between 0 (excluded) and π.

```
load iddata3;
g = spa(z3);
bode(g)
```



Estimate Frequency Response at Specified Frequencies

Define the frequency vector.

```
w = logspace(-2,pi,128);
```

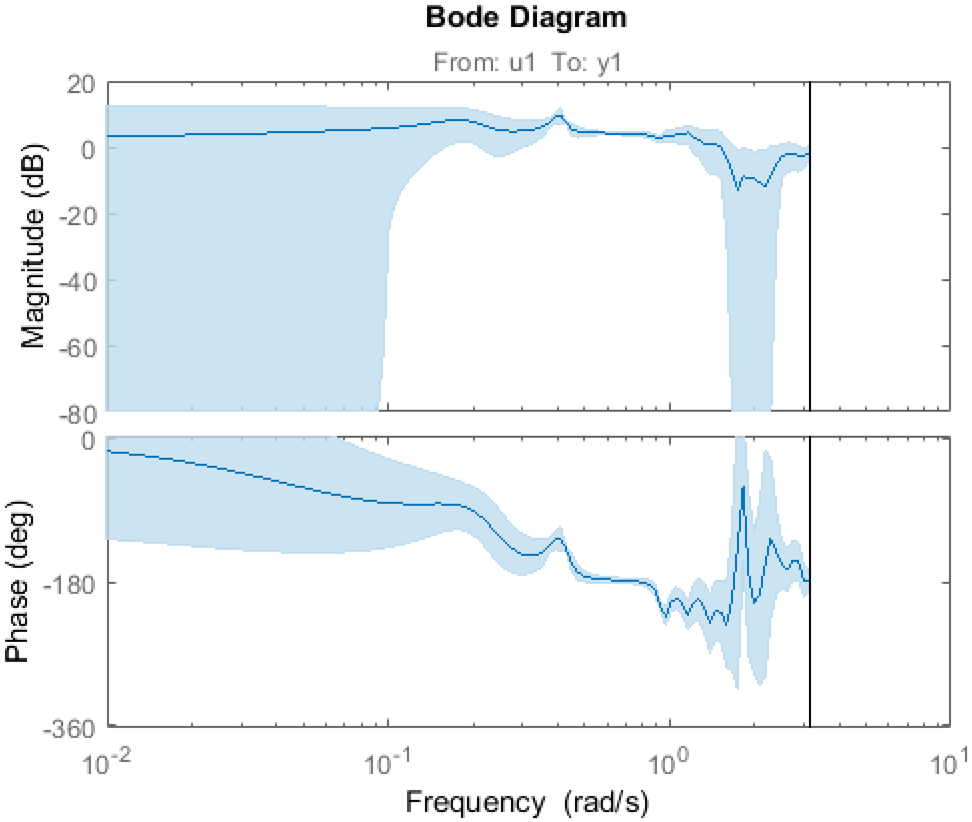
Compute the frequency response.

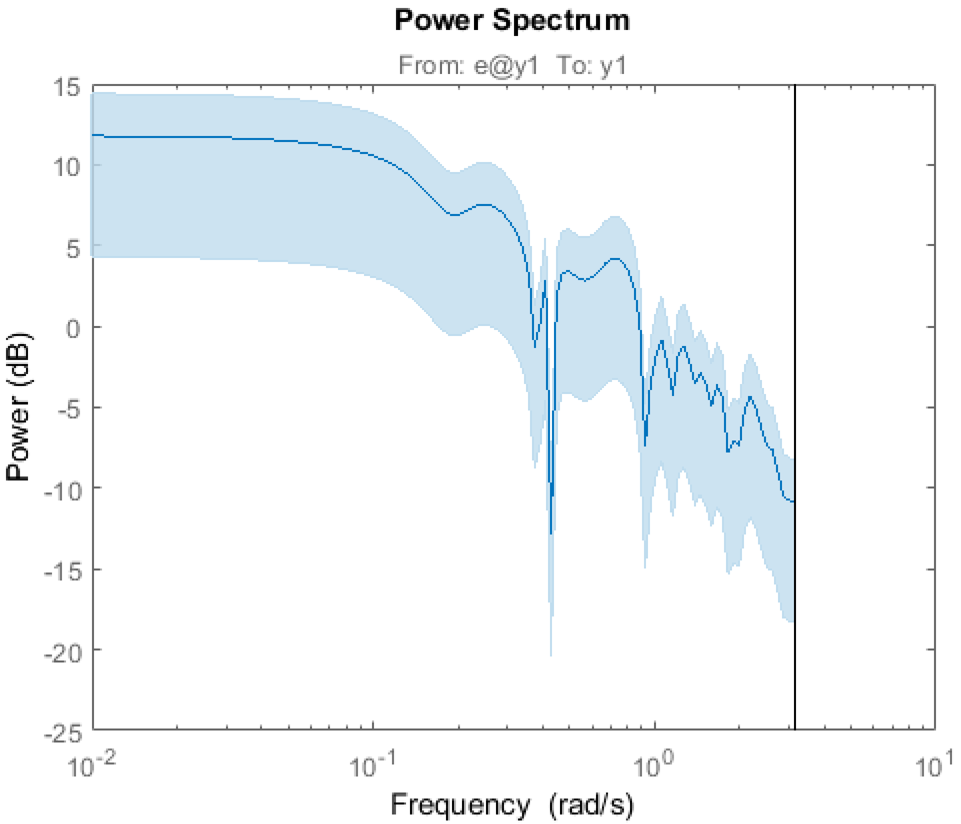
```
load iddata3;
g = spa(z3,[],w);
```

`[]` specifies the default lag window size.

Plot the Bode response and disturbance spectrum with confidence interval of 3 standard deviations.

```
h = bodeplot(g);
showConfidence(h,3)
figure
h = spectrumplot(g);
showConfidence(h,3)
```





More About

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:

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

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

Together, $G(q=e^{iw})$ 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:

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

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

Output Noise Spectrum

The output noise spectrum (spectrum of *v(t)*) is given by the following equation:

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

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

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

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

where the noise spectrum is given by the following equation:

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

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

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

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

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

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

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)*:

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

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 2π/`M` rad/sample time.

By default, this operation uses 128 equally spaced frequency values between 0 (excluded) and π, where `w = [1:128]/128*pi/Ts` and `Ts` is the sample time of that data set. The default lag size of the Hann window is `M = min(length(data)/10,30)`. For default frequencies, uses fast Fourier transforms (FFT)—which is more efficient than for user-defined frequencies.

Note: `M =`γ 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 $\hat{G}_N\left(e^{i\omega}\right)$ and the output noise spectrum $\hat{\Phi}_v(\omega)$.

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

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

`spectrum` is the spectrum matrix for both the output and the input channels. That is, if `z = [data.OutputData`, `data.InputData]`, `spectrum` contains as spectrum data the matrix-valued power spectrum of `z`.

$$S=\sum_{m=-M}^{M} Ez(t+m)z(t)' W_M(T_s)\exp(-i\omega m)$$

`'` is a complex-conjugate transpose.

- “Estimate Impulse-Response Models at the Command Line”
- “Spectrum Normalization”

References

Ljung, L. *System Identification: Theory for the User*, Second Ed., Prentice Hall PTR, 1999.

See Also

`etfe` | `freqresp` | `idfrd` | `spafdr` | `bode` | `spectrum`

Introduced before R2006a

spafdr

Estimate frequency response and spectrum using spectral analysis with frequency-dependent resolution

Syntax

```
g = spafdr(data)
g = spafdr(data,Resol,w)
```

Description

`g = spafdr(data)` estimates the transfer function and noise spectrum Φ_v of the general linear model

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

where $\Phi_v(\omega)$ is the spectrum of $v(t)$. `data` contains the output-input data as an `iddata` object. The data can be complex valued, and either time or frequency domain. It can also be an `idfrd` object containing frequency-response data. `g` is an `idfrd` object with the estimate of $G\left(e^{i\omega}\right)$ at the frequencies ω specified by row vector `w`. `g` also includes information about the spectrum estimate of $\Phi_v(\omega)$ at the same frequencies. Both results are returned with estimated covariances, included in `g`. The normalization of the spectrum is the same as described in `spa`.

Information about the estimation results and options used is stored in the model's `Report` property. `Report` has the following fields:

- `Status` — Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.
- `Method` — Estimation command used.
- `WindowSize` — Frequency resolution.
- `DataUsed` — Attributes of the data used for estimation. Structure with the following fields:
  - `Name` — Name of the data set.

- `Type` — Data type.
- `Length` — Number of data samples.
- `Ts` — Sample time.
- `InterSample` — Input intersample behavior.
- `InputOffset` — Offset removed from time-domain input data during estimation.
- `OutputOffset` — Offset removed from time-domain output data during estimation.

`g = spafdr(data,Resol,w)` specifies frequencies and frequency resolution.

Frequencies

The frequency variable `w` is either specified as a row vector of frequencies, or as a cell array `{wmin,wmax}`. In the latter case the covered frequencies will be 50 logarithmically spaced points from `wmin` to `wmax`. You can change the number of points to `NP` by entering `{wmin,wmax,NP}`.

Omitting `w` or entering it as an empty matrix gives the default value, which is 100 logarithmically spaced frequencies between the smallest and largest frequency in data. For time-domain data, this means from `1/N*Ts` to `pi*Ts`, where `Ts` is the sample time 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.Report.WindowSize`.

More About

Algorithms

If the data is given in the time domain, it is first converted to the frequency domain. Then averages of `Y(w)Conj(U(w))` and `U(w)Conj(U(w))` are formed over the frequency ranges `w`, corresponding to the desired resolution around the frequency in question. The ratio of these averages is then formed for the frequency-function estimate, and corresponding expressions define the noise spectrum estimate.

See Also

`bode` | `etfe` | `freqresp` | `idfrd` | `nyquist` | `spa` | `spectrum`

Introduced before R2006a

spectrum

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 `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 sample time `Ts`, `spectrum` uses the transformation `z = exp(j*w*Ts)` to map the unit circle to the real frequency axis. The spectrum is only plotted for frequencies smaller than the Nyquist frequency `pi/Ts`, and the default value 1 (time unit) is assumed when Ts 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

Plot Noise Spectrum of SISO Linear Identified Model

Load the estimation data.

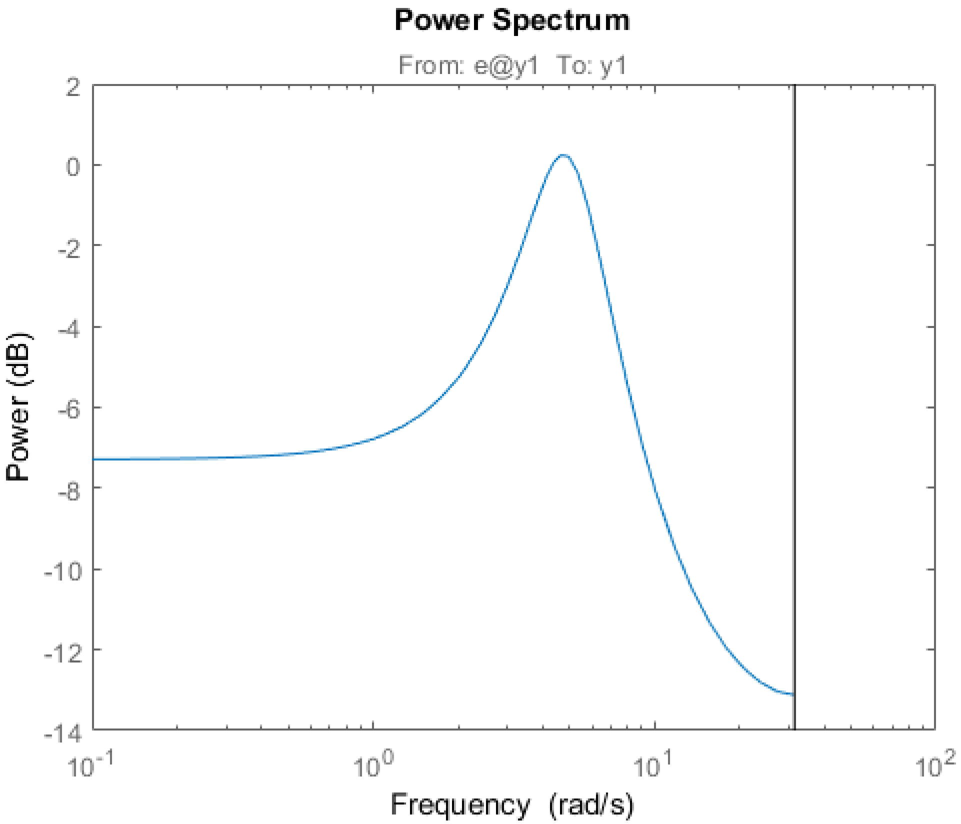
```
load iddata1 z1;
```

Estimate a single-input single-output state-space model.

```
sys = n4sid(z1,2);
```

Plot the noise spectrum for the model.

```
spectrum(sys);
```



Plot Output Spectrum of Time-Series Model

Load the time-series estimation data.

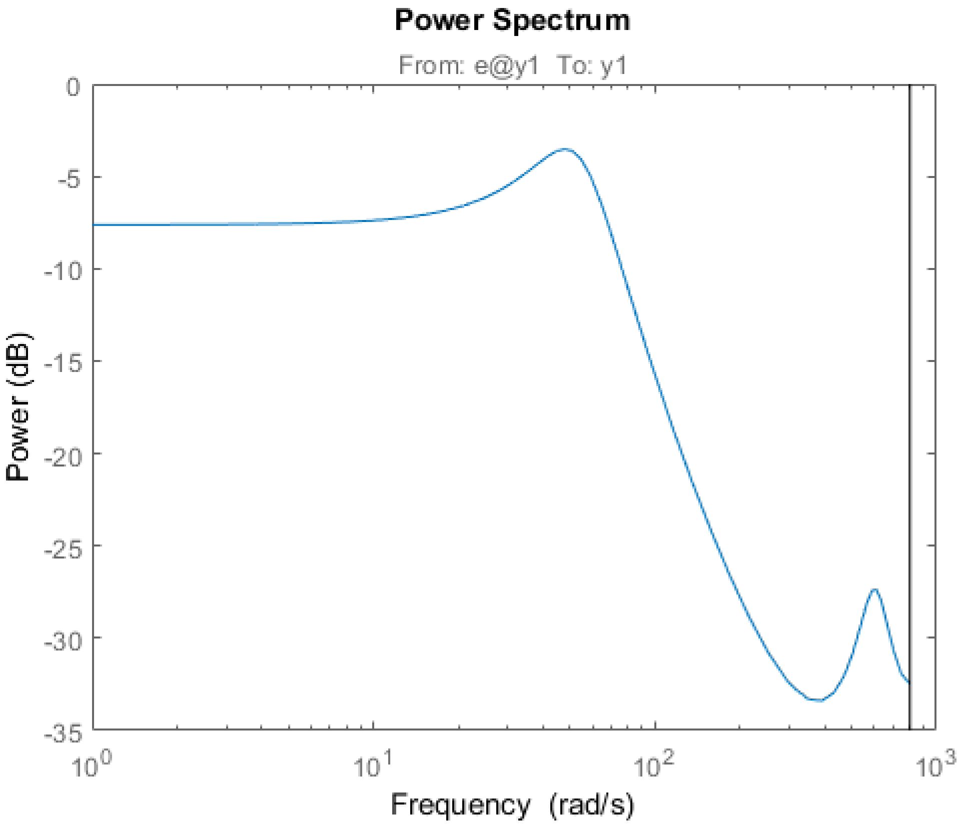
```
load iddata9 z9
```

Estimate a fourth-order AR model using a least-squares approach.

```
sys = ar(z9,4,'ls');
```

Plot the output spectrum of the model.

```
spectrum(sys);
```



See Also

`ar` | `armax` | `arx` | `bode` | `forecast` | `freqresp` | `nlarx`

Introduced in R2012a

spectrumoptions

Option set for `spectrumplot`

Syntax

```
opt = spectrumoptions
opt = spectrumoptions('identpref')
```

Description

`opt = spectrumoptions` creates the default option set for `spectrumplot`. Use dot notation to customize the option set, if needed.

`opt = spectrumoptions('identpref')` initializes the plot options with the System Identification Toolbox preferences. Use this syntax to change a few plot options but otherwise use your toolbox preferences.

Examples

Specify Options for Spectrum Plot

Specify the plot options.

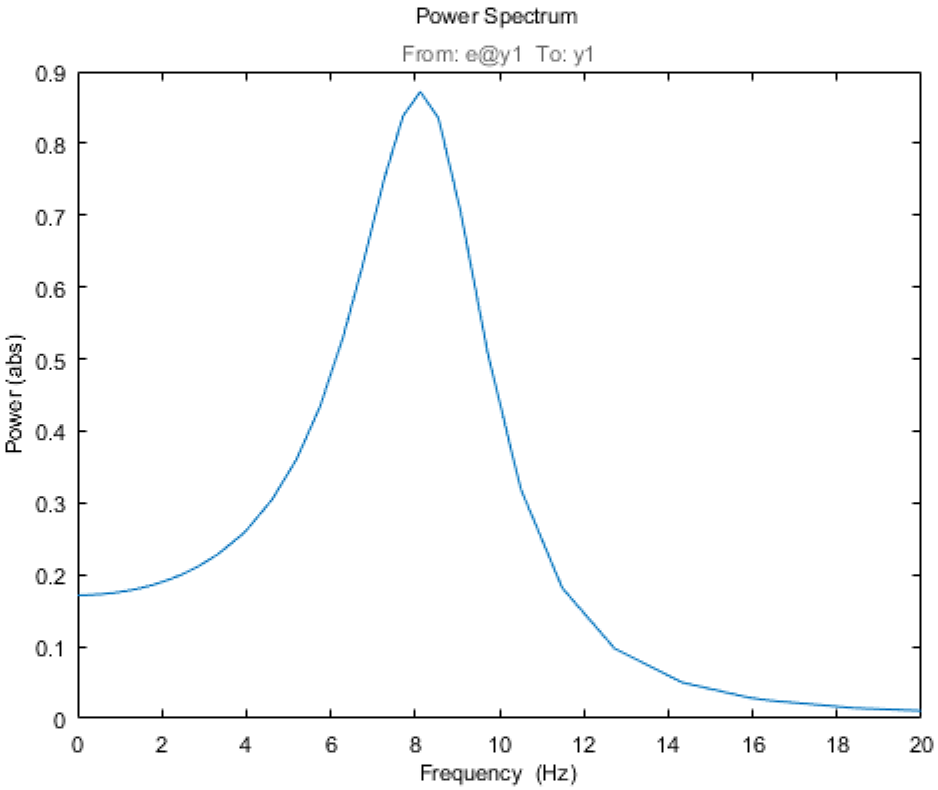
```
plot_options = spectrumoptions;
plot_options.FreqUnits = 'Hz';
plot_options.FreqScale = 'linear';
plot_options.Xlim = {[0 20]};
plot_options.MagUnits = 'abs';
```

Estimate an AR model.

```
load iddata9 z9
sys = ar(z9,4);
```

Plot the output spectrum for the model.

```
spectrumplot(sys,plot_options);
```



Initialize Plot Options Using Toolbox Preferences

```
opt = spectrumoptions('identpref');
```

Output Arguments

`opt` — Option set for `spectrumplot`

`spectrumpoptions` option set

Option set containing the specified options for `spectrumplot`.

Field	Description
`Title`, `XLabel`, `YLabel`	Text and style for axes labels and plot title, specified as a structure array with the following fields: • `String` — Title and axes label text, specified as a character vector. **Default `Title`**: `'Power Spectrum'` **Default `XLabel`**: `'Frequency'` **Default `YLabel`**: `'Power'` • `FontSize` — Font size, specified as data type `scalar`. **Default**: `8` • `FontWeight` — Thickness of text, specified as one of the following values: `'Normal'` | `'Bold'` **Default**: `'Normal'` • `Font Angle` — Text character angle, specified as one of the following values: `'Normal'` | `'Italic'` **Default**: `'Normal'` • `Color` — Color of text, specified as vector of RGB values between `0` to `1`. **Default**: `[0,0,0]` • `Interpreter` — Interpretation of text characters, specified as one of the following values: `'tex'` | `'latex'`| `'none'` **Default**: `'tex'`

Field	Description
`TickLabel`	Tick label style, specified as a structure array with the following fields: • `FontSize` — Font size, specified as data type `scalar`. **Default**: `8` • `FontWeight` — Thickness of text, specified as one of the following values: `'Normal'` | `'Bold'` **Default**: `'Normal'` • `Font Angle` — Text character angle, specified as one of the following values: `'Normal'` | `'Italic'` **Default**: `'Normal'` • `Color` — Color of text, specified as vector of RGB values between `0` to `1` | character vector of color name | `'none'`. For example, for yellow color, specify as one of the following: `[1 1 0]`, `'yellow'`, or `'y'`. **Default**: `[0,0,0]`
`Grid`	Show or hide the grid, specified as one of the following values: `'off'` | `'on'` **Default**: `'off'`
`GridColor`	Color of the grid lines, specified as one of the following: vector of RGB values in the range `[0,1]` | character vector of color name | `'none'`. For example, for yellow color, specify as one of the following: `[1 1 0]`, `'yellow'`, or `'y'`. **Default**: `[0.15,0.15,0.15]`

Field	**Description**
`XLimMode`, `YLimMode`	Axes limit modes, specified as one of the following values: • `'auto'` — The axes limits are based on the data plotted • `'manual'` — The values are explicitly set with `Xlim`, `Ylim` **Default**: `'auto'`
`XLim`, `YLim`	Axes limits, specified as an array of the form `[min,max]`
`IOGrouping`	Grouping of input-output pairs in the plot, specified as one of the following values: `'none'` | `'inputs'`| `'outputs'`|`'all'` **Default**: `'none'`

Field	Description
`InputLabels`, `OutputLabels`	Input and output label styles, specified as a structure array with the following fields: • `FontSize` — Font size, specified as data type `scalar`. **Default**: `8` • `FontWeight` — Thickness of text, specified as one of the following values: `'Normal'` | `'Bold'` **Default**: `'Normal'` • `Font Angle` — Text character angle, specified as one of the following values: `'Normal'` | `'Italic'` **Default**: `'Normal'` • `Color` — Color of text, specified as a vector of RGB values between `0` to `1` | character vector of color name | `'none'`. For example, for yellow color, specify as one of the following: `[1 1 0]`, `'yellow'`, or `'y'`. **Default**: `[0.4,0.4,0.4]` • `Interpreter` — Interpretation of text characters, specified as one of the following values: `'tex'` | `'latex'`| `'none'` **Default**: `'tex'`
`InputVisible`, `OutputVisible`	Visibility of input and output channels, specified as one of the following values: `'off'` | `'on'` **Default**: `'on'`
`ConfidenceRegionNumberSD`	Number of standard deviations to use to plot the response confidence region. **Default**: `1`

Field	Description
`FreqUnits`	Frequency units, specified as one of the following values: • `'Hz'` • `'rad/second'` • `'rpm'` • `'kHz'` • `'MHz'` • `'GHz'` • `'rad/nanosecond'` • `'rad/microsecond'` • `'rad/millisecond'` • `'rad/minute'` • `'rad/hour'` • `'rad/day'` • `'rad/week'` • `'rad/month'` • `'rad/year'` • `'cycles/nanosecond'` • `'cycles/microsecond'` • `'cycles/millisecond'` • `'cycles/hour'` • `'cycles/day'` • `'cycles/week'` • `'cycles/month'` • `'cycles/year'` **Default**: `'rad/s'` You can also specify `'auto'`, which uses frequency units `rad/TimeUnit` relative

Field	Description
	to system time units specified in the `TimeUnit` property. For multiple systems with different time units, the units of the first system are used.
`FreqScale`	Frequency scale, specified as one of the following values: `'linear'` | `'log'` **Default**: `'log'`
`MagUnits`	Magnitude units, specified as one of the following values: `'dB'` | `'abs'` **Default**: `'dB'`
`MagScale`	Magnitude scale, specified as one of the following values: `'linear'` | `'log'` **Default**: `'linear'`
`MagLowerLimMode`	Enables a lower magnitude limit, specified as one of the following values: `'auto'` | `'manual'` **Default**: `'auto'`
`MagLowerLim`	Lower magnitude limit, specified as data type `double`.

See Also

`getoptions` | `identpref` | `setoptions` | `spectrumplot`

Introduced in R2012a

spectrumplot

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.

Default:

`line_spec`

Line style, marker, and color of both the line and marker, specified as a character vector. For example, `'b'`, `'b+:'`.

For more information, see Chart Line 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, `spectrumoptions`, to create `plot_options`. For more information, type `help spectrumoptions`.

w

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

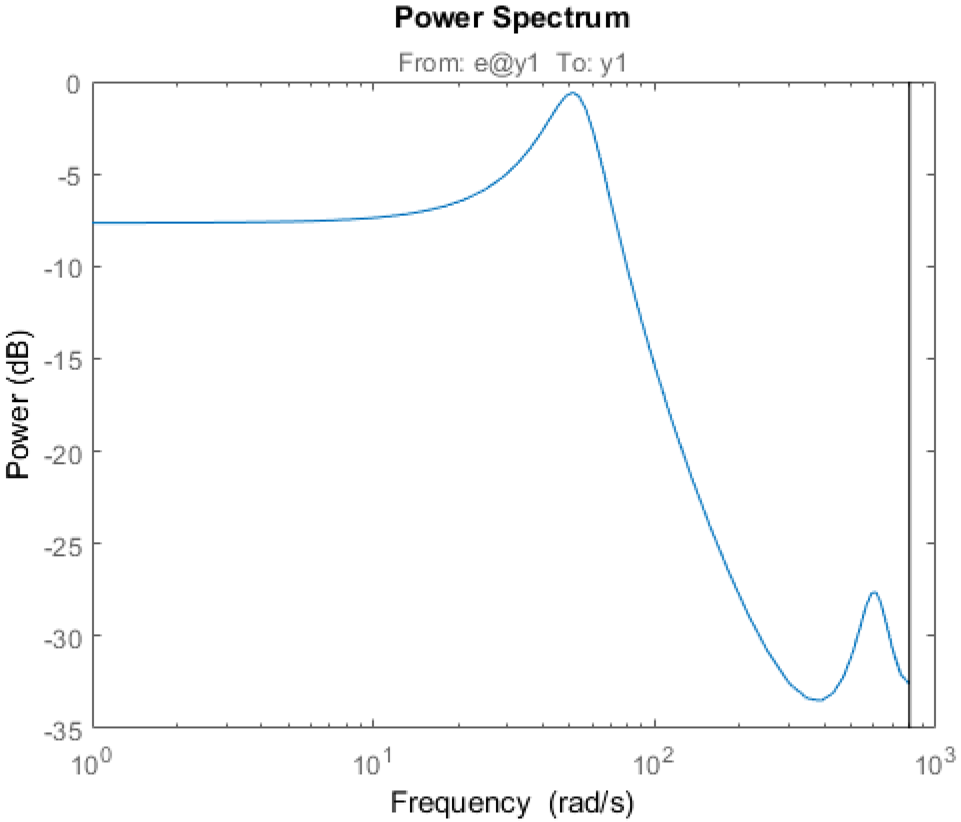
Plot Model Output Spectrum for Identified Model

Obtain the identified model.

```
load iddata9 z9
sys = ar(z9,4);
```

Plot the output spectrum for the model.

```
spectrumplot(sys);
```



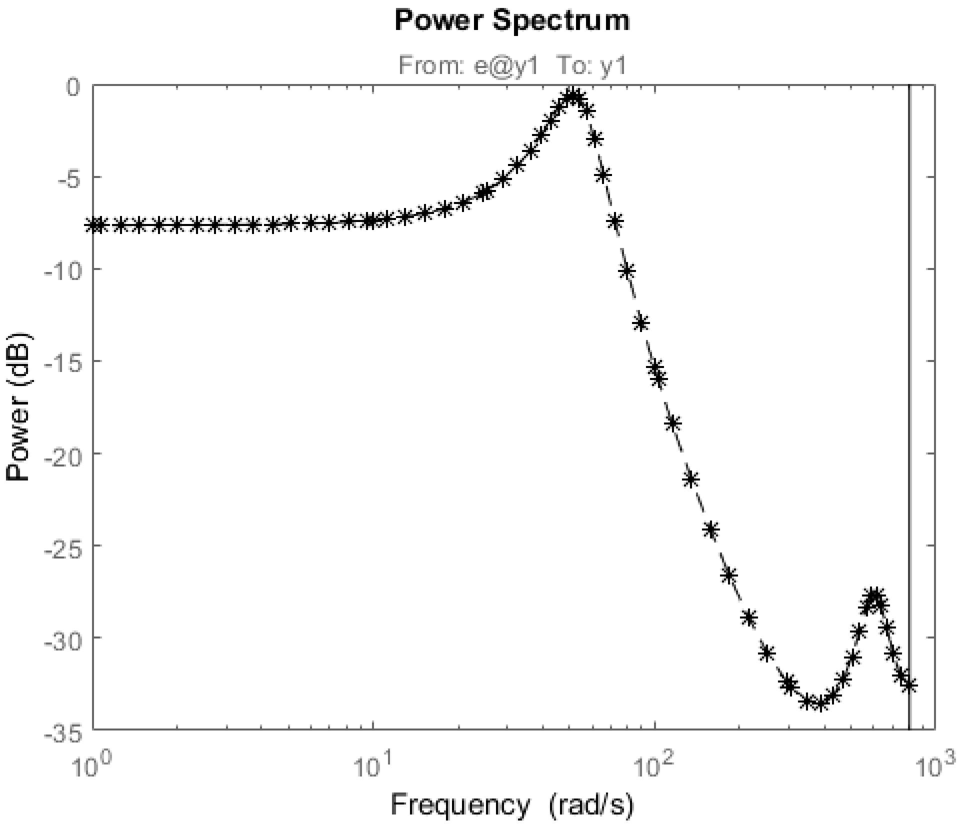
Specify Line Width and Marker Style on Spectrum Plot

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*--');
```



`'k*--'`, specifies a dashed line (`--`) that is black (`k`), with star markers (`*`).

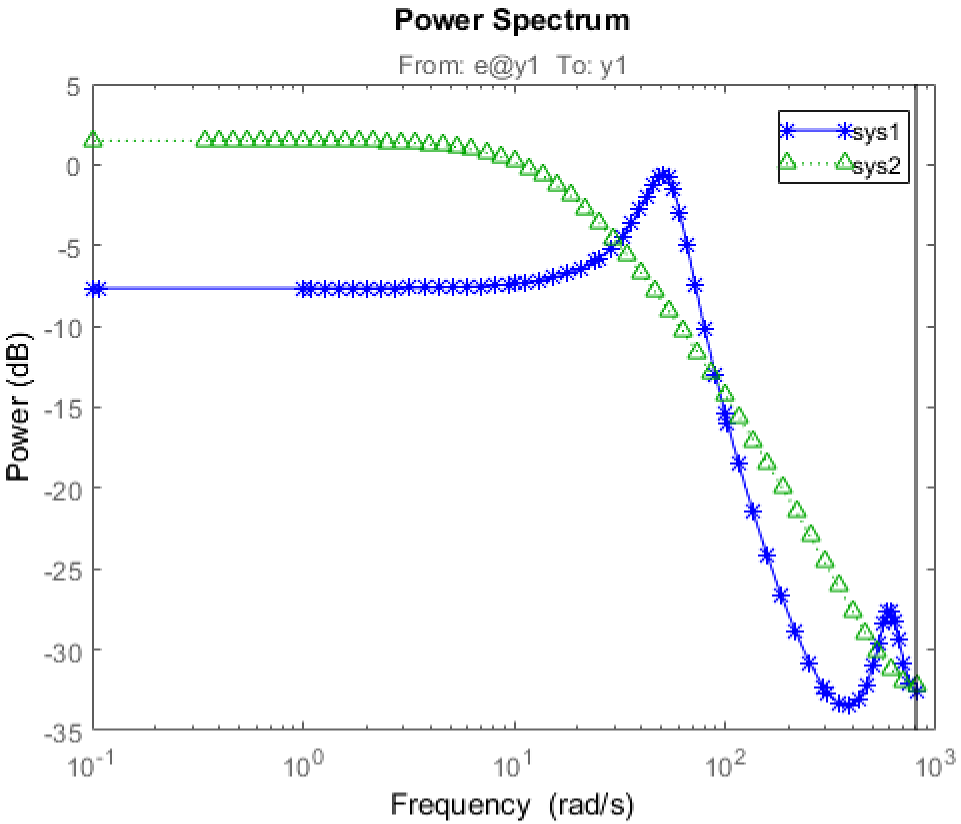
Plot Multiple Models on the Same Axes

Obtain multiple identified models.

```
load iddata9 z9
sys1 = ar(z9,4);
sys2 = ar(z9,2);
```

Plot the output spectrum for both models.

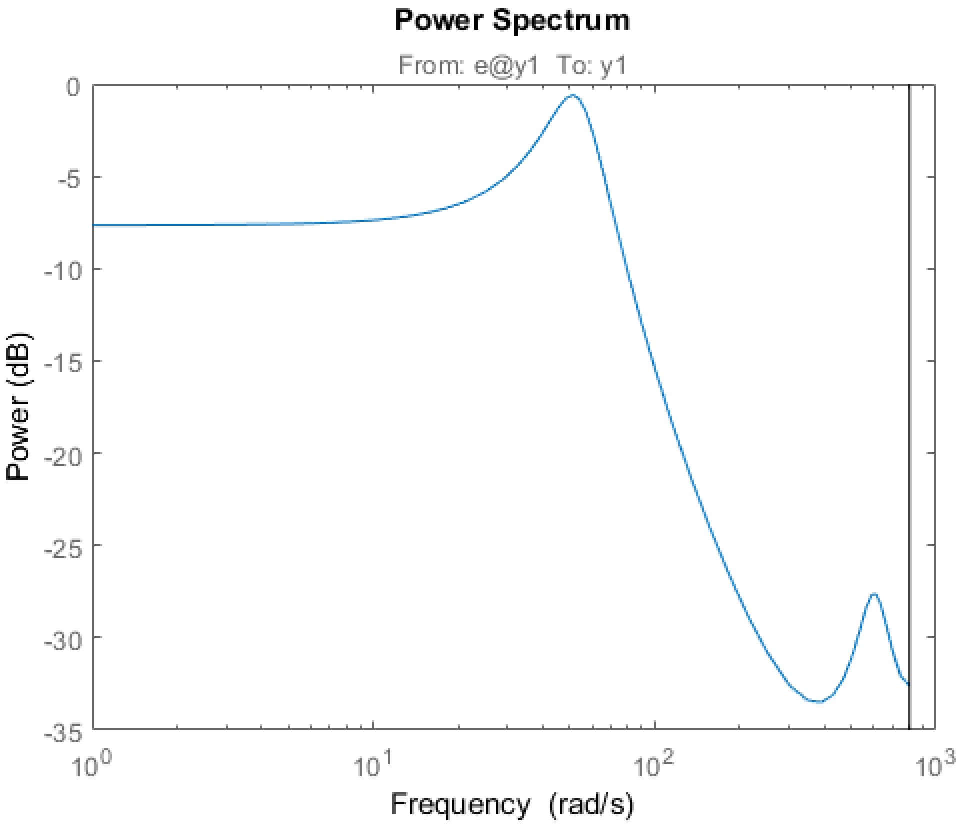
```
spectrumplot(sys1,'b*-',sys2,'g^:');
legend('sys1','sys2');
```



Specify Plot Axes for Spectrum Plot

Obtain the axes handle for a plot.

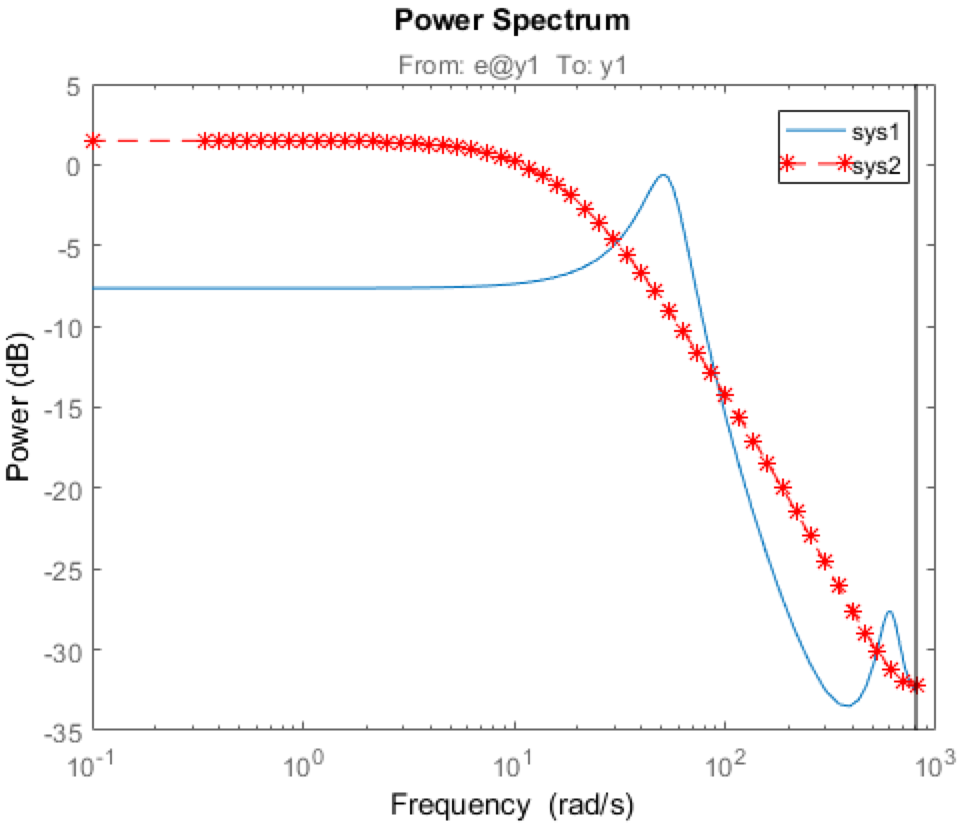
```
load iddata9 z9
sys1 = ar(z9,4);
spectrumplot(sys1);
ax = gca;
```



`ax` is the handle for the spectrum plot axes.

Plot the output spectrum for another model on the specified axes.

```
sys2 = ar(z9,2);

hold on;
spectrumplot(ax,sys2,'r*--');

legend('sys1','sys2');
```



Specify Options for Spectrum Plot

Specify the plot options.

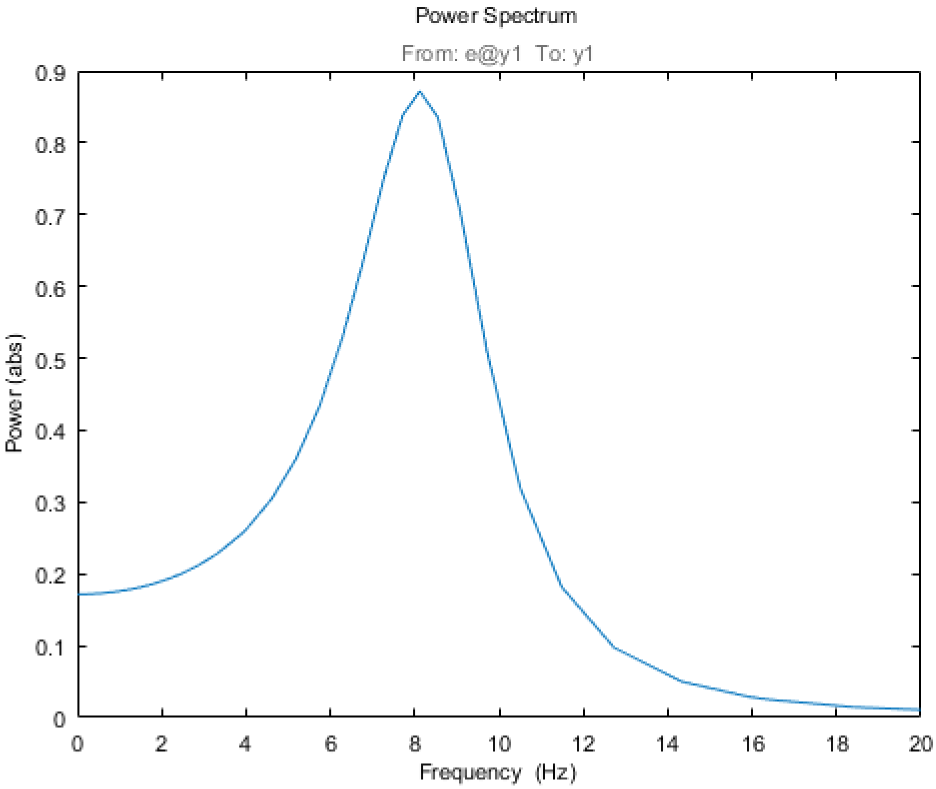
```
plot_options = spectrumoptions;
plot_options.FreqUnits = 'Hz';
plot_options.FreqScale = 'linear';
plot_options.Xlim = {[0 20]};
plot_options.MagUnits = 'abs';
```

Estimate an AR model.

```
load iddata9 z9
sys = ar(z9,4);
```

Plot the output spectrum for the model.

```
spectrumplot(sys,plot_options);
```



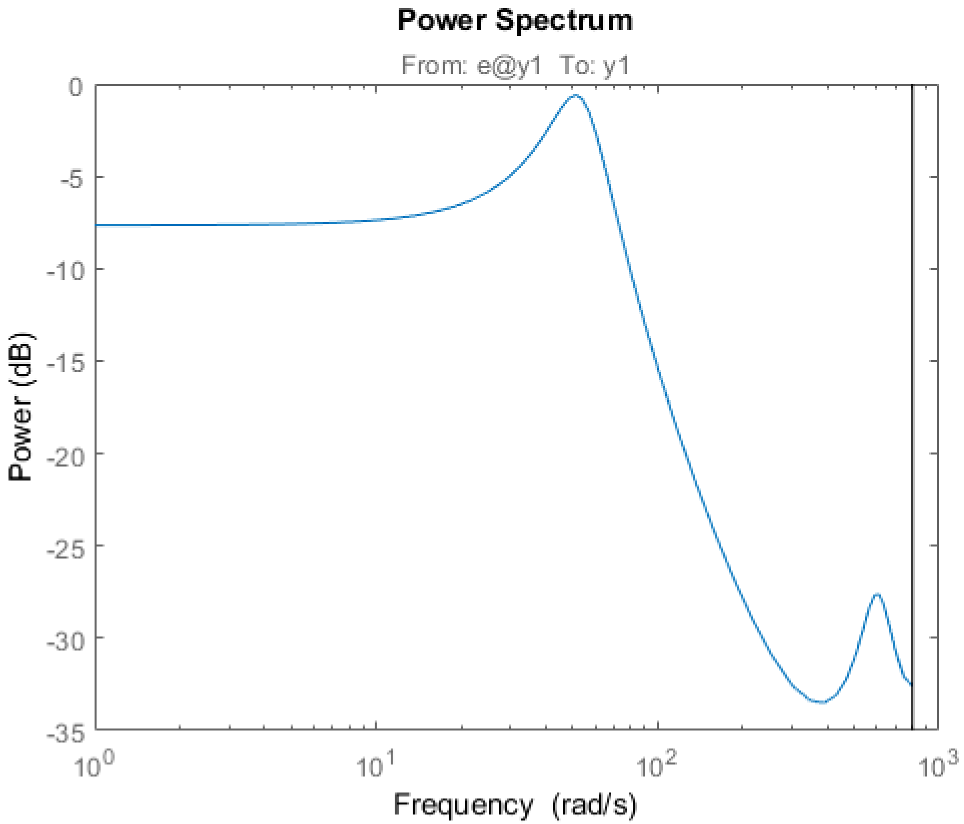
Specify Spectrum Plot Frequency Range

Obtain the identified model.

```
load iddata9 z9
sys = ar(z9,4);
```

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.

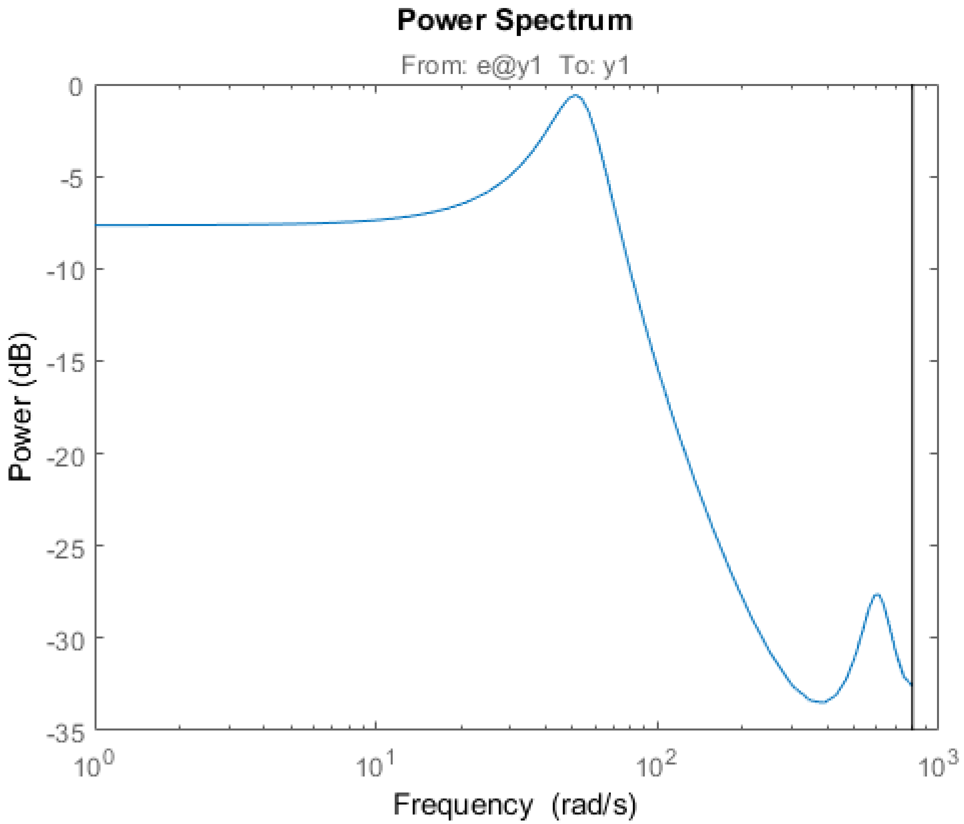
Get Plot Handle for Spectrum Plot Customization

Obtain the identified model.

```
load iddata9 z9
sys = ar(z9,4);
```

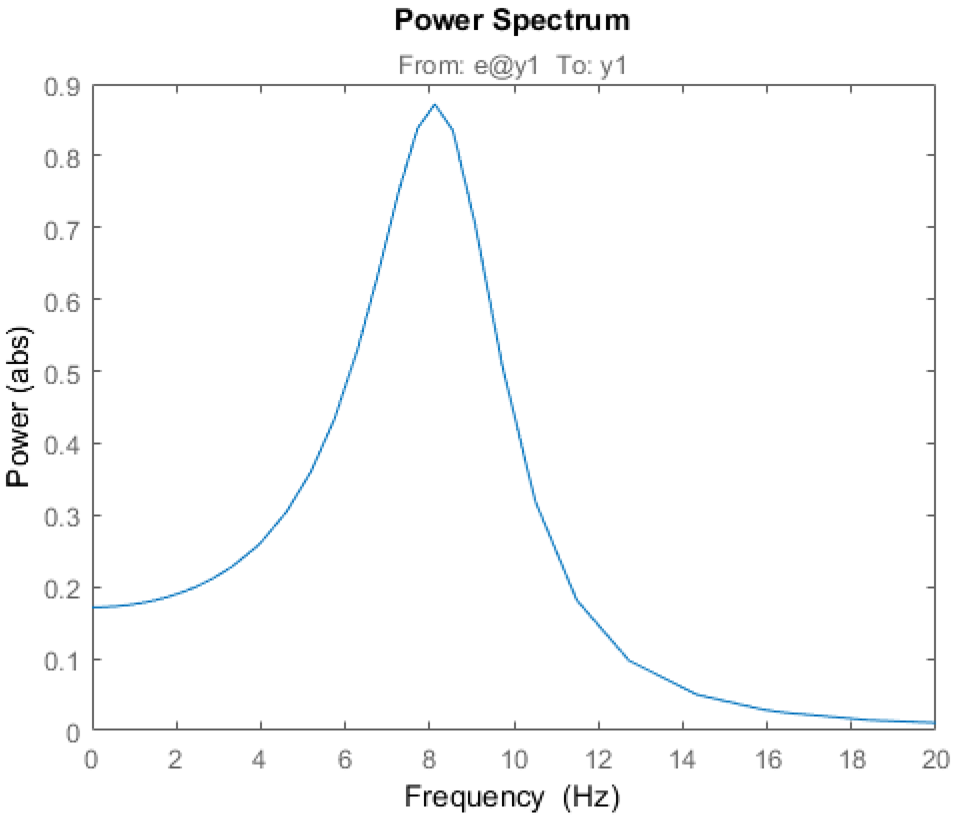
Get the plot handle for the model spectrum plot.

```
h = spectrumplot(sys);
```



(Optional) Specify the plot options, using the plot handle.

```
setoptions(h,'FreqUnits','Hz','FreqScale','linear','Xlim',{[0 20]},'MagUnits','abs');
```



See Also

Axes Properties | Chart Line Properties | `getoptions` | `setoptions` | `showConfidence` | `spectrum` | `spectrumoptions`

Introduced in R2012b

ss2ss

State coordinate transformation for state-space model

Syntax

```
sysT = ss2ss(sys,T)
```

Description

Given a state-space model `sys` with equations

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned}$$

or the innovations form used by the identified state-space (IDSS) models:

$$\begin{aligned} \frac{dx}{dt} &= Ax + Bu + Ke \\ y &= Cx + Du + e \end{aligned}$$

(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.

$$\begin{aligned} \dot{\bar{x}} &= TAT^{-1}\bar{x} + TBu \\ y &= CT^{-1}\bar{x} + Du \end{aligned}$$

or, in the case of an IDSS model:

$$\begin{aligned} \dot{\bar{x}} &= TAT^{-1}\bar{x} + TBu + TKe \\ y &= CT^{-1}\bar{x} + Du + e \end{aligned}$$

`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

`canon` | `balreal`

Introduced before R2006a

ssdata

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` | `tfdata` | `zpkdata`

Introduced before R2006a

ssest

Estimate state-space model using time or frequency domain data

Syntax

```
sys = ssest(data,nx)
sys = ssest(data,nx,Name,Value)
sys = ssest( ___ ,opt)

sys = ssest(data,init_sys)
sys = ssest(data,init_sys,Name,Value)
sys = ssest( ___ ,opt)

[sys,x0] = ssest( ___ )
```

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:

$$\begin{aligned}\dot{x}(t) &= Ax(t) + Bu(t) + Ke(t)\\ y(t) &= Cx(t) + Du(t) + e(t)\end{aligned}$$

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 estimable parameters by default. D is fixed to zero by default, meaning that there is no feedthrough, except for static systems (`nx=0`).

`sys = ssest(data,nx,Name,Value)` estimates the model using the additional options specified by one or more `Name,Value` pair arguments. Use the `Form`, `Feedthrough` and `DisturbanceModel` name-value pair arguments to modify the default behavior of the A, B, C, D, and K matrices.

`sys = ssest( ___ ,opt)` estimates the model using an option set, `opt`, that specifies options such as estimation objective, handling of initial conditions and numerical search method used for estimation.

`sys = ssest(data,init_sys)` estimates a state-space model using the linear system `init_sys` to configure the initial parameterization.

`sys = ssest(data,init_sys,Name,Value)` estimates the model using additional options specified by one or more `Name,Value` pair arguments.

`sys = ssest(___,opt)` estimates the model using an option set, `opt`.

`[sys,x0] = ssest(___)` returns the value of initial states computed during estimation.

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, created using `ssestOptions`, that specifies options including:

- Estimation objective
- Handling of initial conditions
- Numerical search method used for estimation

If `opt` is not specified and `init_sys` is a previously estimated `idss` model, the options from `init_sys.Report.OptionsUsed` are used.

`init_sys`

Linear system that configures the initial parameterization of `sys`.

You obtain `init_sys` by either performing an estimation using measured data or by direct construction.

If `init_sys` is an `idss` model, `ssest` uses the parameter values of `init_sys` as the initial guess for estimating `sys`. For information on how to specify `idss`, see “Estimate State-Space Models with Structured Parameterization”. Constraints on the parameters of `init_sys`, such as fixed coefficients and minimum/maximum bounds are honored in estimating `sys`.

Use the `Structure` property of `init_sys` to configure initial guesses and constraints for the A, B, C, D, and K matrices. For example:

- To specify an initial guess for the A matrix of `init_sys`, set `init_sys.Structure.A.Value` as the initial guess.
- To specify constraints for 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

If `init_sys` is not a state-space (`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.

If `opt` is not specified, and `init_sys` was obtained 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. `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'` — Sample time

`0` (continuous-time) (default) | sample time of data (`data.Ts`) | positive scalar

Sample time, specified as a positive scalar.

For continuous-time models, use `Ts = 0`. For discrete-time models, specify `Ts` as a positive scalar whose value is equal to the data sample time.

`'InputDelay'` — Input delays

`0` (default) | scalar | vector

Input delay for each input channel, specified as a numeric vector. 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 sample time `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. Each entry of this vector is a numerical value that represents 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.

`'Form'` — Type of canonical form

`'free'` (default) | `'modal'` | `'companion'` | `'canonical'`

Type of canonical form of `sys`, specified as 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 the observability canonical form [1].

Use the `Form`, `Feedthrough` and `DisturbanceModel` name-value pair arguments to modify the default behavior of the A, B, C, D, and K matrices.

For more information, see “Estimate State-Space Models with Canonical Parameterization”.

`'Feedthrough'` — Direct feedthrough from input to output

`0` (default) | `1` | logical vector

Direct feedthrough from input to output, specified as a logical vector 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.

Use the `Form`, `Feedthrough` and `DisturbanceModel` name-value pair arguments to modify the default behavior of the *A*, *B*, *C*, *D*, and *K* matrices.

`'DisturbanceModel'` — Specify whether to estimate the *K* matrix

`'estimate'` (default) | `'none'`

Specify whether to estimate the *K* matrix which specifies the noise component, specified as 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.

Use the `Form`, `Feedthrough` and `DisturbanceModel` name-value pair arguments to modify the default behavior of the *A*, *B*, *C*, *D*, and *K* matrices.

Output Arguments

`sys`

Identified state-space model, returned as a `idss` model. This model is created using the specified model orders, delays, and estimation options.

Information about the estimation results and options used is stored in the `Report` property of the model. `Report` has the following fields:

Report Field	Description
`Status`	Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.

Report Field	Description
`Method`	Estimation command used.
`InitialSt`	How initial states were handled during estimation, returned as 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. • Column vector of length *Nx*, where *Nx* is the number of states. For multi-experiment data, a matrix with *Ne* columns, where *Ne* is the number of experiments. • Parametric initial condition object (`x0obj`) created using `idpar`. Only for discrete-time state-space models. This field is especially useful when the `InitialState` option in the estimation option set is `'auto'`.
`N4Weight`	Weighting scheme used for singular-value decomposition by the N4SID algorithm, returned as one of the following values: • `'MOESP'` — Uses the MOESP algorithm. • `'CVA'` — Uses the Canonical Variable Algorithm. • `'SSARX'` — A subspace identification method that uses an ARX estimation based algorithm to compute the weighting. This option is especially useful when the `N4Weight` option in the estimation option set is `'auto'`.
`N4Horizor`	Forward and backward prediction horizons used by the N4SID algorithm, returned as a row vector with three elements — `[r sy su]`, where `r` is the maximum forward prediction horizon. `sy` is the number of past outputs, and `su` is the number of past inputs that are used for the predictions.

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Fit</code></td><td>Quantitative assessment of the estimation, returned as a structure. See “Loss Function and Model Quality Metrics” for more information on these quality metrics. The structure has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>FitPerc</code></td><td>Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as a percentage.</td></tr>
<tr><td><code>LossFcn</code></td><td>Value of the loss function when the estimation completes.</td></tr>
<tr><td><code>MSE</code></td><td>Mean squared error (MSE) measure of how well the response of the model fits the estimation data.</td></tr>
<tr><td><code>FPE</code></td><td>Final prediction error for the model.</td></tr>
<tr><td><code>AIC</code></td><td>Raw Akaike Information Criteria (AIC) measure of model quality.</td></tr>
<tr><td><code>AICc</code></td><td>Small sample-size corrected AIC.</td></tr>
<tr><td><code>nAIC</code></td><td>Normalized AIC.</td></tr>
<tr><td><code>BIC</code></td><td>Bayesian Information Criteria (BIC).</td></tr>
</table></td></tr>
<tr><td><code>Parameter</code></td><td>Estimated values of model parameters.</td></tr>
<tr><td><code>OptionsUs</code></td><td>Option set used for estimation. If no custom options were configured, this is a set of default options. See <code>ssestOptions</code> for more information.</td></tr>
<tr><td><code>RandState</code></td><td>State of the random number stream at the start of estimation. Empty, <code>[]</code>, if randomization was not used during estimation. For more information, see <code>rng</code> in the MATLAB documentation.</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation. Structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
<tr><td><code>Type</code></td><td>Data type. For ARX models, this is set to <code>'Time domain data'</code>.</td></tr>
<tr><td><code>Length</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time. This is equivalent to <code>Data.Ts</code>.</td></tr>
<tr><td><code>InterSam</code></td><td>Input intersample behavior. One of the following values:

<code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
<code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
<code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.

The value of <code>Intersample</code> has no effect on estimation results for discrete-time models.</td></tr>
<tr><td><code>InputOff</code></td><td>Offset removed from time-domain input data during estimation.</td></tr>
<tr><td><code>OutputOf</code></td><td>Offset removed from time-domain output data during estimation.</td></tr>
</table>
</td></tr>
</table>

Report Field	Description
`Terminati`	Termination conditions for the iterative search used for prediction error minimization. Structure with the following fields: <table><tr><th>Field</th><th>Description</th></tr><tr><td>`WhyStop`</td><td>Reason for terminating the numerical search.</td></tr><tr><td>`Iteratio`</td><td>Number of search iterations performed by the estimation algorithm.</td></tr><tr><td>`FirstOrd`</td><td>∞ -norm of the gradient search vector when the search algorithm terminates.</td></tr><tr><td>`FcnCount`</td><td>Number of times the objective function was called.</td></tr><tr><td>`UpdateNo`</td><td>Norm of the gradient search vector in the last iteration. Omitted when the search method is `'lsqnonlin'`.</td></tr><tr><td>`LastImpr`</td><td>Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is `'lsqnonlin'`.</td></tr><tr><td>`Algorith`</td><td>Algorithm used by `'lsqnonlin'` search method. Omitted when other search methods are used.</td></tr></table> For estimation methods that do not require numerical search optimization, the `Termination` field is omitted.

For more information on using `Report`, see “Estimation Report”.

x0

Initial states computed during the estimation.

If `data` contains multiple experiments, then `x0` is an array with each column corresponding to an experiment.

This value is also stored in the `Parameters` field of the model’s `Report` property.

Examples

Determine Optimal Estimated Model Order

Obtain measured input-output data.

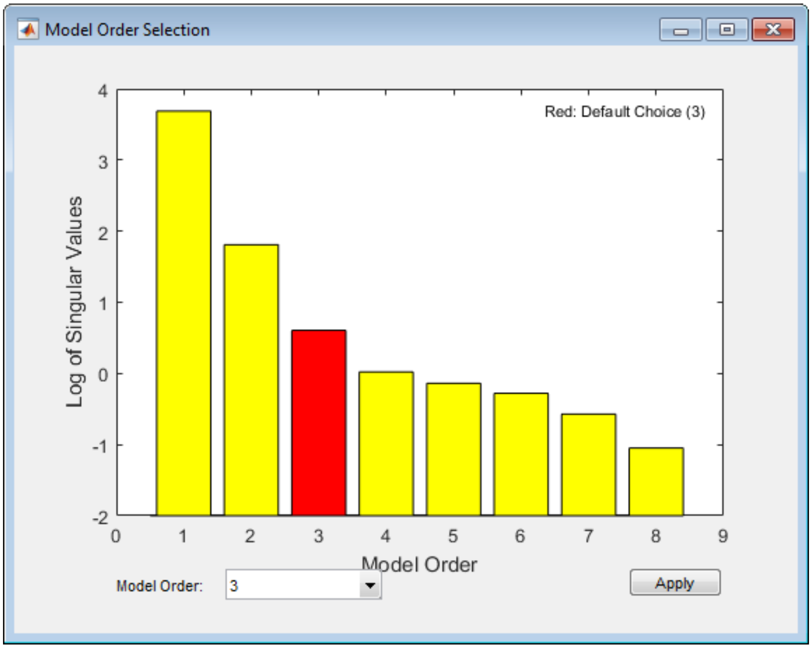
```
load icEngine.mat;
data = iddata(y,u,0.04);
```

`data` is an `iddata` object containing 1500 input-output data samples. The data sample 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 that shows the Hankel singular values (SVD) for models of the orders specified by `nx` appears.



Model Order: 3

States with relatively small Hankel singular values can be safely discarded. The suggested default order choice is 3.

Select the model order in the **Model Order** drop-down list and click **Apply**.

Identify State-Space Model With Input Delay

Load time-domain system response data.

```
load iddata7 z7;
```

Identify a fourth-order state-space model of the data. Specify a known delay of 2 seconds for the first input and 0 seconds for the second input.

```
nx = 4;
sys = ssest(z7(1:300),nx,'InputDelay',[2;0]);
```

Estimate State-Space Model Using Regularization

Obtain a regularized 5th order state-space model for a 2nd order system from a narrow bandwidth signal.

Load estimation data.

```
load regularizationExampleData eData;
```

Create the transfer function model used for generating the estimation data (true system).

```
trueSys = idtf([0.02008 0.04017 0.02008],[1 -1.561 0.6414],1);
```

Estimate an unregularized state-space model.

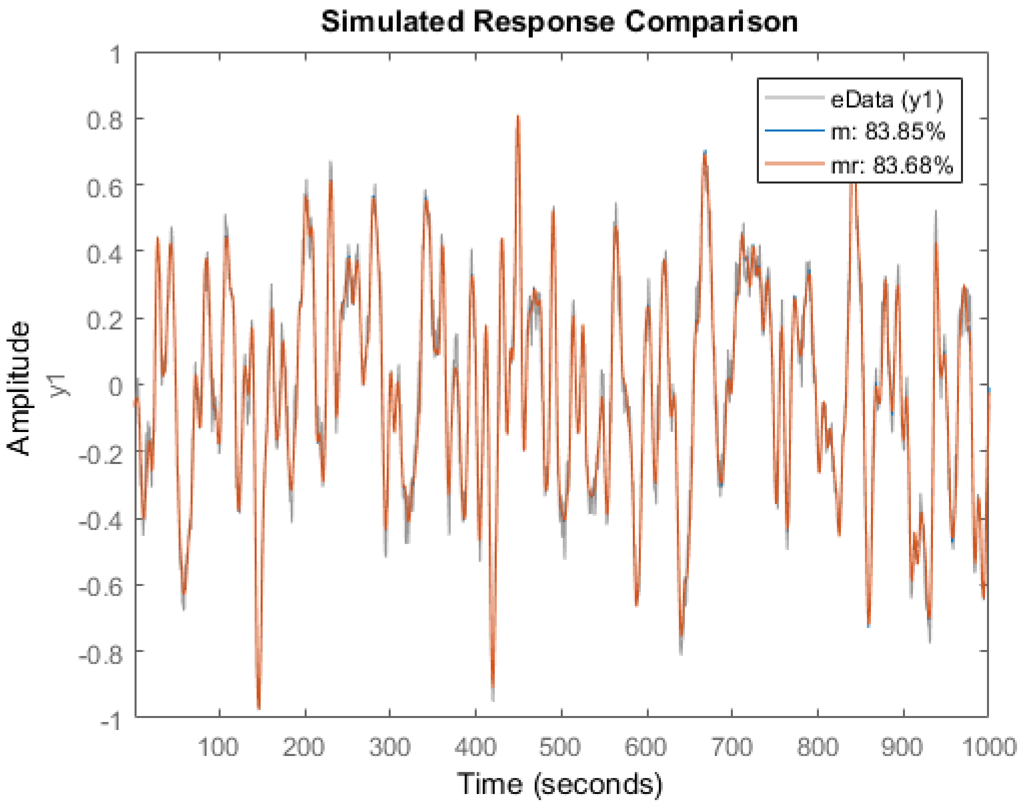
```
opt = ssestOptions('SearchMethod','lm');
m = ssest(eData,5,'form','modal','DisturbanceModel','none','Ts',eData.Ts,opt);
```

Estimate a regularized state-space model.

```
opt.Regularization.Lambda = 10;
mr = ssest(eData,5,'form','modal','DisturbanceModel','none','Ts',eData.Ts,opt);
```

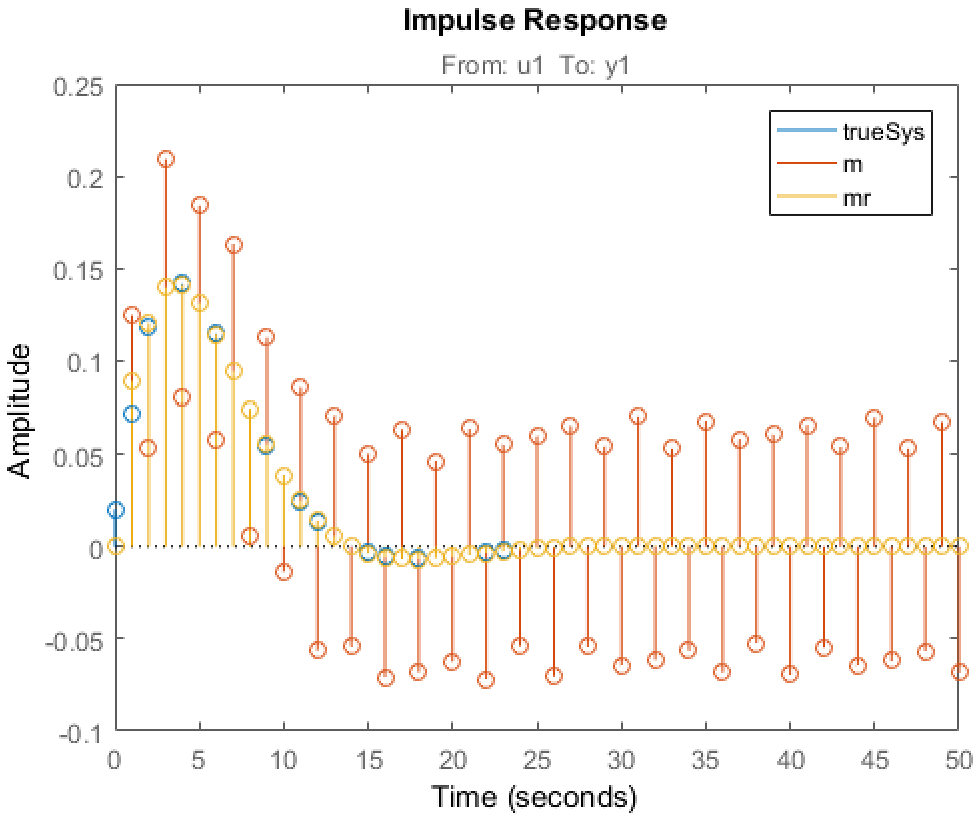
Compare the model outputs with the estimation data.

```
compare(eData,m,mr);
```



Compare the model impulse responses.

```
impulse(trueSys,m,mr,50);
legend('trueSys','m','mr');
```



Estimate Partially Known State-Space Model Using Structured Estimation

Estimate a state-space model of 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.

```
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` to `0` creates an `idss` model with no state disturbance element.

Use the `Structure` property to fix the values of some of the model parameters. Configure the model so that `B` and `K` are fixed, and only the nonzero entries of `A` are estimable.

```
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 (`true`) or fixed (`false`).

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);
```

The estimated parameters of `sys` satisfy the constraints specified by `init_sys`.

Model Order Reduction by Estimation

Consider the Simulink model `idF14Model`. Linearizing this model gives a ninth-order model. However, the dynamics of the model can be captured, without compromising the fit quality too much, using a lower-order model.

Obtain the linearized model.

```
load_system('idF14Model');
io = getlinio('idF14Model');
sys_lin = linearize('idF14Model',io);
```

`sys_lin` is a ninth-order state-space model with two outputs and one input.

Simulate the step response of the linearized model, and use the data to create an `iddata` object.

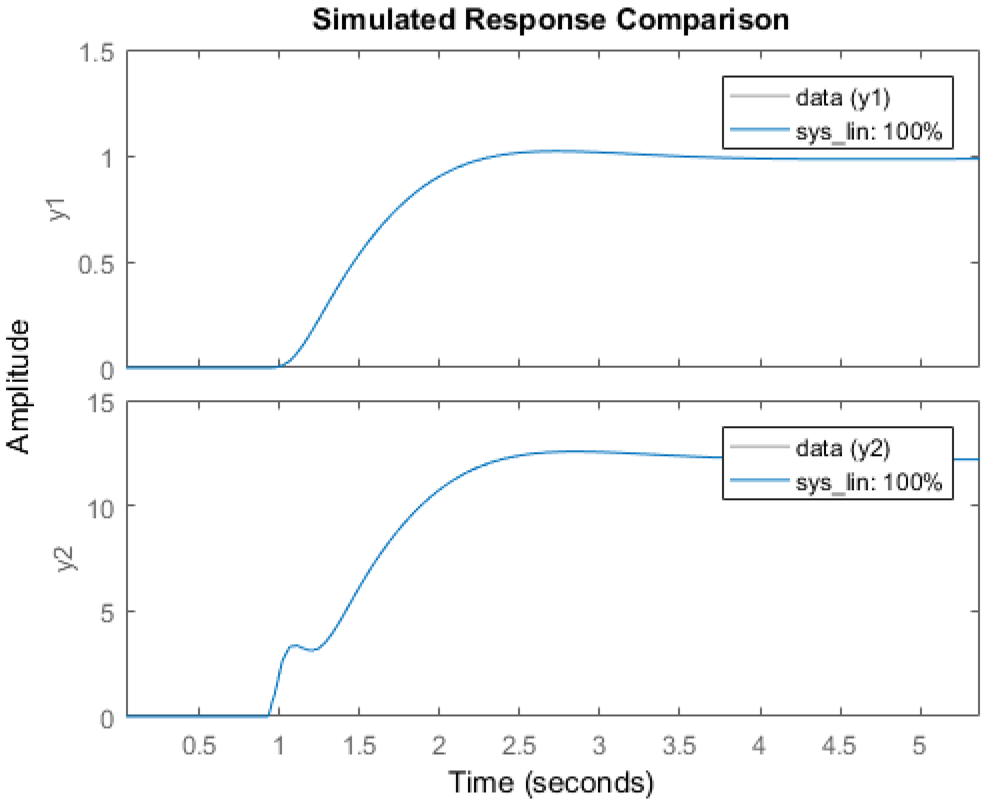
```
Ts = 0.0444;
```

```
t = (0:Ts:4.44)';
y = step(sys_lin,t);

data = iddata([zeros(20,2);y],[zeros(20,1); ones(101,1)],Ts);
```

`data` is an `iddata` object that encapsulates the step response of `sys_lin`.

Compare the data to the model linearization.

```
compare(data,sys_lin);
```



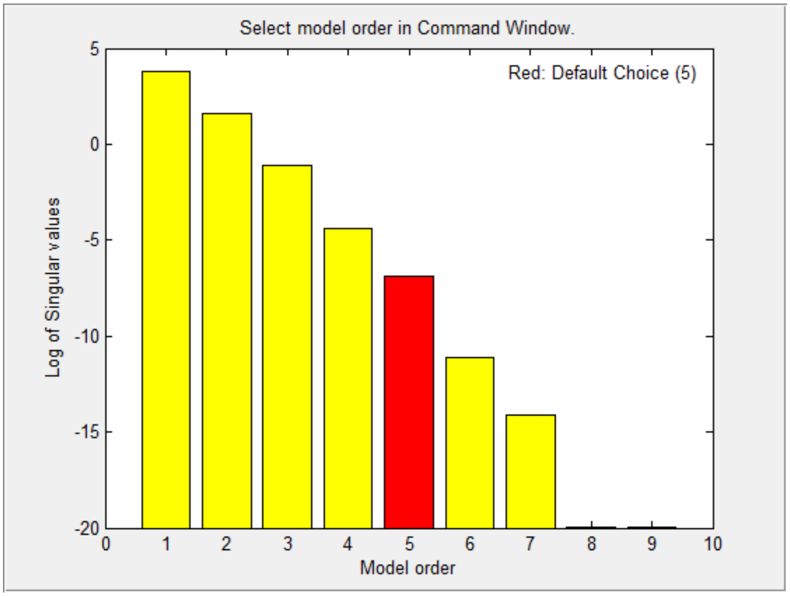
Because the data was obtained by simulating the linearized model, there is a 100% match between the data and model linearization response.

Identify a state-space model with a reduced order that adequately fits the data.

Determine an optimal model order.

```
nx = 1:9;
sys1 = ssest(data,nx,'DisturbanceModel','none');
```

A plot showing the Hankel singular values (SVD) for models of the orders specified by `nx` appears.

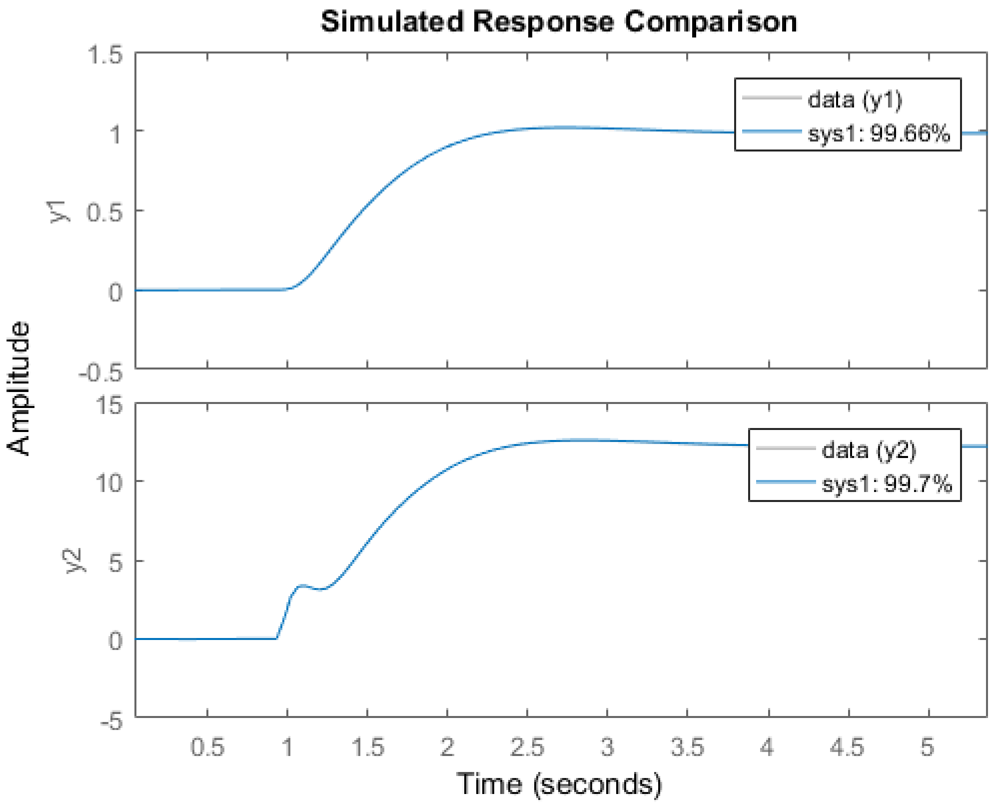


States with relatively small Hankel singular values can be safely discarded. The plot suggests using a fifth-order model.

At the MATLAB command prompt, select the model order for the estimated state-space model. Specify the model order as `5`, or press **Enter** to use the default order value.

Compare the data to the estimated model.

```
compare(data,sys1);
```



`sys1` provides a 98.4% fit for the first output and a 97.7% 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 options set that specifies the `'lm'` search method and allows a maximum of 50 search iterations.

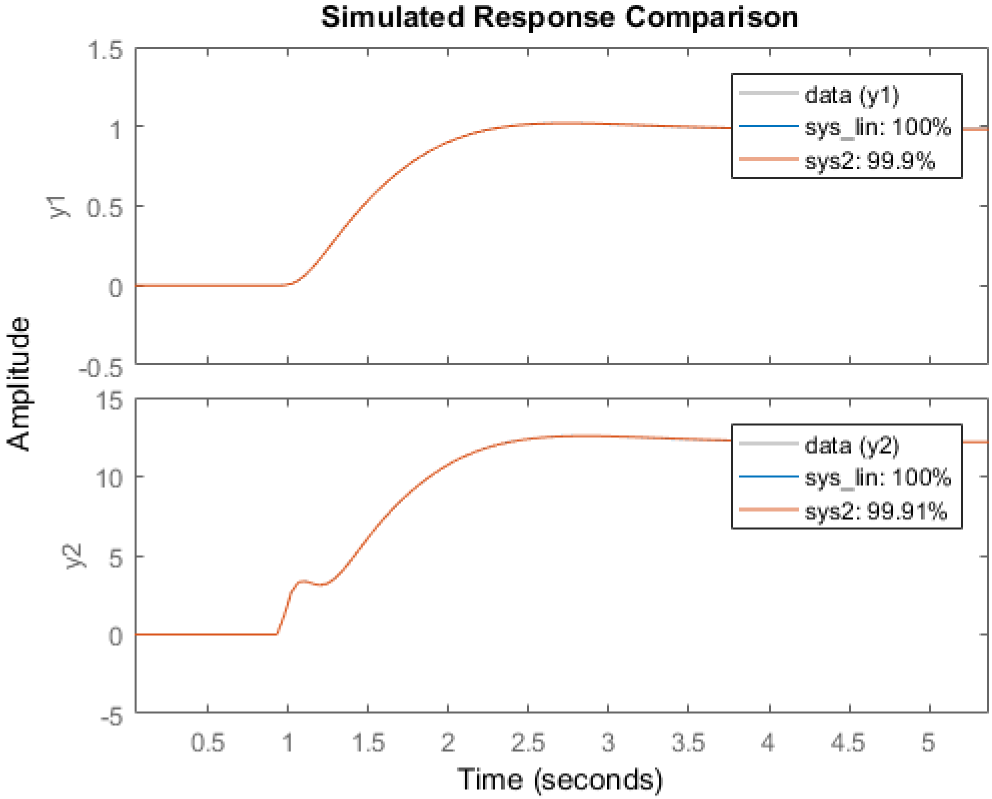
```
opt = ssestOptions('SearchMethod','lm');
opt.SearchOption.MaxIter = 50;
opt.Display = 'on';
```

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% fit for the first output and a 98% fit for the second output while using 4 less states than `sys_lin` .

More About

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 right-most column of the A matrix. For a system with characteristic polynomial

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

the corresponding companion A matrix is

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

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.

Algorithms

`ssest` initializes the parameter estimates using a noniterative subspace approach. It then refines the parameter values using the prediction error minimization approach. See `pem` for more information.

- “What Are State-Space Models?”
- “Supported State-Space Parameterizations”
- “State-Space Model Estimation Methods”
- “Regularized Estimates of Model Parameters”
- “Estimating Models Using Frequency-Domain Data”

References

[1] Ljung, L. *System Identification: Theory For the User*, Second Edition, Upper Saddle River, N.J: Prentice Hall, 1999.

See Also

`canon` | `iddata` | `idfrd` | `idgrey` | `idss` | `n4sid` | `pem` | `polyest` | `procest` | `ssestOptions` | `ssregest` | `tfest`

Introduced in R2012a

ssestOptions

Option set for `ssest`

Syntax

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

Description

`opt = ssestOptions` creates the default option 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. `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'` — Handling of initial states

`'auto'` (default) | `'zero'` | `'estimate'` | `'backcast'` | vector | parametric initial condition object (`x0obj`)

Handling of initial states during estimation, specified as 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.
- Parametric initial condition object (`x0obj`) — Specify initial conditions by using `idpar` 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.

`'N4Weight'` — Weighting scheme used for singular-value decomposition by the N4SID algorithm

`'auto'` (default) | `'MOESP'` | `'CVA'` | `'SSARX'`

Weighting scheme used for singular-value decomposition by the N4SID algorithm, specified as one of the following values:

- `'MOESP'` — Uses the MOESP algorithm by Verhaegen [2].
- `'CVA'` — Uses the Canonical Variable Algorithm by Larimore [1].
- `'SSARX'` — A subspace identification method that uses an ARX estimation based algorithm to compute the weighting.

  Specifying this option allows unbiased estimates when using data that is collected in closed-loop operation. For more information about the algorithm, see [6].
- `'auto'` — The estimating function chooses between the MOESP and CVA algorithms.

`'N4Horizon'` — Forward- and backward-prediction horizons used by the `N4SID` algorithm

`'auto'` (default) | vector `[r sy su]` | k-by-3 matrix

Forward and backward prediction horizons used by the N4SID algorithm, specified as one of the following values:

- A row vector with three elements — `[r sy su]`, where `r` is the maximum forward prediction horizon. 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 [4] for more information. 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`.

`'Focus'` — Error to be minimized

`'prediction'` (default) | `'simulation'`

Error to be minimized in the loss function during estimation, specified as the comma-separated pair consisting of `'Focus'` and one of the following values:

- `'prediction'` — The one-step ahead prediction error between measured and predicted outputs is minimized during estimation. As a result, the estimation focuses on producing a good predictor model.
- `'simulation'` — The simulation error between measured and simulated outputs is minimized during estimation. As a result, the estimation focuses on making a good fit for simulation of model response with the current inputs.

The `Focus` option can be interpreted as a weighting filter in the loss function. For more information, see “Loss Function and Model Quality Metrics”.

`'WeightingFilter'` — Weighting prefilter

`[]` (default) | vector | matrix | cell array | linear system

Weighting prefilter applied to the loss function to be minimized during estimation. To understand the effect of `WeightingFilter` on the loss function, see “Loss Function and Model Quality Metrics”.

Specify `WeightingFilter` as one of the following values:

- `[]` — No weighting prefilter is used.
- Passbands — Specify a row vector or matrix containing frequency values that define desired passbands. You select a frequency band where the fit between estimated model and estimation data is optimized. For example, `[wl,wh]` where `wl` and `wh` represent lower and upper limits of a passband. For a matrix with several rows

defining frequency passbands, `[w1l,w1h;w2l,w2h;w3l,w3h;...]`, the estimation algorithm uses the union of the frequency ranges to define the estimation passband.

Passbands are expressed in `rad/TimeUnit` for time-domain data and in `FrequencyUnit` for frequency-domain data, where `TimeUnit` and `FrequencyUnit` are the time and frequency units of the estimation data.

- SISO filter — Specify a single-input-single-output (SISO) linear filter in one of the following ways:
  - A SISO LTI model
  - `{A,B,C,D}` format, which specifies the state-space matrices of a filter with the same sample time as estimation data.
  - `{numerator,denominator}` format, which specifies the numerator and denominator of the filter as a transfer function with same sample time as estimation data.

    This option calculates the weighting function as a product of the filter and the input spectrum to estimate the transfer function.
- Weighting vector — Applicable for frequency-domain data only. Specify a column vector of weights. This vector must have the same length as 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.

`'EnforceStability'` — Control whether to enforce stability of model

`false` (default) | `true`

Control whether to enforce stability of estimated model, specified as the comma-separated pair consisting of `'EnforceStability'` and either `true` or `false`.

Data Types: `logical`

`'EstCovar'` — Control whether to generate parameter covariance data

`true` (default) | `false`

Controls whether parameter covariance data is generated, specified as `true` or `false`.

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

`'Display'` — Specify whether to display the estimation progress

`'off'` (default) | `'on'`

Specify whether to display the estimation progress, specified as one of the following values:

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

`'InputOffset'` — Removal of offset from time-domain input data during estimation

[ ] (default) | vector of positive integers | matrix

Removal of offset from time-domain input data during estimation, specified as the comma-separated pair consisting of `'InputOffset'` and one of the following:

- A column vector of positive integers of length *Nu*, where *Nu* is the number of inputs.
- `[]` — Indicates no offset.
- *Nu*-by-*Ne* matrix — For multi-experiment data, specify `InputOffset` as an *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.

`'OutputOffset'` — Removal of offset from time-domain output data during estimation

[ ] (default) | vector | matrix

Removal of offset from time-domain output data during estimation, specified as the comma-separated pair consisting of `'OutputOffset'` and one of the following:

- A column vector of length *Ny*, where *Ny* is the number of outputs.
- `[]` — Indicates no offset.
- *Ny*-by-*Ne* matrix — For multi-experiment 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.

`'OutputWeight'` — Weighting of prediction errors in multi-output estimations

[ ] (default) | `'noise'` | positive semidefinite symmetric matrix

Weighting of prediction errors in multi-output estimations, specified as one of the following values:

- `'noise'` — Minimize $\det(E'* E / N)$, where E represents the prediction error and `N` is the number of data samples. This choice is optimal in a statistical sense and leads to maximum likelihood estimates if 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 semidefinite symmetric matrix (`W`) — Minimize the trace of the weighted prediction error matrix `trace(E'*E*W/N)` where:
  - *E* is the matrix of prediction errors, with one column for each output, and *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-output models, or the reliability of corresponding data.
  - `N` is the number of data samples.
- `[]` — The software chooses between the `'noise'` or using the identity matrix for `W`.

This option is relevant for only multi-output models.

`'Regularization'` — Options for regularized estimation of model parameters

structure

Options for regularized estimation of model parameters. For more information on regularization, see “Regularized Estimates of Model Parameters”.

`Regularization` is a structure with the following fields:

- `Lambda` — Constant that determines the bias versus variance tradeoff.

  Specify a positive scalar to add the regularization term to the estimation cost.

  The default value of zero implies no regularization.

  Default: 0
- `R` — Weighting matrix.

  Specify a vector of nonnegative numbers or a square positive semi-definite matrix. The length must be equal to the number of free parameters of the model.

  For black-box models, using the default value is recommended. For structured and grey-box models, you can also specify a vector of `np` positive numbers such that each entry denotes the confidence in the value of the associated parameter.

The default value of 1 implies a value of `eye(npfree)`, where `npfree` is the number of free parameters.

Default: 1

- `Nominal` — The nominal value towards which the free parameters are pulled during estimation.

  The default value of zero implies that the parameter values are pulled towards zero. If you are refining a model, you can set the value to `'model'` to pull the parameters towards the parameter values of the initial model. The initial parameter values must be finite for this setting to work.

  Default: 0

`'SearchMethod'` — Numerical search method used for iterative parameter estimation

`'auto'` (default) | `'gn'` | `'gna'` | `'lm'` | `'grad'` | `'lsqnonlin'`

Numerical search method used for iterative parameter estimation, specified as one of the following values:

- `'gn'` — The subspace Gauss-Newton direction. Singular values of the Jacobian matrix less than `GnPinvConst*eps*max(size(J))*norm(J)` are discarded when computing the search direction. *J* is the Jacobian matrix. The Hessian matrix is approximated by J^TJ. 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 `gamma*max(sv)` of the Hessian are ignored, 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). *gamma* is increased by the factor `LMStep` each time the search fails to find a lower value of the criterion in less than 5 bisections. *gamma* is decreased by the factor `2*LMStep` each time a search is successful without any bisections.
- `'lm'` — Uses the Levenberg-Marquardt method, so that the next parameter value is `-pinv(H+d*I)*grad` from the previous one. *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 the Optimization Toolbox software. This search method can only handle the Trace criterion.

- `'grad'` — The steepest descent gradient search method.
- `'auto'` — The algorithm chooses one of the preceding options. The descent direction is calculated using `'gn'`, `'gna'`, `'lm'`, and `'grad'` successively, at each iteration. The iterations continue until a sufficient reduction in error is achieved.

`'SearchOption'` — Option set for the search algorithm

search option set

Option set for the search algorithm with fields that depend on the value of `SearchMethod`.

SearchOption structure when SearchMethod is specified as 'gn', 'gna', 'lm', 'grad', or 'auto'

Field Name	Description
`Toleran`	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 stop. The estimate of the expected loss-function improvement at the next iteration is based on the Gauss-Newton vector computed for the current parameter value. **Default:** `0.01`
`MaxIter`	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 start-up procedure. Use `sys.Report.Termination.Iterations` to get the actual number of iterations during an estimation, where *sys* is an `idtf` model. **Default:** `20`
`Advance`	Advanced search settings. Specified as a structure with the following fields:

Field Name	Description
`GnPinvCon`	Singular values of the Jacobian matrix that are smaller than `GnPinvConst*max(size(J)*norm(J)*eps)` are discarded

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>when computing the search direction. Applicable when <code>SearchMethod</code> is <code>'gn'</code>.
<code>GnPinvConst</code> must be a positive, real value.
Default: <code>10000</code></td></tr>
<tr><td><code>InitGnaTo</code></td><td>Initial value of <i>gamma</i>. Applicable when <code>SearchMethod</code> is <code>'gna'</code>.
Default: <code>0.0001</code></td></tr>
<tr><td><code>LMStartVa</code></td><td>Starting value of search-direction length <i>d</i> in the Levenberg-Marquardt method. Applicable when <code>SearchMethod</code> is <code>'lm'</code>.
Default: <code>0.001</code></td></tr>
<tr><td><code>LMStep</code></td><td>Size of the Levenberg-Marquardt step. The next value of the search-direction length <i>d</i> in the Levenberg-Marquardt method is <code>LMStep</code> times the previous one. Applicable when <code>SearchMethod</code> is <code>'lm'</code>.
Default: <code>2</code></td></tr>
<tr><td><code>MaxBisect</code></td><td>Maximum number of bisections used by the line search along the search direction.
Default: <code>25</code></td></tr>
<tr><td><code>MaxFunEva</code></td><td>Iterations stop if the number of calls to the model file exceeds this value.
<code>MaxFunEvals</code> must be a positive, integer value.
Default: <code>Inf</code></td></tr>
<tr><td><code>MinParCha</code></td><td>Smallest parameter update allowed per iteration.
<code>MinParChange</code> must be a positive, real value.
Default: <code>0</code></td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td><code>RelImprov</code></td><td>Iterations stop if the relative improvement of the criterion function is less than <code>RelImprovement</code>.
<code>RelImprovement</code> must be a positive, integer value.
Default: <code>0</code></td></tr>
<tr><td><code>StepReduc</code></td><td>Suggested parameter update is reduced by the factor <code>StepReduction</code> after each try. This reduction continues until either <code>MaxBisections</code> tries are completed or a lower value of the criterion function is obtained.
<code>StepReduction</code> must be a positive, real value that is greater than 1.
Default: <code>2</code></td></tr>
</table>
</td></tr>
</table>

SearchOption structure when SearchMethod is specified as 'lsqnonlin'

Field Name	Description
`TolFun`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values. The value of `TolFun` is the same as that of `opt.SearchOption.Advanced.TolFun`. **Default:** `1e-5`
`TolX`	Termination tolerance on the estimated parameter values. The value of `TolX` is the same as that of `opt.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.

Field Name	Description
	The value of `MaxIter` is the same as that of `opt.SearchOption.Advanced.MaxIter`. **Default:** `20`
`Advance`	Options set for `lsqnonlin`. For more information, see the Optimization Options table in “Optimization Options”. Use `optimset('lsqnonlin')` to create an options set for `lsqnonlin`, and then modify it to specify its various options.

`'Advanced'` — Additional advanced options

structure

Additional advanced options, specified as 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 loss function. The standard deviation is estimated robustly as the median of the absolute deviations from the median of the prediction errors, divided by `0.7`. For more information on robust norm choices, see section 15.2 of [4].

  `ErrorThreshold = 0` disables robustification and leads to a purely quadratic loss function. When estimating with frequency-domain data, the software sets `ErrorThreshold` to zero. For time-domain data that contains outliers, try setting `ErrorThreshold` to `1.6`.

  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.

  The initial condition is estimated when

  $$\frac{\left\| y_{p,z} - y_{meas} \right\|}{\left\| y_{p,e} - y_{meas} \right\|} > \text{AutoInitThreshold}$$

  - y_{meas} is the measured output.
  - $y_{p,z}$ is the predicted output of a model estimated using zero initial states.
  - $y_{p,e}$ is the predicted output of a model estimated using estimated initial states.

  Applicable when `InitialState` is `'auto'`.

  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 for `ssest`

`ssestOptions` option set

Option set for `ssest`, returned as an `ssestOptions` option set.

Examples

Create Default Option Set for State Space Estimation

```
opt = ssestOptions;
```

Specify Options for State Space Estimation

Create an option 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';
```

More About

- “Loss Function and Model Quality Metrics”

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.

[6] Jansson, M. “Subspace identification and ARX modeling.” *13th IFAC Symposium on System Identification* , Rotterdam, The Netherlands, 2003.

See Also

`ssest`

Introduced in R2012a

ssform

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

Default:

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 as one of the following values:

- `'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].

Default:

`'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.

Default:

`'DisturbanceModel'`

Specify whether to estimate the noise component of the model, specified as one of the following values:

- `'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.

Default:

Output Arguments

`sys1`

State-space model with configured parameterization, feedthrough, and disturbance dynamics

Examples

Convert 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);
```

- "Estimate State-Space Models at the Command Line"

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.

More About

- "Supported State-Space Parameterizations"

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.

See Also

`idss` | `n4sid` | `ssest`

Introduced in R2012b

ssregest

Estimate state-space model by reduction of regularized ARX model

Syntax

```
sys = ssregest(data,nx)
sys = ssregest(data,nx,Name,Value)
sys = ssregest(___,opt)

[sys,x0] = ssregest(___)
```

Description

`sys = ssregest(data,nx)` estimates a state-space model by reduction of a regularized ARX model.

`sys = ssregest(data,nx,Name,Value)` specifies additional options using one or more `Name,Value` pair arguments.

`sys = ssregest(___,opt)` specifies estimation options that configure the estimation objective, ARX orders, and order reduction options. This syntax can include any of the input argument combinations in the previous syntaxes.

`[sys,x0] = ssregest(___)` returns the value of initial states computed during estimation. This syntax can include any of the input argument combinations in the previous syntaxes.

Examples

Estimate State-Space Model by Reduction of Regularized ARX Model

Load estimation data.

```
load iddata2 z2;
```

`z2` is an `iddata` object that contains time-domain system response data.

Identify a third-order state-space model.

```
sys = ssregest(z2,3);
```

Estimate State-Space Model With Input Delay

Load estimation data.

```
load iddata2 z2
```

Estimate a third-order state-space model with input delay.

```
sys = ssregest(z2,3,'InputDelay',2);
```

Configure the ARX Orders and Estimation Focus

Load estimation data.

```
load iddata2 z2;
```

Specify the order of the regularized ARX model used by the software during estimation. Also, set the estimation focus to simulation.

```
opt = ssregestOptions('ARXOrder',[100 100 1],'Focus','simulation');
```

Identify a third-order state-space model.

```
sys = ssregest(z2,3,opt);
```

Return Initial State Values Computed During Estimation

Load estimation data.

```
load iddata2 z2;
```

Obtain the initial state values when identifying a third-order state-space model.

```
[sys,x0] = ssregest(z2,3);
```

Compare Regularized State-Space Models Estimated Using Impulse Response and Reduction of ARX Models

Load data.

```
load regularizationExampleData eData;
```

Create a transfer function model used for generating the estimation data (true system).

```
trueSys = idtf([0.02008 0.04017 0.02008],[1 -1.561 0.6414],1);
```

Obtain regularized impulse response (FIR) model.

```
opt = impulseestOptions('RegulKernel','DC');
m0 = impulseest(eData,70,opt);
```

Convert the model into a state-space model and reduce the model order.

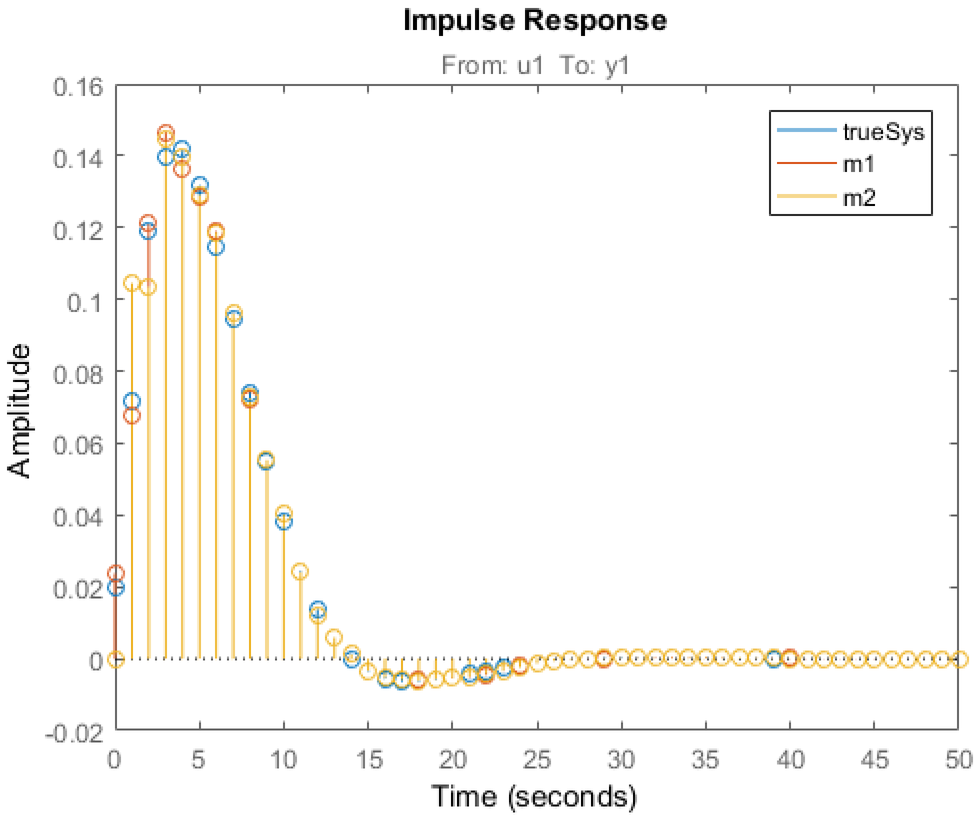
```
m1 = balred(idss(m0),15);
```

Obtain a second state-space model using regularized reduction of an ARX model.

```
m2 = ssregest(eData,15);
```

Compare the impulse responses of the true system and the estimated models.

```
impulse(trueSys,m1,m2,50);
legend('trueSys','m1','m2');
```



Input Arguments

`data` — Estimation data

`iddata` | `idfrd` | `frd`

Estimation data, specified as an `iddata`, `idfrd` or `frd` object.

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'`

  The sample time `Ts` of the `iddata` object must be nonzero.

`nx` — Order of estimated model

positive scalar | positive vector | `'best'`

Order of the estimated model, specified as a positive scalar or vector.

If `nx` is a vector, then `ssregest` creates a plot which you can use to choose a suitable model order. The plot shows the Hankel singular values for models of chosen values in the vector. States with relatively small Hankel singular values can be safely discarded. A default choice is suggested in the plot.

You can also specify `nx = 'best'`, as in `ssregest(data,'best')`, in which case the optimal order is chosen automatically in the 1:10 range.

`opt` — Options set for `ssregest`

`ssregestOptions` options set

Estimation options for `ssregest`, specified as an options set you create using `ssregestOptions`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example: `sys = ssregest(z2,3,'InputDelay',2)` specifies a delay of 2 sampling periods.

`'Ts'` — Sample time

sample time of data (`data.Ts`) (default) | positive scalar | 0

Sample time of the model, specified as 0 or equal to the sample time of `data`.

For continuous-time models, use `Ts = 0`. For discrete-time models, specify `Ts` as a positive scalar whose value is equal to the data sample time.

`'InputDelay'` — Input delays

0 (default) | scalar | vector

Input delay for each input channel, specified as a numeric vector. 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 sample time `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. Each entry of this vector is a numerical value that represents 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.

`'Form'` — Type of canonical form

`'free'` (default) | `'modal'` | `'companion'` | `'canonical'`

Type of canonical form of `sys`, specified as 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 the observability canonical form [1].

Use the `Form`, `Feedthrough` and `DisturbanceModel` name-value pair arguments to modify the default behavior of the A, B, C, D, and K matrices.

`'Feedthrough'` — Direct feedthrough from input to output

`0` (default) | `1` | logical vector

Direct feedthrough from input to output, specified as a logical vector 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.

Use the `Form`, `Feedthrough` and `DisturbanceModel` name-value pair arguments to modify the default behavior of the A, B, C, D, and K matrices.

`'DisturbanceModel'` — Specify whether to estimate the *K* matrix

`'estimate'` (default) | `'none'`

Specify whether to estimate the *K* matrix which specifies the noise component, specified as 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.

Use the `Form`, `Feedthrough` and `DisturbanceModel` name-value pair arguments to modify the default behavior of the *A*, *B*, *C*, *D*, and *K* matrices.

Output Arguments

`sys` — Estimated state-space model

`idss`

Estimated state-space model of order `nx`, returned as an `idss` model object. The model represents:

$$\dot{x}(t) = Ax(t) + Bu(t) + Ke(t)$$
$$y(t) = Cx(t) + Du(t) + e(t)$$

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 estimable parameters by default. *D* is fixed to zero by default, meaning that there is no feedthrough, except for static systems (`nx=0`).

Information about the estimation results and options used is stored in the `Report` property of the model. `Report` has the following fields:

Report Field	Description
`Status`	Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.
`Method`	Estimation command used.
`InitialSt`	Handling of initial states during estimation, returned as one of the following values:

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td></td><td>

- `'zero'` — The initial state was set to zero.
- `'estimate'` — The initial state was treated as an independent estimation parameter.

This field is especially useful when the `InitialState` option in the estimation option set is `'auto'`.</td></tr>
<tr><td>`ARXOrder`</td><td>ARX model orders, returned as a matrix of nonnegative integers `[na nb nk]`.</td></tr>
<tr><td>`Fit`</td><td>Quantitative assessment of the estimation, returned as a structure. See “Loss Function and Model Quality Metrics” for more information on these quality metrics. The structure has the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>`FitPerc`</td><td>Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as a percentage.</td></tr>
<tr><td>`LossFcn`</td><td>Value of the loss function when the estimation completes.</td></tr>
<tr><td>`MSE`</td><td>Mean squared error (MSE) measure of how well the response of the model fits the estimation data.</td></tr>
<tr><td>`FPE`</td><td>Final prediction error for the model.</td></tr>
<tr><td>`AIC`</td><td>Raw Akaike Information Criteria (AIC) measure of model quality.</td></tr>
<tr><td>`AICc`</td><td>Small sample-size corrected AIC.</td></tr>
<tr><td>`nAIC`</td><td>Normalized AIC.</td></tr>
<tr><td>`BIC`</td><td>Bayesian Information Criteria (BIC).</td></tr>
</table></td></tr>
<tr><td>`Parameter`</td><td>Estimated values of model parameters.</td></tr>
<tr><td>`OptionsUs`</td><td>Option set used for estimation. If no custom options were configured, this is a set of default options. See `ssregestOptions` for more information.</td></tr>
<tr><td>`RandState`</td><td>State of the random number stream at the start of estimation. Empty, `[]`, if randomization was not used during estimation. For more information, see `rng` in the MATLAB documentation.</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation, returned as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
<tr><td><code>Type</code></td><td>Data type.</td></tr>
<tr><td><code>Length</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time.</td></tr>
<tr><td><code>InterSam</code></td><td>Input intersample behavior, returned as one of the following values:
• <code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
• <code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
• <code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.</td></tr>
<tr><td><code>InputOff</code></td><td>Offset removed from time-domain input data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
<tr><td><code>OutputOf</code></td><td>Offset removed from time-domain output data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
</table>
</td></tr>
</table>

For more information on using `Report`, see “Estimation Report”.

`x0` — Initial states computed during estimation

scalar | matrix

Initial states computed during estimation, returned as a scalar. If `data` contains multiple experiments, then `x0` is a matrix with each column corresponding to an experiment.

This value is also stored in the `Parameters` field of the model’s `Report` property.

More About

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 right-most column of the A matrix. For a system with characteristic polynomial

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

the corresponding companion A matrix is

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

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.

Tips

- `ssregest` function provides improved accuracy than `n4sid` for short, noisy data sets.
- For some problems, the quality of fit using `n4sid` is sensitive to options, such as `N4Horizon`, whose values can be difficult to determine. In comparison, the quality of fit with `ssregest` is less sensitive to its options, which makes `ssregest` simpler to use.

Algorithms

`ssregest` estimates a regularized ARX model and converts the ARX model to a state-space model. The software then uses balanced model reduction techniques to reduce the state-space model to the specified order.

- “Regularized Estimates of Model Parameters”

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.

See Also

`arx` | `arxRegul` | `balred` | `n4sid` | `ssest` | `ssregestOptions`

Introduced in R2014a

ssregestOptions

Option set for `ssregest`

Syntax

```
options = ssregestOptions
options = ssregestOptions(Name,Value)
```

Description

`options = ssregestOptions` creates a default option set for `ssregest`.

`options = ssregestOptions(Name,Value)` specifies additional options using one or more `Name,Value` pair arguments.

Examples

Create Default Option Set for State-Space Estimation Using Reduction of Regularized ARX Model

```
options = ssregestOptions;
```

Specify Options for State-Space Estimation Using Reduction of Regularized ARX Model

Create an option set for `ssregest` that fixes the value of the initial states to `'zero'`. Also, set the `Display` to `'on'`.

```
opt = ssregestOptions('InitialState','zero','Display','on');
```

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

```
opt = ssregestOptions;
opt.InitialState = 'zero';
```

```
opt.Display = 'on';
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example: `opt = ssregestOptions('InitialState','zero')` fixes the value of the initial states to zero.

`'InitialState'` — Handling of initial states

`'estimate'` (default) | `'zero'`

Handling of initial states during estimation, specified as one of the following values:

- `'zero'` — The initial state is set to zero.
- `'estimate'` — The initial state is treated as an independent estimation parameter.

`'ARXOrder'` — ARX model orders

`'auto'` (default) | matrix of nonnegative integers

ARX model orders, specified as a matrix of nonnegative integers `[na nb nk]`. The `max(ARXOrder)+1` must be greater than the desired state-space model order (number of states). If you specify a value, it is recommended that you use a large value for `nb` order. To learn more about ARX model orders, see `arx`.

`'RegulKernel'` — Regularizing kernel

`'TC'` (default) | `'SE'` | `'SS'` | `'HF'` | `'DI'` | `'DC'`

Regularizing kernel used for regularized estimates of the underlying ARX model, specified as one of the following values:

- `'TC'` — Tuned and correlated kernel
- `'SE'` — Squared exponential kernel
- `'SS'` — Stable spline kernel
- `'HF'` — High frequency stable spline kernel

- `'DI'` — Diagonal kernel
- `'DC'` — Diagonal and correlated kernel

For more information, see [1].

`'Reduction'` — Options for model order reduction

structure

Options for model order reduction, specified as a structure with the following fields:

- `StateElimMethod`

  State elimination method. Specifies how to eliminate the weakly coupled states (states with smallest Hankel singular values). Specified as one of the following values:

`'MatchDC'`	Discards the specified states and alters the remaining states to preserve the DC gain.
`'Truncate'`	Discards the specified states without altering the remaining states. This method tends to product a better approximation in the frequency domain, but the DC gains are not guaranteed to match.

  Default: `'Truncate'`
- `AbsTol, RelTol`

  Absolute and relative error tolerance for stable/unstable decomposition. Positive scalar values. For an input model G with unstable poles, the reduction algorithm of `ssregest` first extracts the stable dynamics by computing the stable/unstable decomposition $G \rightarrow GS + GU$. The `AbsTol` and `RelTol` tolerances control the accuracy of this decomposition by ensuring that the frequency responses of G and $GS + GU$ differ by no more than `AbsTol` + `RelTol`*abs(G). Increasing these tolerances helps separate nearby stable and unstable modes at the expense of accuracy. See `stabsep` for more information.

  Default: `AbsTol = 0; RelTol = 1e-8`
- `Offset`

  Offset for the stable/unstable boundary. Positive scalar value. In the stable/unstable decomposition, the stable term includes only poles satisfying

  - `Re(s) < -Offset * max(1,|Im(s)|)` (Continuous time)

- `|z| < 1 - Offset` (Discrete time)

  Increase the value of `Offset` to treat poles close to the stability boundary as unstable.

  Default: `1e-8`

`'Focus'` — Error to be minimized

`'prediction'` (default) | `'simulation'`

Error to be minimized in the loss function during estimation, specified as the comma-separated pair consisting of `'Focus'` and one of the following values:

- `'prediction'` — The one-step ahead prediction error between measured and predicted outputs is minimized during estimation. As a result, the estimation focuses on producing a good predictor model.
- `'simulation'` — The simulation error between measured and simulated outputs is minimized during estimation. As a result, the estimation focuses on making a good fit for simulation of model response with the current inputs.

The `Focus` option can be interpreted as a weighting filter in the loss function. For more information, see “Loss Function and Model Quality Metrics”.

`'WeightingFilter'` — Weighting prefilter

`[]` (default) | vector | matrix | cell array | linear system

Weighting prefilter applied to the loss function to be minimized during estimation. To understand the effect of `WeightingFilter` on the loss function, see “Loss Function and Model Quality Metrics”.

Specify `WeightingFilter` as one of the following values:

- `[]` — No weighting prefilter is used.
- Passbands — Specify a row vector or matrix containing frequency values that define desired passbands. You select a frequency band where the fit between estimated model and estimation data is optimized. For example, `[wl,wh]` where `wl` and `wh` represent lower and upper limits of a passband. For a matrix with several rows defining frequency passbands, `[w1l,w1h;w2l,w2h;w3l,w3h;...]`, the estimation algorithm uses the union of the frequency ranges to define the estimation passband.

  Passbands are expressed in `rad/TimeUnit` for time-domain data and in `FrequencyUnit` for frequency-domain data, where `TimeUnit` and `FrequencyUnit` are the time and frequency units of the estimation data.

- SISO filter — Specify a single-input-single-output (SISO) linear filter in one of the following ways:
  - A SISO LTI model
  - `{A,B,C,D}` format, which specifies the state-space matrices of a filter with the same sample time as estimation data.
  - `{numerator,denominator}` format, which specifies the numerator and denominator of the filter as a transfer function with same sample time as estimation data.

    This option calculates the weighting function as a product of the filter and the input spectrum to estimate the transfer function.
- Weighting vector — Applicable for frequency-domain data only. Specify a column vector of weights. This vector must have the same length as 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.

`'EstCovar'` — Control whether to generate parameter covariance data

`true` (default) | `false`

Controls whether parameter covariance data is generated, specified as `true` or `false`.

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

`'Display'` — Specify whether to display the estimation progress

`'off'` (default) | `'on'`

Specify whether to display the estimation progress, specified as one of the following values:

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

`'InputOffset'` — Removal of offset from time-domain input data during estimation

`[]` (default) | vector of positive integers | matrix

Removal of offset from time-domain input data during estimation, specified as the comma-separated pair consisting of `'InputOffset'` and one of the following:

- A column vector of positive integers of length *Nu*, where *Nu* is the number of inputs.
- `[]` — Indicates no offset.
- *Nu*-by-*Ne* matrix — For multi-experiment data, specify `InputOffset` as an *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.

`'OutputOffset'` — Removal of offset from time-domain output data during estimation

[ ] (default) | vector | matrix

Removal of offset from time-domain output data during estimation, specified as the comma-separated pair consisting of `'OutputOffset'` and one of the following:

- A column vector of length *Ny*, where *Ny* is the number of outputs.
- `[]` — Indicates no offset.
- *Ny*-by-*Ne* matrix — For multi-experiment 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.

`'OutputWeight'` — Weight of prediction errors in multi-output estimation

[ ] (default) | positive semidefinite, symmetric matrix

Weight of prediction errors in multi-output estimation, specified as one of the following values:

- Positive semidefinite, symmetric matrix (`W`). The software minimizes the trace of the weighted prediction error matrix `trace(E'*E*W/N)` where:
  - `E` is the matrix of prediction errors, with one column for each output, and `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-output models, or the reliability of corresponding data.
  - `N` is the number of data samples.
- `[]` — No weighting is used. Specifying as `[]` is the same as `eye(Ny)`, where `Ny` is the number of outputs.

This option is relevant only for multi-output models.

`'Advanced'` — Advanced estimation options

structure

Advanced options for regularized estimation, specified as a structure with the following fields:

- `MaxSize` — Maximum allowable size of Jacobian matrices formed during estimation, specified as a large positive number.

  Default: `250e3`
- `SearchMethod` — Search method for estimating regularization parameters, specified as one of the following values:
  - `'gn'`: Quasi-Newton line search.
  - `'fmincon'`: Trust-region-reflective constrained minimizer. In general, `'fmincon'` is better than `'gn'` for handling bounds on regularization parameters that are imposed automatically during estimation. Requires Optimization Toolbox software.

  Default: `'gn'`

  If you have the Optimization Toolbox software, the default is `'fmincon'`.

Output Arguments

`options` — Option set for `ssregest`

`ssregestOptions` options set

Estimation options for `ssregest`, returned as an `ssregestOptions` option set.

More About

- “Loss Function and Model Quality Metrics”

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

`ssregest`

Introduced in R2014a

stack

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
sysc{i} = ssest(z1,i-1,opt);
end
sysArray = stack(1, sysc{:});
bode(sysArray);
```

Introduced in R2012a

step

Step response plot of dynamic system; step response data

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-time, 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 = Tfinal`. Express `Tfinal` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time systems with unspecified sample time (`Ts = -1`), `step` interprets `Tfinal` 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 `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 "Algorithms" on page 1-1479). The `step` command always applies the step input at `t=0`, regardless of `Ti`.

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

$$(\textit{length of } t) \times (\textit{number of outputs}) \times (\textit{number of inputs})$$

and `y(:,:,j)` gives the response to a unit step command injected in the `j`th input channel. Similarly, the dimensions of `x` are

$$(\textit{length of } t) \times (\textit{number of states}) \times (\textit{number of inputs})$$

For identified models (see `idlti` and `idnlmodlel`) `[y,t,x,ysd] = 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`.

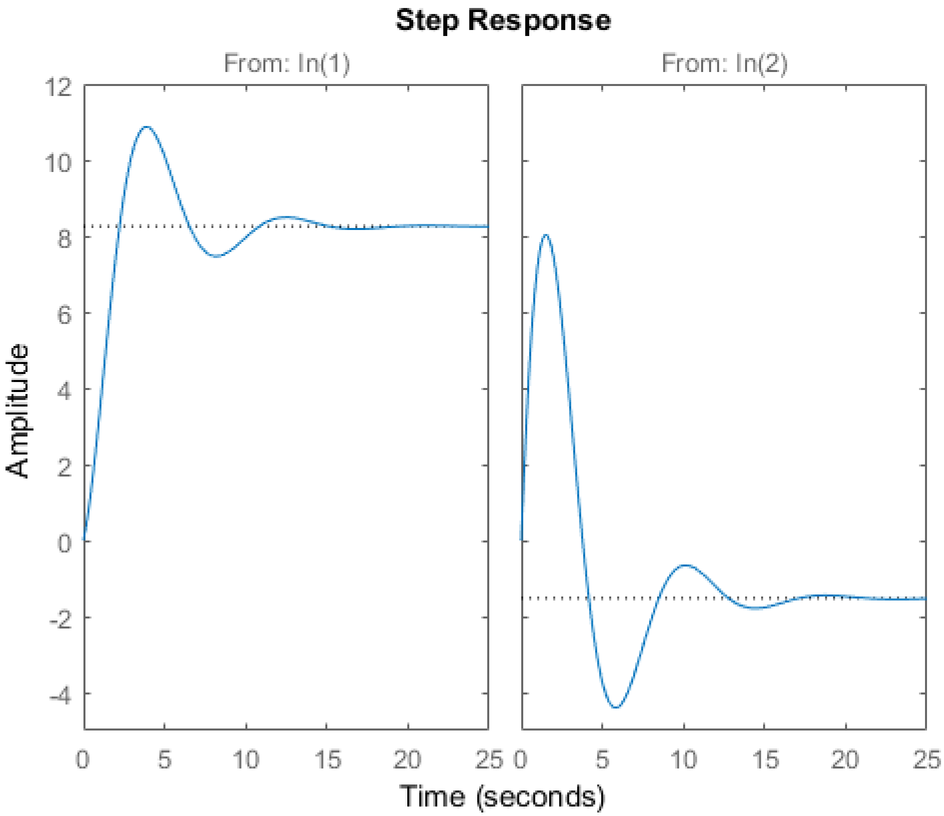
Examples

Step Response Plot of Dynamic System

Plot the step response of the following second-order state-space model:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -0.5572 & -0.7814 \\ 0.7814 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$
$$y = \begin{bmatrix} 1.9691 & 6.4493 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

```
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)
```

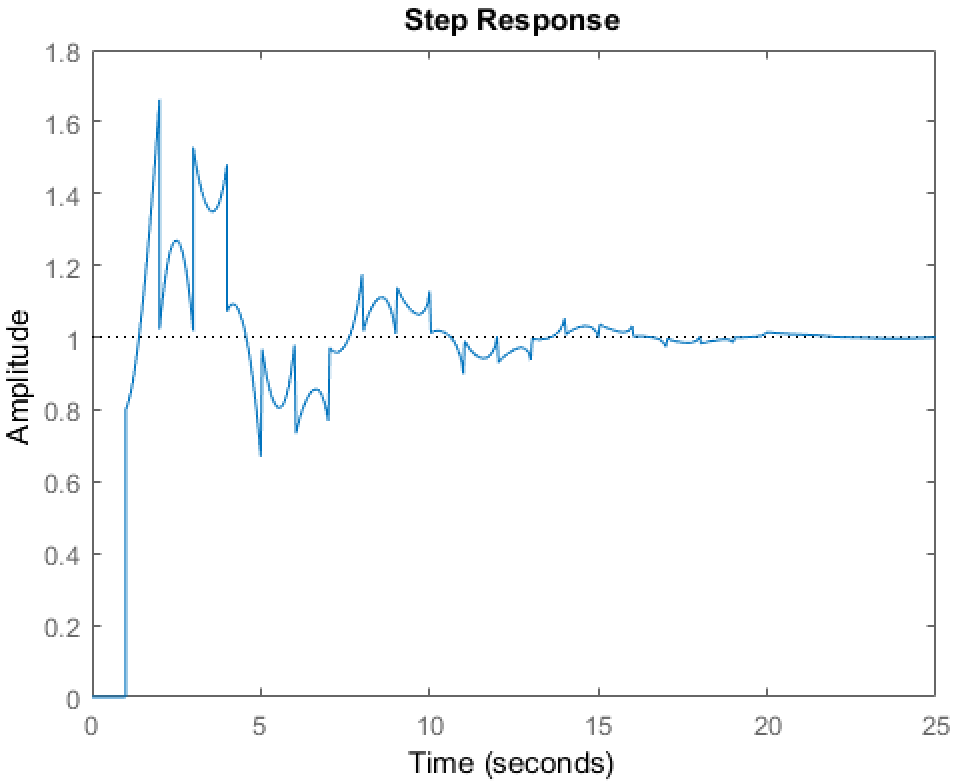


The left plot shows the step response of the first input channel, and the right plot shows the step response of the second input channel.

Step Response Plot of Feedback Loop with Delay

Create a feedback loop with delay and plot its step response.

```
s = tf('s');
G = exp(-s) * (0.8*s^2+s+2)/(s^2+s);
T = feedback(ss(G),1);
step(T)
```



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.

Step Responses of Identified Models with Confidence Regions

Compare the step response of a parametric identified model to a non-parametric (empirical) model. Also view their 3 σ confidence regions.

Load the data.

```
load iddata1 z1
```

Esimate a parametric model.

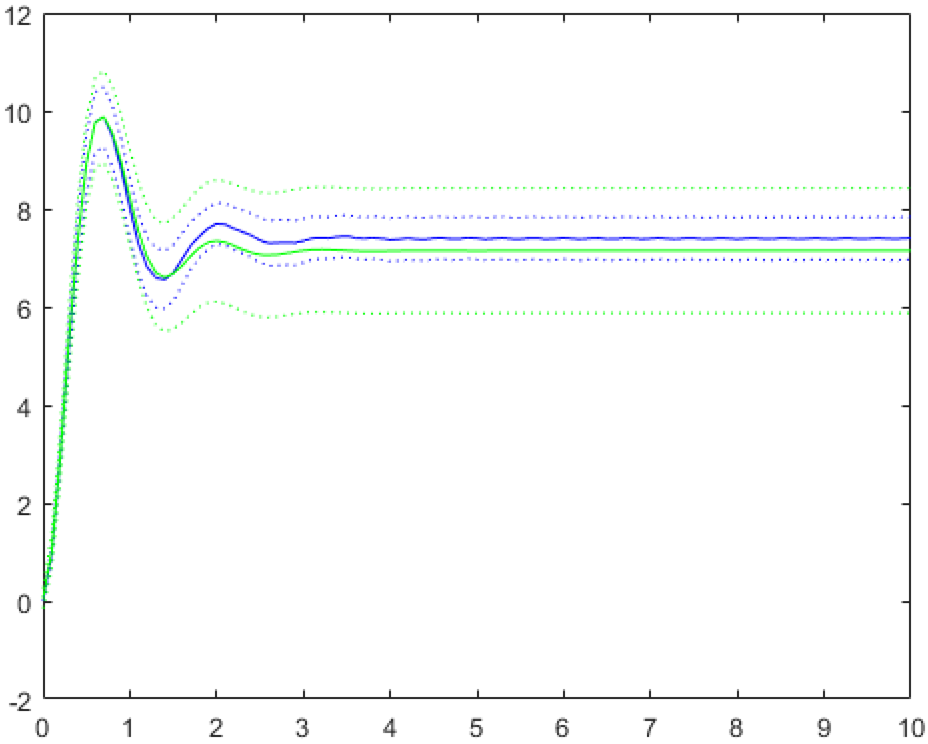
```
sys1 = ssest(z1,4);
```

Estimate a non-parametric model.

```
sys2 = impulseest(z1);
```

Plot the step responses for comparision.

```
t = (0:0.1:10)';
[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:')
```



Validate Linearization of Identified Nonlinear ARX Model

Validate the linearization of a nonlinear ARX model by comparing the small amplitude step responses of the linear and nonlinear models.

Load the data.

```
load iddata2 z2;
```

Estimate a nonlinear ARX model.

```
nlsys = nlarx(z2,[4 3 10],'tree','custom',{'sin(y1(t-2)*u1(t))+y1(t-2)*u1(t)+u1(t).*u1
```

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);
```

Validate the usefulness of `sys` by comparing its small-amplitude step response to that of `nlsys`.

The nonlinear system `nlsys` is operating at an equilibrium level dictated by `(u0, y0)`. Introduce a step perturbation of size 0.1 about this steady-state and compute the corresponding response.

```
opt = stepDataOptions;
opt.InputOffset = u0;
opt.StepAmplitude = 0.1;
t = (0:0.1:10)';
ynl = step(nlsys, t, opt);
```

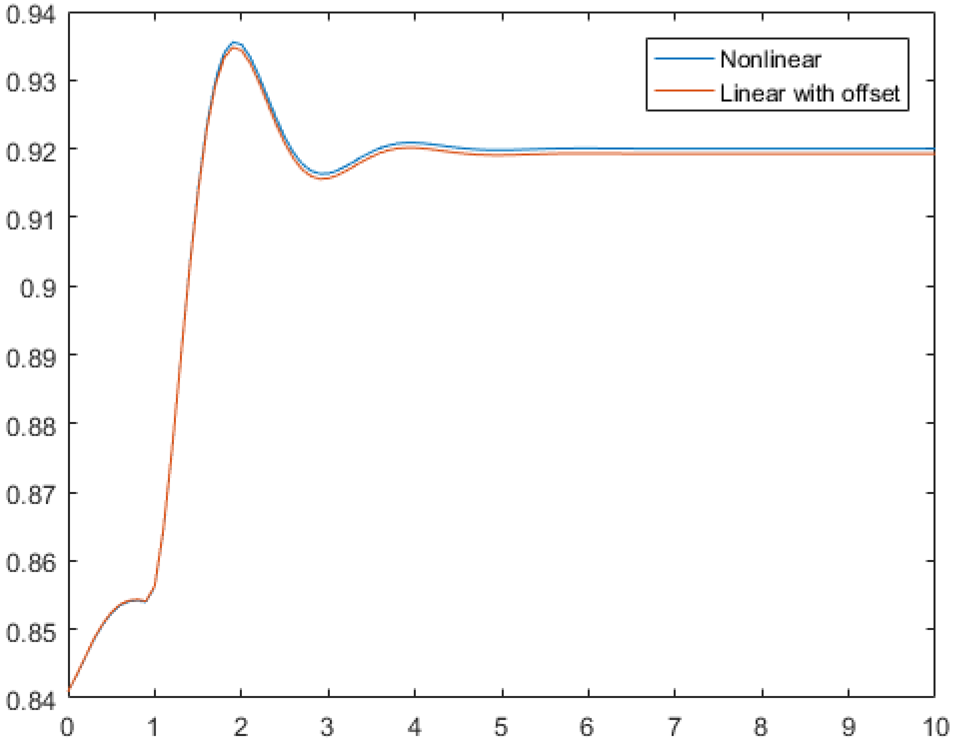
The linear system `sys` expresses the relationship between the perturbations in input to the corresponding perturbation in output. It is unaware of the nonlinear system's equilibrium values.

Plot the step response of the linear system.

```
opt = stepDataOptions;
opt.StepAmplitude = 0.1;
yl = step(sys, t, opt);
```

Add the steady-state offset, `y0` , to the response of the linear system and plot the responses.

```
plot(t, ynl, t, yl+y0)
legend('Nonlinear', 'Linear with offset')
```



Step Response of Identified Time-Series Model

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 the data.

```
load iddata9;
```

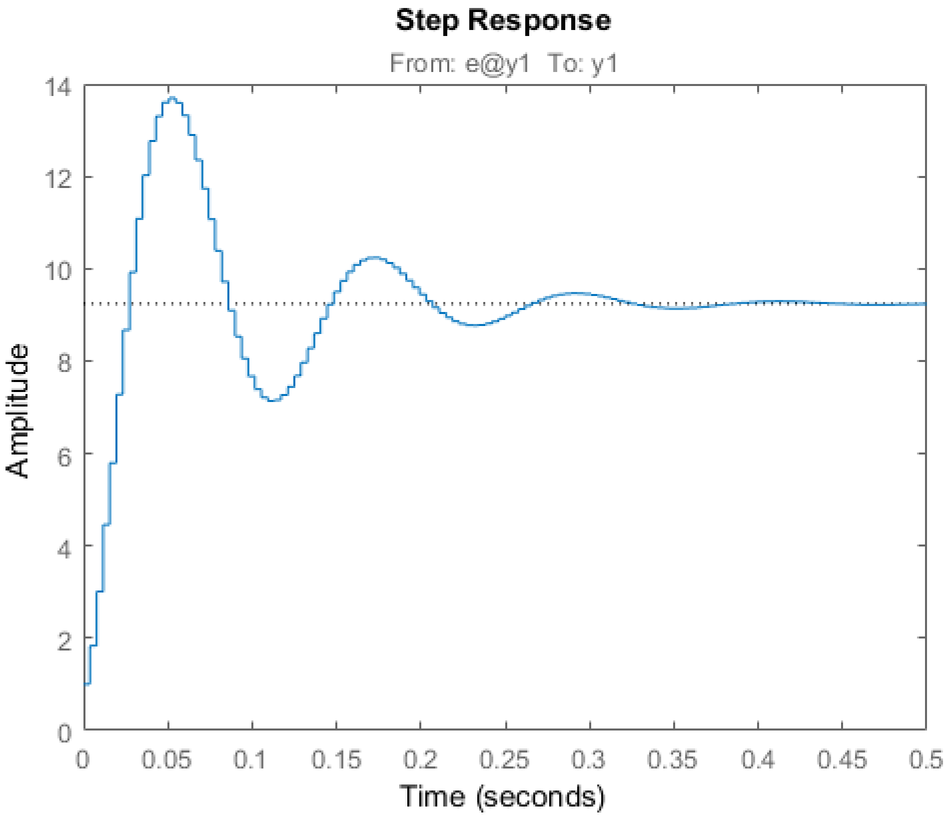
Estimate a time-series model.

```
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`.

Plot the step response.

```
step(sys)
```



More About

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”.

Algorithms

Continuous-time models without internal delays are converted to state space and discretized using zero-order hold on the inputs. The sample time 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` | `initial` | Linear System Analyzer | `lsim` | `stepDataOptions`

Introduced before R2006a

step

Update model parameters and output online using recursive estimation algorithm

Syntax

`[EstimatedParameters,EstimatedOutput] = step(obj,y,InputData)`

Description

`[EstimatedParameters,EstimatedOutput] = step(obj,y,InputData)` updates parameters and output of the model specified in System object, `obj`, using measured output, `y`, and input data.

`step` puts the object into a locked state. In a locked state you cannot change any nontunable properties of the object, such as model order, data type, or estimation algorithm.

The `EstimatedParameters` and `InputData` depend on the online estimation System object:

- `recursiveAR` — `step` returns the estimated polynomial $A(q)$ coefficients of a single output AR model using time-series output data.
  `[A,EstimatedOutput] = step(obj,y)`
- `recursiveARMA` — `step` returns the estimated polynomial $A(q)$ and $C(q)$ coefficients of a single output ARMA model using time-series output data, y.
  `[A,C,EstimatedOutput] = step(obj,y)`
- `recursiveARX` — `step` returns the estimated polynomial $A(q)$ and $B(q)$ coefficients of a SISO or MISO ARX model using measured input and output data, u and y, respectively.
  `[A,B,EstimatedOutput] = step(obj,y,u)`.
- `recursiveARMAX` — `step` returns the estimated polynomial $A(q)$, $B(q)$, and $C(q)$ coefficients of a SISO ARMAX model using measured input and output data, u and y, respectively.
  `[A,B,C,EstimatedOutput] = step(obj,y,u)`.

- `recursiveOE` — `step` returns the estimated polynomial $B(q)$, and $F(q)$ coefficients of a SISO Output-Error polynomial model using measured input and output data, u and y, respectively.
  `[B,F,EstimatedOutput] = step(obj,y,u)`.
- `recursiveBJ` — `step` returns the estimated polynomial $B(q)$, $C(q)$, $D(q)$, and $F(q)$ coefficients of a SISO Box-Jenkins polynomial model using measured input and output data, u and y, respectively.
  `[B,C,D,F,EstimatedOutput] = step(obj,y,u)`.
- `recursiveLS` — `step` returns the estimated system parameters, θ, of a single output system that is linear in estimated parameters, using regressors H and output data y.
  `[theta,EstimatedOutput] = step(obj,y,H)`.

Examples

Estimate an ARMAX Model Online

Create a System object for online parameter estimation of an ARMAX model.

```
obj = recursiveARMAX;
```

The ARMAX model has a default structure with polynomials of order 1 and initial polynomial coefficient values, `eps`.

Load the estimation data. In this example, use a static data set for illustration.

```
load iddata1 z1;
output = z1.y;
input = z1.u;
```

Estimate ARMAX model parameters online using `step`.

```
for i = 1:numel(input)
[A,B,C,EstimatedOutput] = step(obj,output(i),input(i));
end
```

View the current estimated values of polynomial `A` coefficients.

```
obj.A
```

```
ans =
```

```
    1.0000   -0.8298
```

View the current covariance estimate of the parameters.

```
obj.ParameterCovariance
```

```
ans =

    0.0001    0.0001    0.0001
    0.0001    0.0032    0.0000
    0.0001    0.0000    0.0001
```

View the current estimated output.

```
EstimatedOutput
```

```
EstimatedOutput =

   -4.5595
```

Tune Recursive Estimation Algorithm Properties During Online Parameter Estimation

Create a System object for online parameter estimation of an ARMAX model.

```
obj = recursiveARMAX;
```

The ARMAX model has a default structure with polynomials of order 1 and initial polynomial coefficient values, `eps`.

Load the estimation data. In this example, use a static data set for illustration.

```
load iddata1 z1;
output = z1.y;
input = z1.u;
dataSize = numel(input);
```

Estimate ARMAX model parameters online using the default recursive estimation algorithm, Forgetting Factor. Change the `ForgettingFactor` property during online parameter estimation.

```
for i = 1:dataSize
    if i == dataSize/2
```

```
        obj.ForgettingFactor = 0.98;
    end
[A,B,C,EstimatedOutput] = step(obj,output(i),input(i));
end
```

Estimate Parameters of System Using Recursive Least Squares Algorithm

The system has two parameters and is represented as:

$$y(t) = a_1 u(t) + a_2 u(t-1).$$

Here,

- u and y are the real-time input and output data, respectively.
- $u(t)$ and $u(t-1)$ are the regressors, `H`, of the system.
- a_1 and a_2 are the parameters, `theta`, of the system.

Create a System object for online estimation using recursive least squares algorithm.

```
obj = recursiveLS(2);
```

Load the estimation data. In this example, we are using a static data set for illustration.

```
load iddata3
input = z3.u;
output = z3.y;
```

Create a variable to store `u(t-1)`. This variable is updated at each time step.

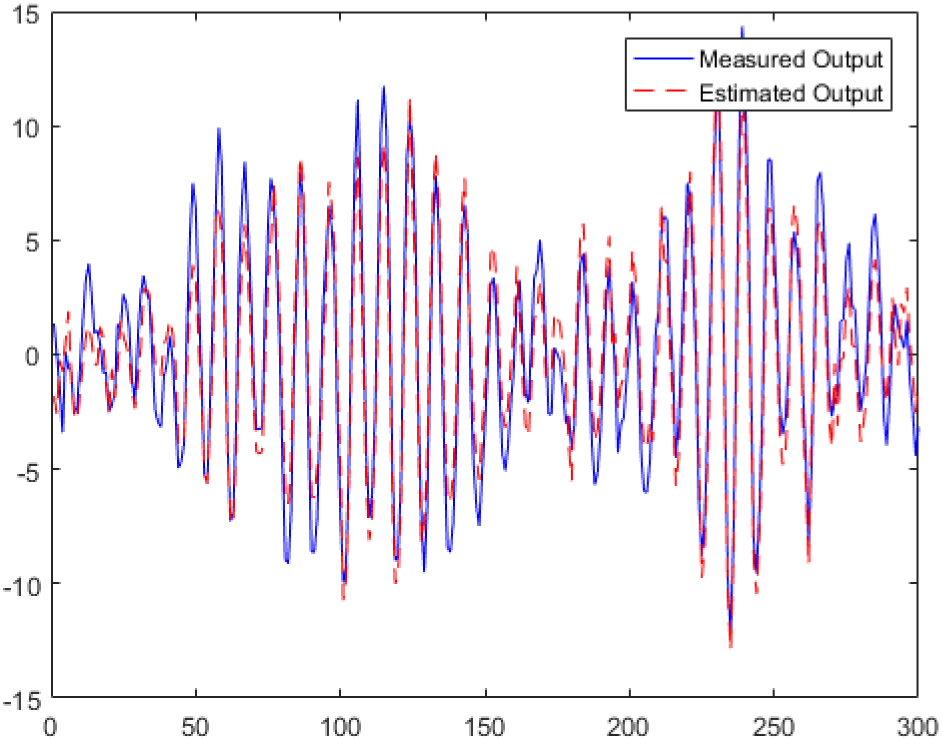
```
oldInput = 0;
```

Estimate the parameters and output using `step` and input-output data.

```
for i = 1:numel(input)
    H = [input(i) oldInput];
    [theta, EstimatedOutput] = step(obj,output(i),H);
    estimatedOut(i)= EstimatedOutput;
    oldInput = input(i);
end
```

Plot the measured and estimated output data.

```
numSample = 1:numel(input);
plot(numSample,output,'b',numSample,estimatedOut,'r--');
legend('Measured Output','Estimated Output');
```



- “Perform Online Parameter Estimation at the Command Line”
- “Validate Online Parameter Estimation at the Command Line”
- “Online ARX Parameter Estimation for Tracking Time-Varying System Dynamics”
- “Line Fitting with Online Recursive Least Squares Estimation”

Input Arguments

`obj` — System object for online parameter estimation

`recursiveAR` object | `recursiveARMA` object | `recursiveARX` object | `recursiveARMAX` object | `recursiveOE` object | `recursiveBJ` object | `recursiveLS` object

System object for online parameter estimation, created using one of the following commands:

- `recursiveAR`
- `recursiveARMA`
- `recursiveARX`
- `recursiveARMAX`
- `recursiveOE`
- `recursiveBJ`
- `recursiveLS`

The `step` command updates parameters of the model using the recursive estimation algorithm specified in `obj` and the incoming input-output data.

y — Output data

real scalar

Output data acquired in real time, specified as a real scalar.

`InputData` — Input data

scalar | vector of real values

Input data acquired in real time, specified as a scalar or vector of real values depending on the type of System object.

System object	Model Type	`InputData`
`recursiveAR`	Time-series	Not Applicable
`recursiveARMA`	Time-series	Not Applicable
`recursiveARX`	SISO ARX	Real scalar
	MISO ARX with *N* inputs	Column vector of length *N*, specified as real values
`recursiveARMAX`	SISO	Real scalar
`recursiveOE`	SISO	Real scalar
`recursiveBJ`	SISO	Real scalar
`recursiveLS`	Single output system with *Np* system parameters	Regressors, *H*, specified as a vector of real values of length *Np*

Output Arguments

`EstimatedParameters` — Estimated model parameters

vector of real values for each parameter

Estimated model parameters, returned as vectors of real values. The number of estimated parameters, and so the `step` syntax, depend on the type of System object:

Online Estimation System Object	Estimated Parameters
`recursiveAR`	Polynomial $A(q)$ coefficients
`recursiveARMA`	Polynomials $A(q)$ and $C(q)$ coefficients
`recursiveARX`	Polynomials $A(q)$ and $B(q)$ coefficients
`recursiveARMAX`	Polynomials $A(q)$, $B(q)$, and $C(q)$ coefficients
`recursiveOE`	Polynomials $B(q)$ and $F(q)$
`recursiveBJ`	Polynomials $B(q)$, $C(q)$, $D(q)$, and $F(q)$ coefficients
`recursiveLS`	System parameters, θ

`EstimatedOutput` — Estimated output

real scalar

Estimated output, returned as a real scalar. The output is estimated using input-output estimation data, current parameter values, and recursive estimation algorithm specified in `obj`.

More About

Tips

- Starting in R2016b, instead of using the `step` command to update model parameter estimates, you can call the System object with input arguments, as if it were a function. For example, `[A,EstimatedOutput] = step(obj,y)` and `[A,EstimatedOutput] = obj(y)` perform equivalent operations.
- “What Is Online Estimation?”

See Also

`clone` | `isLocked` | `recursiveAR` | `recursiveARMA` | `recursiveARMAX` | `recursiveARX` | `recursiveBJ` | `recursiveLS` | `recursiveOE` | `release` | `reset`

Introduced in R2015b

stepDataOptions

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.

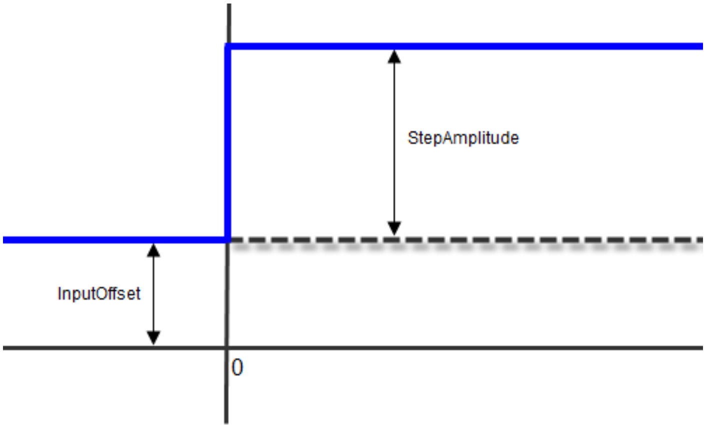
Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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

`'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`.

Examples

Specify Input Offset and Step Amplitude Level for Step Response

Create a transfer function model.

```
sys = tf(1,[1,1]);
```

Create an option set for `step` to specify input offset and step amplitude level.

```
opt = stepDataOptions('InputOffset',-1,'StepAmplitude',2);
```

Calculate the step response using the specified options.

```
[y,t] = step(sys,opt);
```

See Also

`step`

Introduced in R2012a

stepinfo

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) - yfinal|` 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`.

This command requires Control System Toolbox license.

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`

Introduced in R2006a

stepplot

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)
stepplot(..., dataoptions)
```

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 = Tfinal`. Express `Tfinal` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time systems with unspecified sample time (`Ts = -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`.

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)` customizes the plot appearance using the options set, `plotoptions`. Use `timeOptions` to create the options set.

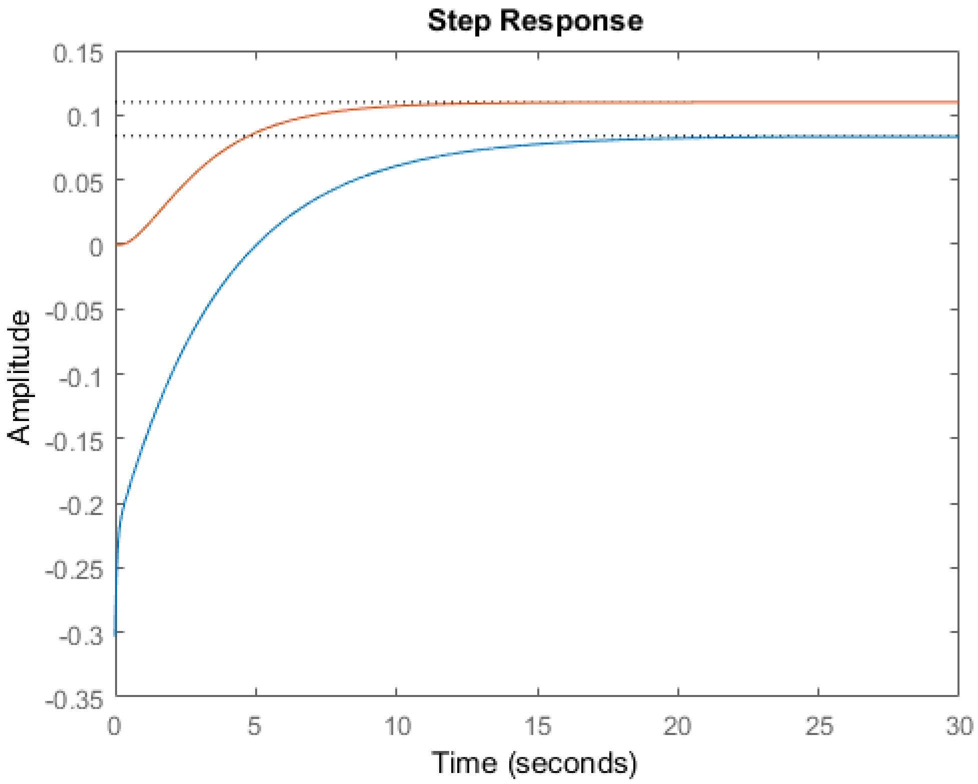
`stepplot(..., dataoptions)` specifies options such as the step amplitude and input offset using the options set, `dataoptions`. Use `stepDataOptions` to create the options set.

Examples

Normalized Response on Step Plot

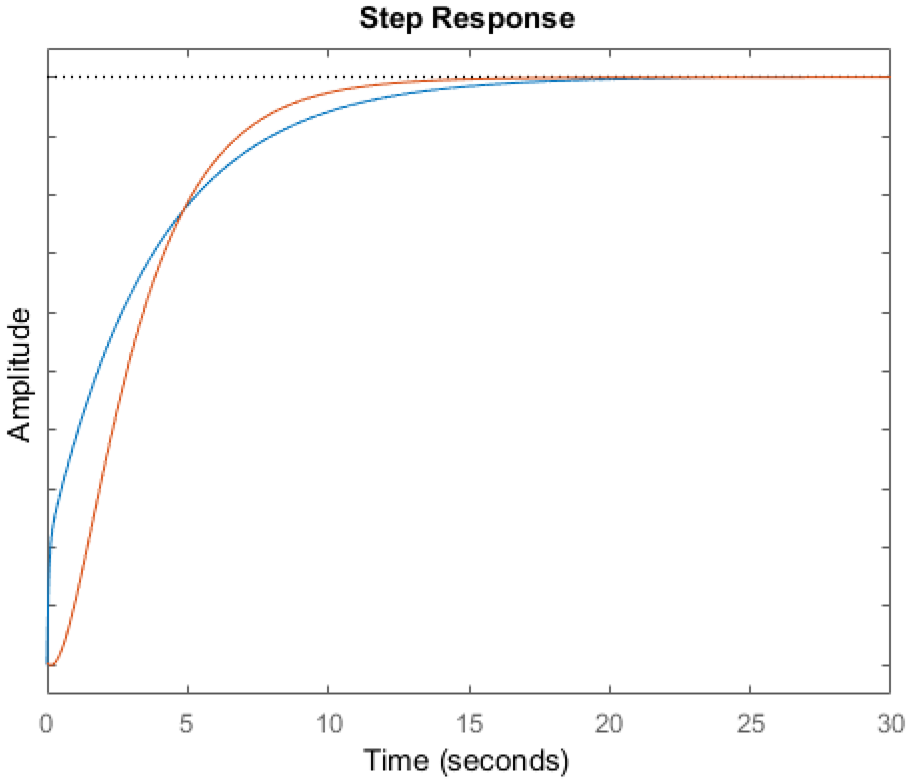
Generate a step response plot for two dynamic systems.

```
sys1 = rss(3);
sys2 = rss(3);
h = stepplot(sys1,sys2);
```



Each step response settles at a different steady-state value. Use the plot handle to normalize the plotted response.

```
setoptions(h,'Normalize','on')
```



Now, the responses settle at the same value expressed in arbitrary units.

Step Responses of Identified Models with Confidence Region

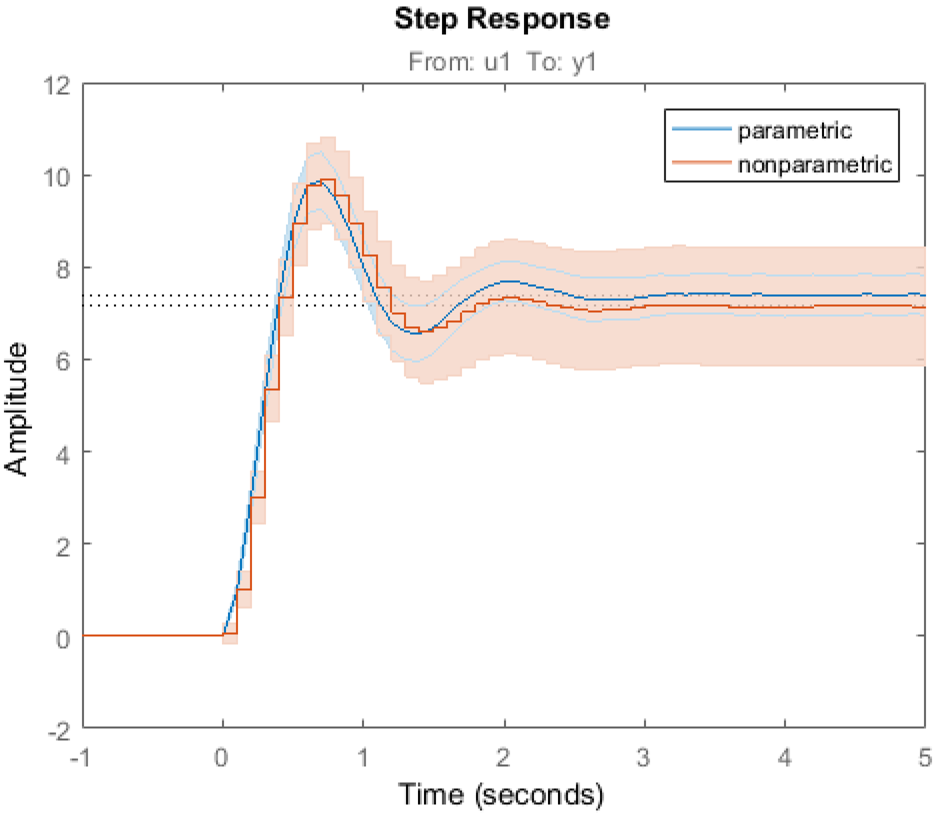
Compare the step response of a parametric identified model to a nonparametric (empirical) model, and view their 3-σ confidence regions. (Identified models require System Identification Toolbox™ software.)

Identify a parametric and a nonparametric model from sample data.

```
load iddata1 z1
sys1 = ssest(z1,4);
sys2 = impulseest(z1);
```

Plot the step responses of both identified models. Use the plot handle to display the 3-σ confidence regions.

```
t = -1:0.1:5;
h = stepplot(sys1,sys2,t);
showConfidence(h,3)
legend('parametric','nonparametric')
```



The nonparametric model `sys2` shows higher uncertainty.

Step Response of Nonlinear Model

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);

dataoptions = stepDataOptions('InputOffset', 2, 'StepAmplitude', 0.5);
stepplot(sys,60,dataoptions);
```

More About

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”.

See Also

`showConfidence` | `step` | `getoptions` | `setoptions`

Introduced in R2012a

strseq

Create sequence of indexed character vectors

Syntax

```
txtarray = strseq(TXT,INDICES)
```

Description

`txtarray = strseq(TXT,INDICES)` creates a sequence of indexed character vectors in the cell array `txtarray` by appending the integer values `INDICES` to the character vector `TXT`.

Note: You can use `strvec` to aid in system interconnection. For an example, see the `sumblk` reference page.

Examples

Create a Cell Array of Indexed Text

Index the text 'e' with the numbers 1, 2, and 4.

```
txtarray = strseq('e',[1 2 4])
```

```
txtarray =

  3×1 cell array

    'e1'
    'e2'
    'e4'
```

See Also

`strcat` | `connect`

Introduced in R2012a

struc

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, single-output ARX model estimation. *na* and *nb* are row vectors that specify ranges of model orders. *nk* is a row vector that specifies a 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 an ARX model with *nu* input channels.

Examples

Generate Model-Order Combinations and Estimate ARX Model Using IV Method

Create estimation and validation data sets

```
load iddata1;
ze = z1(1:150);
zv = z1(151:300);
```

Generate model-order combinations for estimation, specifying ranges for model orders and delays.

```
NN = struc(1:3,1:2,2:4);
```

Estimate ARX models using the instrumental variable method, and compute the loss function for each model order combination.

```
V = ivstruc(ze,zv,NN);
```

Select the model order with the best fit to the validation data.

```
order = selstruc(V,0);
```

Estimate an ARX model of selected order.

```
M = iv4(ze,order);
```

Generate Model-Order Combinations and Estimate Multi-Input ARX Model

Create estimation and validation data sets.

```
load co2data;
Ts = 0.5; % Sample time 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.

```
NN = struc(2:4,2:5,[1 4],1:4,0);
```

Estimate an ARX model for each model order combination.

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

Select the model order with the best fit to the validation data.

```
order = selstruc(V,0);
```

Estimate an ARX model of selected order.

```
M = arx(ze,order);
```

- “Estimating Model Orders Using an ARX Model Structure”

More About

Tips

- Use with `arxstruc` or `ivstruc` to compute loss functions for ARX models, one for each model order combination returned by `struc`.

- “Preliminary Step – Estimating Model Orders and Input Delays”

See Also

`arxstruc` | `ivstruc` | `selstruc`

Introduced before R2006a

System Identification

Identify models of dynamic systems from measured data

Description

The **System Identification** app enables you to identify models of dynamic systems from measured input-output data. You can estimate both linear and nonlinear models and compare responses of different models.

Using this app you can:

- Import, plot, and preprocess measured input-output data.
- Estimate linear models such as transfer functions, process models, polynomial models, and state-space models using time-domain, time series, or frequency-domain data.
- Estimate nonlinear ARX and Hammerstein-Wiener models using time-domain data.
- Validate estimated models using independent data sets.
- Export estimated models for further analysis to MATLAB workspace or to the **Linear System Analyzer** app in Control System Toolbox.

For more information, in the **System Identification** app, select **Help** > **System Identification App Help**.

Open the System Identification App

- MATLAB Toolstrip: On the **Apps** tab, under **Control System Design and Analysis**, click the app icon.
- MATLAB command prompt: Enter `systemIdentification`.

Examples

- "Working with System Identification App"
- "Identify Linear Models Using System Identification App"
- "Identify Nonlinear Black-Box Models Using System Identification App"

Programmatic Use

`systemIdentification` opens the **System Identification** app. If the app is already open, the command brings the app into focus.

`systemIdentification(sessionFile)` opens the app and loads a previously saved session file, `sessionFile`, on the MATLAB path. A session includes data sets and models in the app at the time of saving. If the app is already open, the command merges the contents of the session file with those already present in the app.

For example, `systemIdentification('mySession')` opens the app and loads the previously saved app session `mySession.sid`.

To save a session, in the **System Identification** app, select **File** > **Save session**. The session is saved to a file with a `.sid` extension.

`systemIdentification(sessionFile,path)` specifies the path to the session file. Use this syntax if the file is not on the MATLAB path.

For example, `systemIdentification('mySession','C:\matlab\work')` opens the app and loads the previously saved app session `mySession.sid` located at `C:\matlab\work`.

See Also

Functions

`arx` | `iddata` | `midprefs` | `nlarx` | `nlhw` | `polyest` | `procest` | `ssest` | `tfest`

Introduced before R2006a

tfdata

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 `num{i,j}(k)` and `sdden{i,j}(k)` is the 1 standard uncertainty in the value `den{i,j}(k)`. If `sys` does not contain uncertainty information, `sdnum` 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`

Introduced before R2006a

tfest

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. All input arguments described for previous syntaxes also apply here.

`sys = tfest(data,init_sys)` uses the linear system `init_sys` to configure the initial parameterization of `sys`.

`sys = tfest(___,opt)` specifies the estimation behavior using the option set `opt`. All input arguments described for previous syntaxes also apply here.

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:

- `frd` or `idfrd` object that represents recorded frequency response data:
  - Complex-values $G(e^{i\omega})$, for given frequencies ω
  - Amplitude $|G|$ and phase shift $\varphi = \arg G$ values
- `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 intersample 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 *ny*-by-*nu* matrix. *ny* is the number of outputs, and *nu* 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 *ny*-by-*nu* matrix. *ny* is the number of outputs, and *nu* is the number of inputs.

For a continuous-time model, estimated using discrete-time data, set `nz <= 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 sample time `Ts`.

For a MIMO system with *ny* outputs and *nu* inputs, set `iodelay` to an *ny*-by-*nu* array. Each entry of this array is a numerical value that represents 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, created using `tfestOptions`, that specifies estimation options including:

- Estimation objective
- Handling of initial conditions
- Numerical search method to be used in estimation

`init_sys`

Linear system that configures the initial parameterization of `sys`.

You obtain `init_sys` by either performing an estimation using measured data or by direct construction.

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. For example:

- To specify an initial guess for the numerator of `init_sys`, set `init_sys.Structure.Numerator.Value` to the initial guess.
- To specify constraints for the numerator of `init_sys`:
  - Set `init_sys.Structure.Numerator.Minimum` to the minimum numerator coefficient values
  - Set `init_sys.Structure.Numerator.Maximum` to the maximum numerator coefficient values
  - Set `init_sys.Structure.Numerator.Free` to indicate which numerator coefficients are free for estimation

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.

If `opt` is not specified, and `init_sys` was obtained 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. `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'`

Sample 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 delay for each input channel, specified as a scalar value or numeric vector. 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 sample time `Ts`. For example, `InputDelay = 3` means a delay of three sample times.

For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector. Each entry of this vector is a numerical value that represents 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

`'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, returned as an `idtf` model. This model is created using the specified model orders, delays and estimation options.

Information about the estimation results and options used is stored in the `Report` property of the model. `Report` has the following fields:

Report Field	Description
`Status`	Summary of the model status, which indicates whether the model was created by construction or obtained by estimation.
`Method`	Estimation command used.
`InitMeth`	Algorithm used to initialize the numerator and denominator for estimation of continuous-time transfer functions using time-domain data, returned as one of the following values: • `'iv'` — Instrument Variable approach. • `'svf'` — State Variable Filters approach. • `'gpmf'` — Generalized Poisson Moment Functions approach. • `'n4sid'` — Subspace state-space estimation approach. This field is especially useful to view the algorithm used when the `InitMethod` option in the estimation option set is `'all'`.
`N4Weight`	Weighting matrices used in the singular-value decomposition step when `InitMethod` is `'n4sid'`, returned as one of the following values: • `'MOESP'` — Uses the MOESP algorithm by Verhaegen. • `'CVA'` — Uses the canonical variable algorithm (CVA) by Larimore. • `'SSARX'` — A subspace identification method that uses an ARX estimation based algorithm to compute the weighting. This field is especially useful to view the weighting matrices used when the `N4Weight` option in the estimation option set is `'auto'`.
`N4Horizor`	Forward and backward prediction horizons used when `InitMethod` is `'n4sid'`, returned as a row vector with three elements — `[r sy su]`, where `r` is the maximum forward prediction horizon. `sy` is the number of past outputs, and `su` is the number of past inputs that are used for the predictions.
`InitialCo`	Handling of initial conditions during model estimation, returned as one of the following values:

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td></td><td>

- `'zero'` — The initial conditions were set to zero.
- `'estimate'` — The initial conditions were treated as independent estimation parameters.
- `'backcast'` — The initial conditions were estimated using the best least squares fit.

This field is especially useful to view how the initial conditions were handled when the `InitialCondition` option in the estimation option set is `'auto'`.
</td></tr>
<tr><td><code>Fit</code></td><td>
Quantitative assessment of the estimation, returned as a structure. See “Loss Function and Model Quality Metrics” for more information on these quality metrics. The structure has the following fields:

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>FitPerc</code></td><td>Normalized root mean squared error (NRMSE) measure of how well the response of the model fits the estimation data, expressed as a percentage.</td></tr>
<tr><td><code>LossFcn</code></td><td>Value of the loss function when the estimation completes.</td></tr>
<tr><td><code>MSE</code></td><td>Mean squared error (MSE) measure of how well the response of the model fits the estimation data.</td></tr>
<tr><td><code>FPE</code></td><td>Final prediction error for the model.</td></tr>
<tr><td><code>AIC</code></td><td>Raw Akaike Information Criteria (AIC) measure of model quality.</td></tr>
<tr><td><code>AICc</code></td><td>Small sample-size corrected AIC.</td></tr>
<tr><td><code>nAIC</code></td><td>Normalized AIC.</td></tr>
<tr><td><code>BIC</code></td><td>Bayesian Information Criteria (BIC).</td></tr>
</table>
</td></tr>
<tr><td><code>Parameter</code></td><td>Estimated values of model parameters.</td></tr>
<tr><td><code>OptionsUs</code></td><td>Option set used for estimation. If no custom options were configured, this is a set of default options. See <code>polyestOptions</code> for more information.</td></tr>
<tr><td><code>RandState</code></td><td>State of the random number stream at the start of estimation. Empty, <code>[]</code>, if randomization was not used during estimation. For more information, see <code>rng</code> in the MATLAB documentation.</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>DataUsed</code></td><td>Attributes of the data used for estimation, returned as a structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>Name</code></td><td>Name of the data set.</td></tr>
<tr><td><code>Type</code></td><td>Data type.</td></tr>
<tr><td><code>Length</code></td><td>Number of data samples.</td></tr>
<tr><td><code>Ts</code></td><td>Sample time.</td></tr>
<tr><td><code>InterSam</code></td><td>Input intersample behavior, returned as one of the following values:
• <code>'zoh'</code> — Zero-order hold maintains a piecewise-constant input signal between samples.
• <code>'foh'</code> — First-order hold maintains a piecewise-linear input signal between samples.
• <code>'bl'</code> — Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency.</td></tr>
<tr><td><code>InputOff</code></td><td>Offset removed from time-domain input data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
<tr><td><code>OutputOf</code></td><td>Offset removed from time-domain output data during estimation. For nonlinear models, it is <code>[]</code>.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Report Field</th><th>Description</th></tr>
<tr><td><code>Terminati</code></td><td>Termination conditions for the iterative search used for prediction error minimization. Structure with the following fields:
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>WhyStop</code></td><td>Reason for terminating the numerical search.</td></tr>
<tr><td><code>Iteratio</code></td><td>Number of search iterations performed by the estimation algorithm.</td></tr>
<tr><td><code>FirstOrd</code></td><td>∞ -norm of the gradient search vector when the search algorithm terminates.</td></tr>
<tr><td><code>FcnCount</code></td><td>Number of times the objective function was called.</td></tr>
<tr><td><code>UpdateNo</code></td><td>Norm of the gradient search vector in the last iteration. Omitted when the search method is <code>'lsqnonlin'</code>.</td></tr>
<tr><td><code>LastImpr</code></td><td>Criterion improvement in the last iteration, expressed as a percentage. Omitted when the search method is <code>'lsqnonlin'</code>.</td></tr>
<tr><td><code>Algorith</code></td><td>Algorithm used by <code>'lsqnonlin'</code> search method. Omitted when other search methods are used.</td></tr>
</table>
For estimation methods that do not require numerical search optimization, the <code>Termination</code> field is omitted.</td></tr>
</table>

For more information on using `Report`, see “Estimation Report”.

Examples

Estimate Transfer Function Model By Specifying Number of Poles

Load time-domain system response data and use it to estimate a transfer function for the system.

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

`z1` is an `iddata` object that contains time-domain, input-output data.

`np` specifies the number of poles in the estimated transfer function.

`sys` is an `idtf` model containing the estimated transfer function.

To see the numerator and denominator coefficients of the resulting estimated model `sys`, enter:

```
sys.Numerator
sys.Denominator
```

```
ans =

    2.4554  176.9856


ans =

    1.0000    3.1625   23.1631
```

To view the uncertainty in the estimates of the numerator and denominator and other information, use `tfdata`.

Specify Number of Poles and Zeros in Estimated Transfer Function

Load time-domain system response data and use it to estimate 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 estimated transfer function, respectively.

`sys` is an `idtf` model containing the estimated transfer function.

Estimate Transfer Function Containing Known Transport Delay

Load time-domain system response data and use it to estimate 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 estimated transfer function, respectively.

`iodelay` specifies the transport delay for the estimated transfer function as 0.2 seconds.

`sys` is an `idtf` model containing the estimated transfer function, with `IODelay` property set to 0.2 seconds.

Estimate Transfer Function Containing Unknown Transport Delay

Load time-domain system response data and use it to estimate 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 estimated transfer function, respectively.

`iodelay` specifies the transport delay for the estimated transfer function. `iodelay = NaN` denotes the transport delay as an unknown parameter to be estimated.

`sys` is an `idtf` model containing the estimated transfer function, whose `IODelay` property is estimated using data.

Estimate Discrete-Time Transfer Function With No Feedthrough

Load time-domain system response data.

```
load iddata2 z2;
```

`z2` is an `iddata` object that contains time-domain system response data.

Estimate 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

Compare two discrete-time models with and without feedthrough and transport delay.

If there is a delay from the measured input to output, it can be attributed to a lack of feedthrough or to a true transport delay. For discrete-time models, absence of feedthrough corresponds to a lag of 1 sample between the input and output. Estimating a model with `Feedthrough = false` and `IODelay = 0` thus produces a discrete-time system that is equivalent to a system with `Feedthrough = true` and `IODelay = 1`. Both systems show the same time- and frequency-domain responses, for example, on step and Bode plots. However, you get different results if you reduce these models using `balred` or convert them to their continuous-time representation. Therefore, you should check if the observed delay should be attributed to transport delay or to a lack of feedthrough.

Estimate a discrete-time model with no feedthrough.

```
load iddata1 z1
np = 2;
nz = 2;
model1 = tfest(z1,np,nz,'Ts',z1.Ts);
```

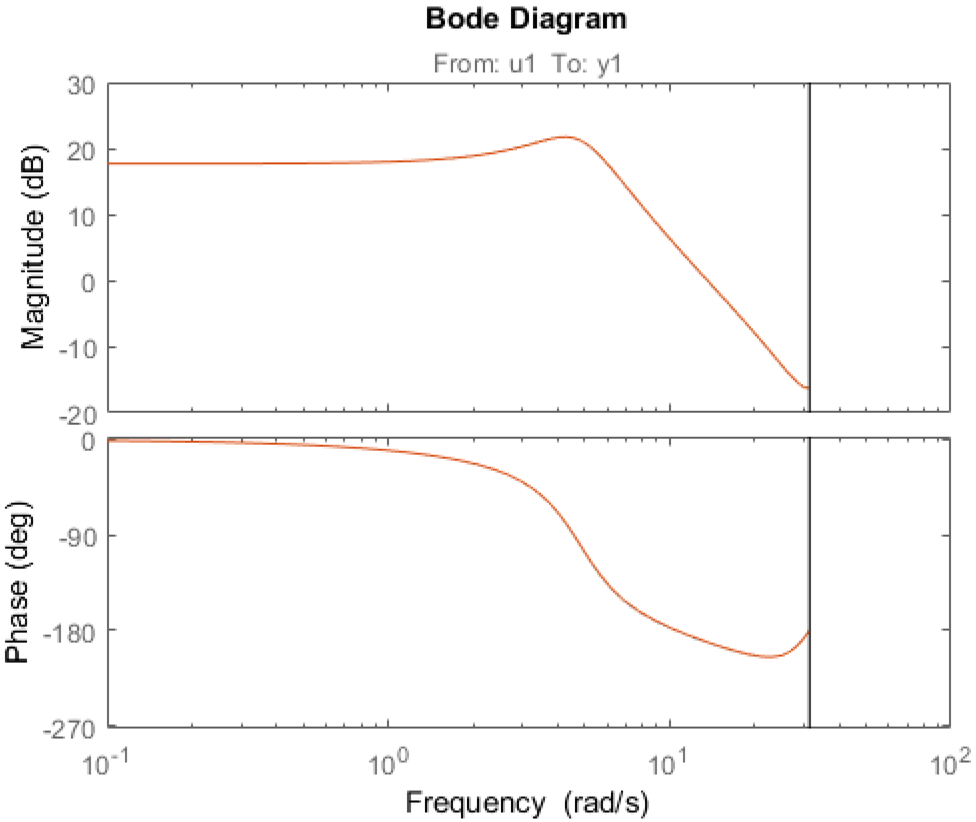
`model1` has a transport delay of 1 sample and its `IODelay` property is 0. Its numerator polynomial begins with z^{-1}.

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.

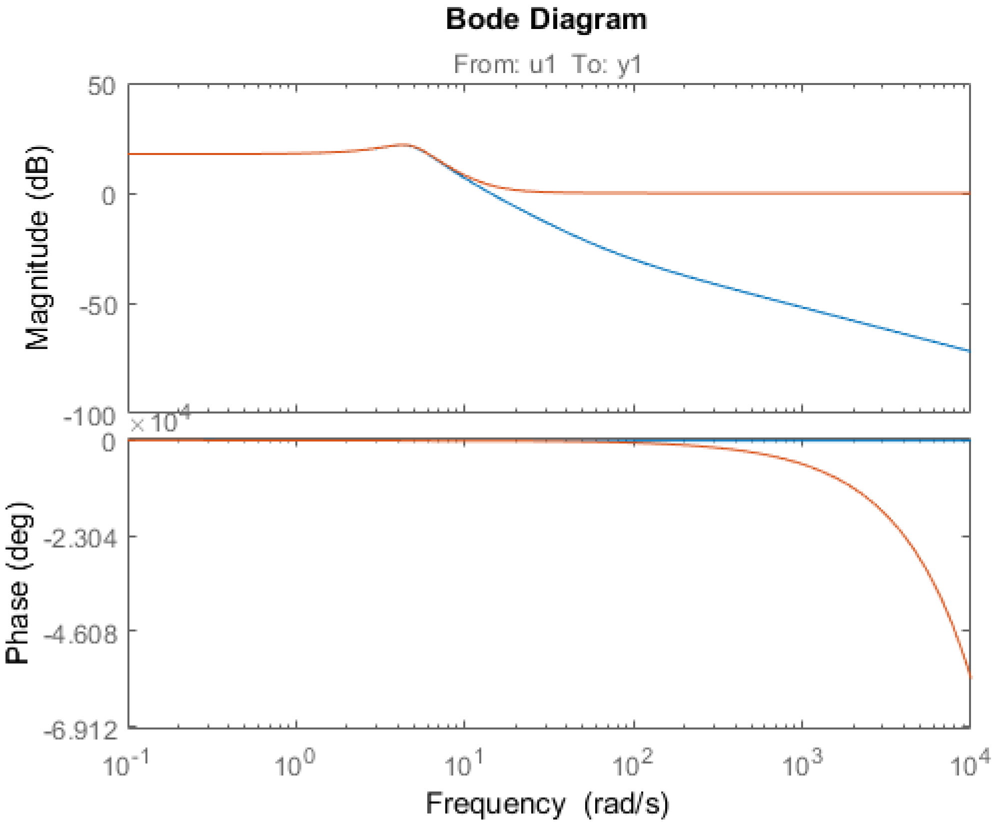
```
bode(model1,model2);
```



The equations for `model1` and `model2` are equivalent, but the transport delay of `model2` has been absorbed into the numerator of `model1`.

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 and zero seconds on the second input. Both inputs have no feedthrough.

Split data into estimation and validation data sets.

```
load iddata7 z7
ze = z7(1:300);
zv = z7(200:400);
```

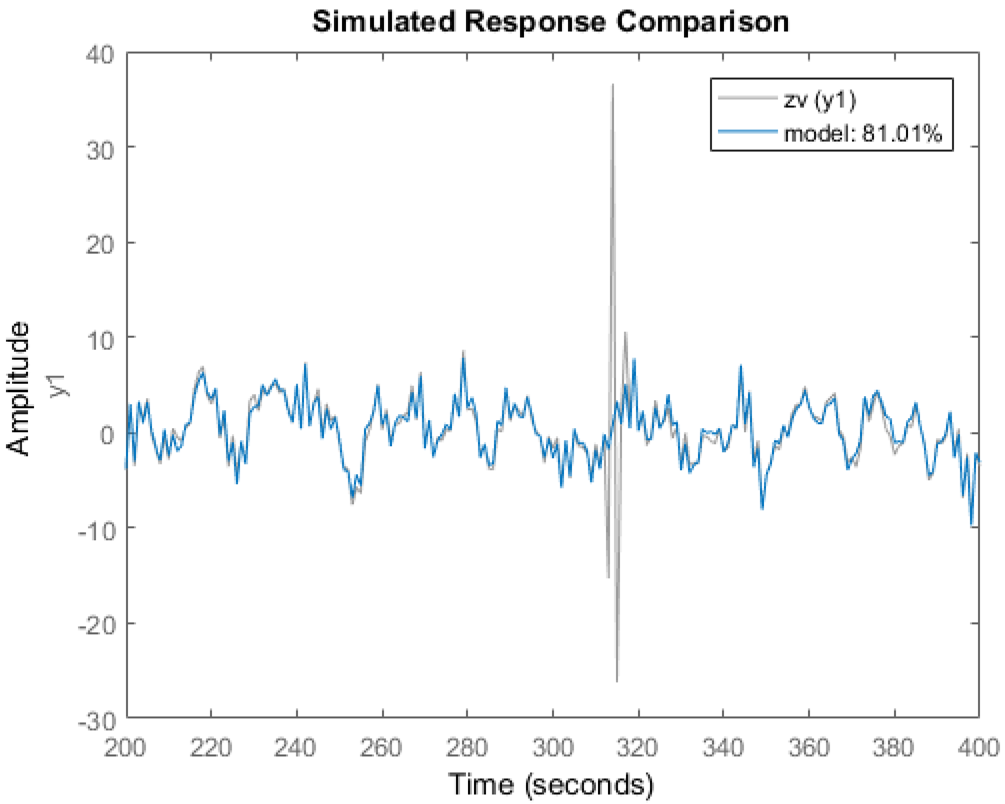
Estimate a 2-input, 1-output transfer function with 2 poles and 1 zero for each input-to-output transfer function.

```
Lag = [2;0];
Ft = [false,false];
model = tfest(ze,2,1,'Ts',z7.Ts,'Feedthrough',Ft,'InputDelay',Lag);
```

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. Exclude the data outliers for validation.

```
I = 1:201;
I(114:118) = [];
opt = compareOptions('Samples',I);
compare(zv,model,opt)
```



Estimate Transfer Function Model Using Regularized Impulse Response Model

Identify a 15th order transfer function model by using regularized impulse response estimation

Load data.

```
load regularizationExampleData m0simdata;
```

Obtain regularized impulse response (FIR) model.

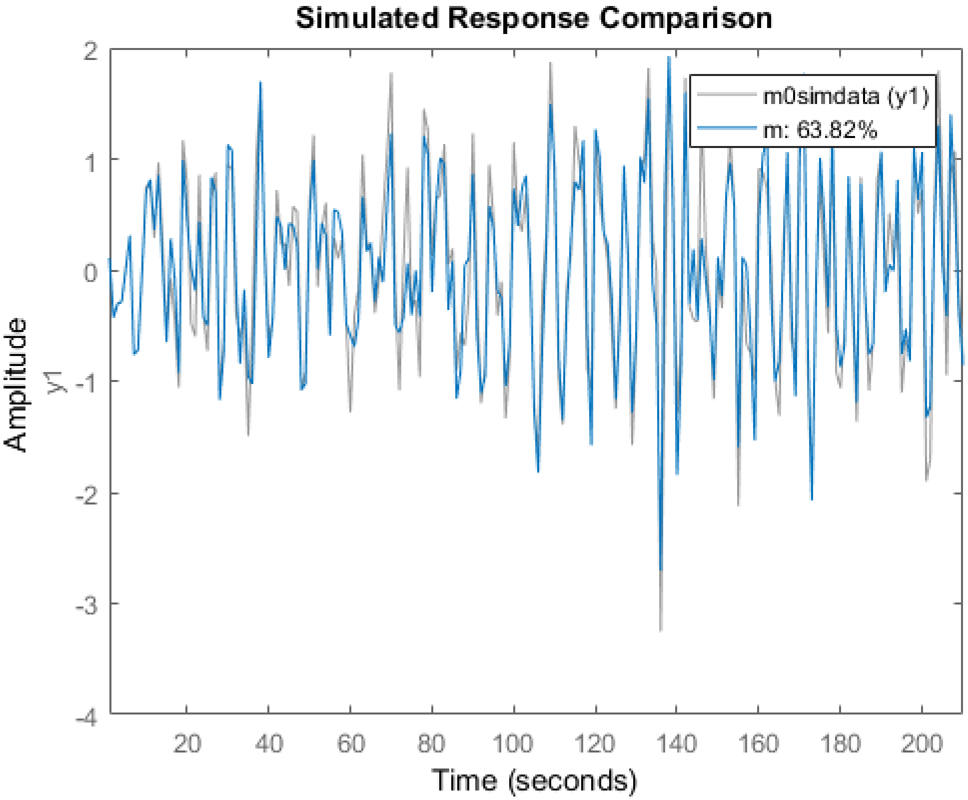
```
opt = impulseestOptions('RegulKernel','DC');
m0 = impulseest(m0simdata,70,opt);
```

Convert model into a transfer function model after reducing order to 15.

```
m = idtf(balred(idss(m0),15));
```

Compare the model output with data.

```
compare(m0simdata,m);
```



Estimate Transfer Function Using an Estimation Option Set

Create the option set for `tfest`.

```
opt = tfestOptions('InitMethod','n4sid','Display','on','SearchMethod','lsqnonlin');
```

`opt` specifies that the initialization method as `'n4sid'`, and the search method as `'lsqnonlin'`. It also specifies that the loss-function values for each iteration be shown.

Load time-domain system response data and use it to estimate 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 estimated transfer function, respectively.

`iodelay` specifies the transport delay for the estimated transfer function as 0.2 seconds.

`opt` specifies the estimation options.

`sys` is an `idtf` model containing the estimated transfer function.

Specify Model Properties of the Estimated Transfer Function

Load time-domain system response data, and use it to estimate a transfer function for the system. Specify the input delay for the estimated 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 estimated transfer function, respectively.

`input_delay` specifies the input delay for the estimated transfer function as 0.2 seconds.

`sys` is an `idtf` model containing the estimated transfer function with an input delay of 0.2 seconds.

Convert Frequency-Response Data into Transfer Function

This example requires a Control System Toolbox™ license.

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 0.2],[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 estimated transfer function, respectively.

`sys` is an `idtf` model containing the estimated transfer function.

Estimate Transfer Function with Known Transport Delays for Multiple Inputs

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`. Also specify the maximum search iterations and input/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.

Estimate 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 estimated transfer function.

Estimate Transfer Function with Known and Unknown Transport Delays

Load time-domain system response data and use it to estimate 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`. Also specify the maximum search iterations and input/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/output offsets.

Estimate the transfer function. 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 estimated transfer function.

Estimate Transfer Function with Unknown, Constrained Transport Delays

Create a transfer function model with the expected numerator and denominator structure and delay constraints.

In 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.

Specify the transfer function from both inputs to the output.

```
init_sys = [init_sys,init_sys];
```

Load time-domain system response data and use it to estimate a transfer function.

```
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);
```

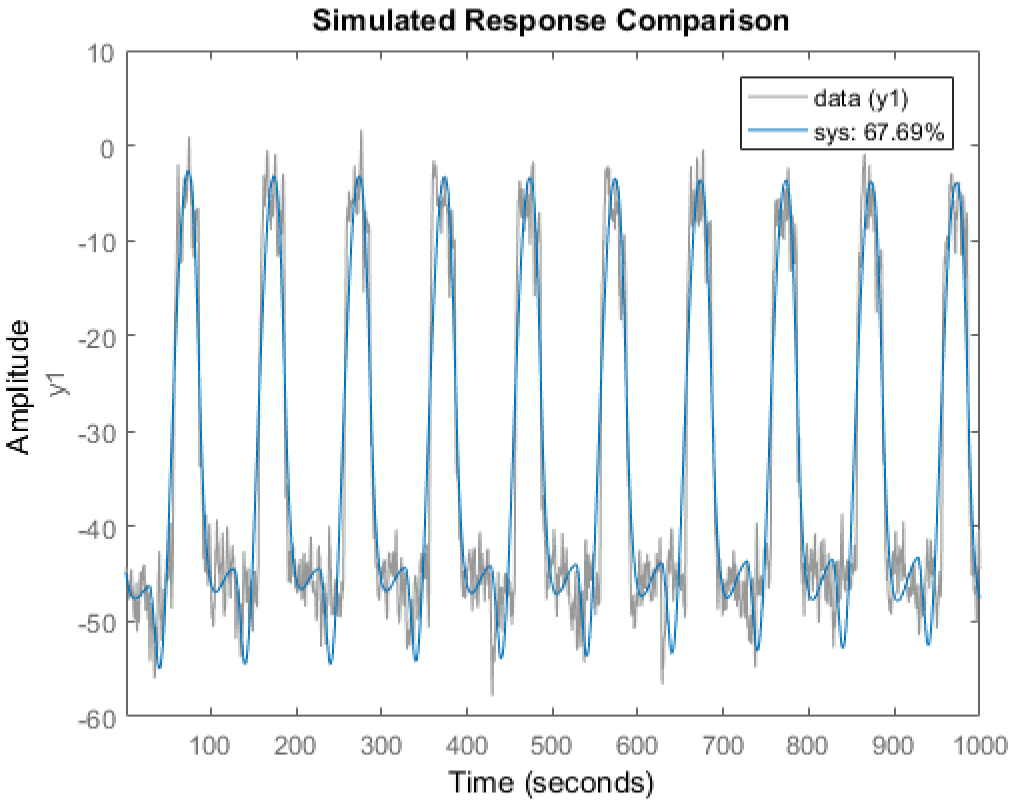
`data` is an `iddata` object and has a sample rate of 0.5 seconds.

`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 estimated transfer function.

Analyze the estimation result by comparison.

```
opt2 = compareOptions;
opt2.InputOffset = opt.InputOffset;
opt2.OutputOffset = opt.OutputOffset;
compare(data,sys,opt2)
```



Estimate Transfer Function Containing Different Number of Poles for Input/Output Pairs

Estimate a multiple-input, single-output transfer function containing different number of poles for input/output pairs for given data.

This example requires a Control System Toolbox™ license.

Obtain frequency-response data.

For example, use `frd` to create a frequency-response data model for the following system:

$$G = \left[\begin{array}{c} e^{-4s}\frac{s+2}{s^3+2s^2+4s+5} \\ e^{-0.6s}\frac{5}{s^4+2s^3+s^2+s} \end{array} \right]$$

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 0.6]);
data = frd(G,logspace(-2,2,100));
```

`data` is an `frd` object containing the continuous-time frequency response for `G`.

Estimate a transfer function for `data`.

```
 np = [3 4];
 nz = [1 0];
 iodelay = [4 0.6];
 sys = tfest(data,np,nz,iodelay);
```

`np` specifies the number of poles in the estimated 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 estimated 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.

`iodelay` 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 estimated transfer function.

Estimate Transfer Function for Unstable System

Estimate a transfer function describing an unstable system for given data.

Obtain frequency-response data.

For example, use `frd` to create 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 = idtf({[1 2], 5},{[1 2 4 5],[1 2 1 1 1]});
data = idfrd(G,logspace(-2,2,100));
```

`data` is an `idfrd` object containing the continuous-time frequency response for `G`.

Estimate a transfer function for `data`.

```
np = [3 4];
nz = [1 0];
sys = tfest(data,np,nz);
```

`np` specifies the number of poles in the estimated 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 estimated 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 `nz` indicates that the transfer function from the second input to the output does not contain any zeros.

`sys` is an `idtf` model containing the estimated transfer function.

```
pole(sys)
```

```
ans =

  -1.5260 + 0.0000i
  -0.2370 + 1.7946i
  -0.2370 - 1.7946i
```

```
  -1.4656 + 0.0000i
  -1.0000 + 0.0000i
   0.2328 + 0.7926i
   0.2328 - 0.7926i
```

`sys` is an unstable system as verified by the pole display.

- "Estimate Transfer Function Models at the Command Line"
- "Estimate Transfer Function Models with Transport Delay to Fit Given Frequency-Response Data"
- "Estimate Transfer Function Models With Prior Knowledge of Model Structure and Constraints"
- "Troubleshoot Frequency-Domain Identification of Transfer Function Models"

More About

Compatibility

Starting in R2016b, a new algorithm is used for performing transfer function estimation from frequency-domain data. You are likely to see faster and more accurate results with the new algorithm, particularly for data with dynamics over a large range of frequencies and amplitudes. However, the estimation results may not match results from previous releases. To perform estimation using the previous estimation algorithm, append `'-R2016a'` to the syntax.

For example, suppose that you are estimating a transfer function model with `np` poles using frequency-domain data, `data`.

```
sys = tfest(data,np)
```

To use the previous estimation algorithm, use the following syntax.

```
sys = tfest(data,np,'-R2016a')
```

Algorithms

The details of the estimation algorithms used by `tfest` 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 prefiltered data [1] [2]. The constant $\frac{1}{\lambda}$ in [1] and [2] corresponds to the initialization option (`InitOption`) field `FilterTimeConstant`. IV is the simplified refined IV method and is called SRIVC in [3]. This method has a prefilter that is the denominator of the current model, initialized with SVF. This prefilter is 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 estimation algorithm, that it transforms 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 Using Time-Domain Data

For 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 Continuous-Time Frequency-Domain Data

The estimation algorithm performs the following tasks:

1. Perform a bilinear mapping to transform the domain (frequency grid) of the transfer function. For continuous-time models, the imaginary axis is transformed to the unit disk. For discrete-time models, the original domain unit disk is transformed to another unit disk.

2. Perform S-K iterations [4] to solve a nonlinear least-squares problem — Consider a multi-input single-output system. The nonlinear least-squares problem is to minimize the loss function:

   $$\underset{D,N_i}{\text{minimize}} \sum_{k=1}^{n_f} \left| W(\omega_k) \left(y(\omega_k) - \sum_{i=1}^{n_u} \frac{N_i(\omega_k)}{D(\omega_k)} u_i(\omega_k) \right) \right|^2$$

   Here *W* is a frequency-dependent weight that you specify. *D* is the denominator of the transfer function model that is to be estimated, and N_i is the numerator corresponding to the *i*th input. *y* and *u* are the measured output and input data, respectively. n_f and n_u are the number of frequencies and inputs, and *w* is the frequency. Rearranging the terms gives:

   $$\underset{D,N_i}{\text{minimize}} \sum_{k=1}^{n_f} \left| \frac{W(\omega_k)}{D(\omega_k)} \left(D(\omega_k) y(\omega_k) - \sum_{i=1}^{n_u} N_i(\omega_k) u_i(\omega_k) \right) \right|^2$$

   To perform the S-K iterations, the algorithm iteratively solves

   $$\underset{D_m,N_{i,m}}{\text{minimize}} \sum_{k=1}^{n_f} \left| \frac{W(\omega_k)}{D_{m-1}(\omega_k)} \left(D_m(\omega_k) y(\omega_k) - \sum_{i=1}^{n_u} N_{i,m}(\omega_k) u_i(\omega_k) \right) \right|^2$$

   where *m* is the current iteration, and $D_{m-1}(\omega)$ is the denominator response identified at the previous iteration. Now each step of the iteration is a linear least-squares problem, where the identified parameters capture the responses $D_m(\omega)$ and $N_{i,m}(\omega)$ for $i = 1,2,...n_u$. The iteration is initialized by choosing $D_0(\omega) = 1$.

   - The first iteration of the algorithm identifies $D_1(\omega)$. The $D_1(\omega)$ and $N_{i,1}(\omega)$ polynomials are expressed in monomial basis.

- The second and following iterations express the polynomials $D_m(\omega)$ and $N_{i,m}(\omega)$ in terms of orthogonal rational basis functions on the unit disk. These basis functions have the form:

$$B_{j,m}(\omega)=\left(\frac{\sqrt{1-\left|\lambda_{j,m-1}\right|^2}}{q-\lambda_{j,m-1}}\right)\prod_{r=0}^{j-1}\frac{1-(\lambda_{j,m-1})^{*}q(\omega)}{q(\omega)-\lambda_{r,m-1}}$$

  Here $\lambda_{j,m-1}$ is the jth pole that is identified at the previous step, m-1, of the iteration. $\lambda_{j,m-1}^{*}$ is the complex conjugate of $\lambda_{j,m-1}$, and q is the frequency-domain variable on the unit disk.

- The algorithm runs for a maximum of 20 iterations. The iterations are terminated early if the relative change in the value of the loss function is less than 0.001 in the last three iterations.

If you specify bounds on transfer function coefficients, these bounds correspond to affine constraints on the identified parameters. If you only have equality constraints (fixed transfer function coefficients), the corresponding equality constrained least-squares problem is solved algebraically. To do so, the software computes an orthogonal basis for the null space of the equality constraint matrix, and then solves least-squares problem within this null space. If you have upper or lower bounds on transfer function coefficients, the corresponding inequality constrained least-squares problem is solved using interior-point methods.

3. Perform linear refinements — The S-K iterations, even when they converge, do not always yield a locally optimal solution. To find a critical point of the optimization problem that may yield a locally optimal solution, a second set of iterations are performed. The critical points are solutions to a set of nonlinear equations. The algorithm searches for a critical point by successively constructing a linear approximation to the nonlinear equations and solving the resulting linear equations in the least-squares sense. The equations are:

   - Equation for the jth denominator parameter:

$$0 = 2\sum_{k=1}^{n_f} \mathrm{Re}\left\{ \frac{\left|W(\omega_k)\right|^2 B_j^*(\omega_k) \sum_{i=1}^{n_u} N_{i,m-1}^*(\omega_k) u_i^*(\omega_k)}{D_{m-1}^*(\omega_k)\left|D_{m-1}(\omega_k)\right|^2} \left(D_m(\omega_k) y(\omega_k) - \sum_{i=1}^{n_u} N_{i,m}(\omega_k) u_i(\omega_k) \right) \right\}$$

- Equation for the *j*th numerator parameter that corresponds to input *l*:

$$0 = -2\sum_{k=1}^{n_f} \mathrm{Re}\left\{ \frac{\left|W(\omega_k)\right|^2 B_j^*(\omega_k) u_l^*(\omega_k)}{\left|D_{m-1}(\omega_k)\right|^2} \left(D_m(\omega_k) y(\omega_k) - \sum_{i=1}^{n_u} N_{i,m}(\omega_k) u_i(\omega_k) \right) \right\}$$

The first iteration is started with the best solution found for the numerators N_i and denominator D parameters during S-K iterations. Unlike S-K iterations, the basis functions $B_j(\omega)$ are not changed at each iteration, the iterations are performed with the basis functions that yielded the best solution in the S-K iterations. As before, the algorithm runs for a maximum of 20 iterations. The iterations are terminated early if the relative change in the value of the loss function is less than 0.001 in the last three iterations.

If you specify bounds on transfer function coefficients, these bounds are incorporated into the necessary optimality conditions via generalized Lagrange multipliers. The resulting constrained linear least-squares problems are solved using the same methods explained in the S-K iterations step.

4. Return the transfer function parameters corresponding to the optimal solution — Both the S-K and linear refinement iteration steps do not guarantee an improvement in the loss function value. The algorithm tracks the best parameter value observed during these steps, and returns these values.
5. Invert the bilinear mapping performed in step 1.
6. Perform an iterative refinement of the transfer function parameters using the nonlinear least-squares search method specified in the `SearchMethod` estimation option. This step is implemented in the following situations:

   - When you specify the `EnforceStability` estimation option as `true` (stability is requested), and the result of step 5 of this algorithm is an unstable model. The unstable poles are reflected inside the stability boundary and the resulting parameters are iteratively refined. For information about estimation options, see `tfestOptions`.

- When you add a regularization penalty to the loss function using the `Regularization` estimation option. For an example about regularization, see "Regularized Identification of Dynamic Systems".
- You estimate a continuous-time model using discrete-time data (see "Discrete-Time Transfer Function Estimation Using Discrete-Time Frequency-Domain Data" on page 1-1540).
- You use frequency domain input-output data to identify a multi-input model.

If you are using the estimation algorithm from R2016a or earlier (see "Compatibility" on page 1-1535) for estimating a continuous-time model using continuous-time frequency-domain data, then for continuous-time data and fixed delays, the Output-Error algorithm is used for model estimation. For continuous-time data and free delays, the state-space estimation algorithm is used. In this algorithm, the model coefficients are initialized using the N4SID estimation method. This initialization is followed by nonlinear least-squares search based updates to minimize a weighted prediction error norm.

Discrete-Time Transfer Function Estimation Using Discrete-Time Frequency-Domain Data

The estimation algorithm is the same as for continuous-time transfer function estimation using continuous-time frequency-domain data, except discrete-time data is used.

If you are using the estimation algorithm from R2016a or earlier (see "Compatibility" on page 1-1535), the algorithm is the same as the algorithm for discrete-time transfer function estimation using time-domain data.

Note: The software does not support estimation of a discrete-time transfer function using continuous-time frequency-domain data.

Continuous-Time Transfer Function Estimation Using Discrete-Time Frequency-Domain Data

`tfest` command first estimates a discrete-time model from the discrete-time data. The estimated model is then converted to a continuous-time model using the `d2c` command.

The frequency response of the resulting continuous-time model is then computed over the frequency grid of the estimation data. A continuous-time model of the desired (user-specified) structure is then fit to this frequency response. The estimation algorithm for using the frequency-response data to obtain the continuous-time model is the same as that for continuous-time transfer function estimation using continuous-time data.

If you are using the estimation algorithm from R2016a or earlier (see “Compatibility” on page 1-1535), the state-space estimation algorithm is used for estimating continuous-time models from discrete-time data. In this algorithm, the model coefficients are initialized using the N4SID estimation method. This initialization is 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 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. Thus, the delay values are not iteratively updated.
- For an initial model, `init_sys`, with:
  - `init_sys.Structure.IODelay.Value` specified as finite values
  - `init_sys.Structure.IODelay.Free` specified as `true`

  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. To do so, use physical insight of the process being modeled and functions such as `arxstruc`, `delayest`, and `impulseest`. For an example of determining input delay, see “Model Structure Selection: Determining Model Order and Input Delay”.

- “What are Transfer Function Models?”
- “Regularized Estimates of Model Parameters”
- “Estimating Models Using Frequency-Domain Data”

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,* 2003, Vol. 76, Issue 13, pp 1337–1357.

[2] Ljung, L. “Experiments With Identification of Continuous-Time Models.” *Proceedings of the 15th IFAC Symposium on System Identification.* 2009.

[3] Young, P. C. and A.J. Jakeman. “Refined instrumental variable methods of time-series analysis: Part III, extensions.” *International Journal of Control 31*, 1980, pp 741–764.

[4] Drmac, Z., S. Gugercin, and C. Beattie. “Quadrature-based vector fitting for discretized H_2 approximation.” *SIAM Journal on Scientific Computing*. Vol. 37, Numer 2, 2014, pp A625–A652.

See Also

`ar` | `arx` | `bj` | `greyest` | `idtf` | `oe` | `polyest` | `procest` | `ssest` | `tfestOptions`

Introduced in R2012a

tfestOptions

Option set for `tfest`

Syntax

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

Description

`opt = tfestOptions` creates the default option 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. `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 numerator and denominator
`'iv'` (default) | `'svf'` | `'gpmf'` | `'n4sid'` | `'all'`

Algorithm used to initialize the values of the numerator and denominator of the output of `tfest`, applicable only for estimation of continuous-time transfer functions using time-domain data, specified as one of 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 preceding approaches. The software tries all these methods and selects the method that yields the smallest value of prediction error norm.

`'InitOption'` — Option set for the initialization algorithm

structure

Option set for the initialization algorithm used to initialize the values of the numerator and denominator of the output of `tfest`, specified as 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 specified as one of the following values:

  - `'MOESP'` — Uses the MOESP algorithm by Verhaegen.
  - `'CVA'` — Uses the canonical variable algorithm (CVA) by Larimore.
  - `'SSARX'` — A subspace identification method that uses an ARX estimation based algorithm to compute the weighting.

    Specifying this option allows unbiased estimates when using data that is collected in closed-loop operation. For more information about the algorithm, see [6].
  - `'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. 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 more information. 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 used by the `iv`, `svf`, and `gpmf` initialization methods (see [4] and [5]).

  `FilterTimeConstant` specifies the cutoff frequency of the differentiating filter, F_{cutoff}, as:

  $$F_{cutoff} = \frac{\text{FilterTimeConstant}}{T_s}$$

  T_s is the sample time of the estimation data.

  Specify `FilterTimeConstant` as a positive number, typically less than 1. A good value of `FilterTimeConstant` is the ratio of T_s to the dominating time constant of the system.

  Default: `0.1`

- `MaxIter` — Maximum number of iterations. Applicable when `InitMethod` is `'iv'`.

  Default: `30`

- `Tolerance` — Convergence tolerance. Applicable when `InitMethod` is `'iv'`.

  Default: `0.01`

`'InitialCondition'` — Handling of initial conditions

`'auto'` (default) | `'zero'` | `'estimate'` | `'backcast'`

Handling of initial conditions during estimation, specified as one of the following values:

- `'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 preceding options is made, guided by the data.

`'WeightingFilter'` — Weighting prefilter

`[]` (default) | vector | matrix | cell array | linear system | `'inv'` | `'invsqrt'`

Weighting prefilter applied to the loss function to be minimized during estimation. To understand the effect of `WeightingFilter` on the loss function, see "Loss Function and Model Quality Metrics".

Specify `WeightingFilter` as one of the following values:

- `[]` — No weighting prefilter is used.
- Passbands — Specify a row vector or matrix containing frequency values that define desired passbands. You select a frequency band where the fit between estimated model and estimation data is optimized. For example, `[wl,wh]` where `wl` and `wh` represent lower and upper limits of a passband. For a matrix with several rows defining frequency passbands, `[w1l,w1h;w2l,w2h;w3l,w3h;...]`, the estimation algorithm uses the union of the frequency ranges to define the estimation passband.

  Passbands are expressed in `rad/TimeUnit` for time-domain data and in `FrequencyUnit` for frequency-domain data, where `TimeUnit` and `FrequencyUnit` are the time and frequency units of the estimation data.
- SISO filter — Specify a single-input-single-output (SISO) linear filter in one of the following ways:
  - A SISO LTI model
  - `{A,B,C,D}` format, which specifies the state-space matrices of a filter with the same sample time as estimation data.
  - `{numerator,denominator}` format, which specifies the numerator and denominator of the filter as a transfer function with the same sample time as estimation data.

    This option calculates the weighting function as a product of the filter and the input spectrum to estimate the transfer function.
- Weighting vector — Applicable for frequency-domain data only. Specify a column vector of weights. This vector must have the same length as 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.

- `'inv'` — Applicable for estimation using frequency-response data only. Use $1/|(G(jw))|$ as the weighting filter. Where $G(jw)$ is the complex frequency-response data. Use this option for capturing relatively low amplitude dynamics in data, or for fitting data with high modal density. This option also makes it easier to specify channel-dependent weighting filters for MIMO frequency-response data.
- `'invsqrt'` — Applicable for estimation using frequency-response data only. Use $1/\sqrt{|G(jw)|}$ as the weighting filter. Use this option for capturing relatively low amplitude dynamics in data, or for fitting data with high modal density. This option also makes it easier to specify channel-dependent weighting filters for MIMO frequency-response data.

`'EnforceStability'` — Control whether to enforce stability of model

`false` (default) | `true`

Control whether to enforce stability of estimated model, specified as the comma-separated pair consisting of `'EnforceStability'` and either `true` or `false`.

Use this option when estimating models using frequency-domain data. Models estimated using time-domain data are always stable.

Data Types: `logical`

`'EstCovar'` — Control whether to generate parameter covariance data

`true` (default) | `false`

Controls whether parameter covariance data is generated, specified as `true` or `false`.

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

`'Display'` — Specify whether to display the estimation progress

`'off'` (default) | `'on'`

Specify whether to display the estimation progress, specified as one of the following values:

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

`'InputOffset'` — Removal of offset from time-domain input data during estimation

[ ] (default) | vector of positive integers | matrix

Removal of offset from time-domain input data during estimation, specified as the comma-separated pair consisting of `'InputOffset'` and one of the following:

- A column vector of positive integers of length *Nu*, where *Nu* is the number of inputs.
- `[]` — Indicates no offset.
- *Nu*-by-*Ne* matrix — For multi-experiment data, specify `InputOffset` as an *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.

`'OutputOffset'` — Removal of offset from time-domain output data during estimation

[ ] (default) | vector | matrix

Removal of offset from time-domain output data during estimation, specified as the comma-separated pair consisting of `'OutputOffset'` and one of the following:

- A column vector of length *Ny*, where *Ny* is the number of outputs.
- `[]` — Indicates no offset.
- *Ny*-by-*Ne* matrix — For multi-experiment 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.

`'OutputWeight'` — Weighting of prediction errors in multi-output estimations

`[]` (default) | `'noise'` | positive semidefinite symmetric matrix

Weighting of prediction errors in multi-output estimations, specified as one of the following values:

- `'noise'` — Minimize $\det(E'^* E / N)$, where E represents the prediction error and `N` is the number of data samples. This choice is optimal in a statistical sense and leads to maximum likelihood estimates if 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 semidefinite symmetric matrix (`W`) — Minimize the trace of the weighted prediction error matrix `trace(E'*E*W/N)` where:
  - E is the matrix of prediction errors, with one column for each output, and 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-output models, or the reliability of corresponding data.
  - `N` is the number of data samples.
- `[]` — The software chooses between the `'noise'` or using the identity matrix for `W`.

This option is relevant for only multi-output models.

`'Regularization'` — Options for regularized estimation of model parameters

structure

Options for regularized estimation of model parameters. For more information on regularization, see “Regularized Estimates of Model Parameters”.

`Regularization` is a structure with the following fields:

- `Lambda` — Constant that determines the bias versus variance tradeoff.

  Specify a positive scalar to add the regularization term to the estimation cost.

  The default value of zero implies no regularization.

  Default: 0
- `R` — Weighting matrix.

  Specify a vector of nonnegative numbers or a square positive semi-definite matrix. The length must be equal to the number of free parameters of the model.

  For black-box models, using the default value is recommended. For structured and grey-box models, you can also specify a vector of `np` positive numbers such that each entry denotes the confidence in the value of the associated parameter.

  The default value of 1 implies a value of `eye(npfree)`, where `npfree` is the number of free parameters.

  Default: 1
- `Nominal` — The nominal value towards which the free parameters are pulled during estimation.

The default value of zero implies that the parameter values are pulled towards zero. If you are refining a model, you can set the value to `'model'` to pull the parameters towards the parameter values of the initial model. The initial parameter values must be finite for this setting to work.

Default: 0

`'SearchMethod'` — Numerical search method used for iterative parameter estimation

`'auto'` (default) | `'gn'` | `'gna'` | `'lm'` | `'grad'` | `'lsqnonlin'`

Numerical search method used for iterative parameter estimation, specified as one of the following values:

- `'gn'` — The subspace Gauss-Newton direction. Singular values of the Jacobian matrix less than `GnPinvConst*eps*max(size(J))*norm(J)` are discarded when computing the search direction. 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 `gamma*max(sv)` of the Hessian are ignored, 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). *gamma* is increased by the factor `LMStep` each time the search fails to find a lower value of the criterion in less than 5 bisections. *gamma* is decreased by the factor `2*LMStep` each time a search is successful without any bisections.
- `'lm'` — Uses the Levenberg-Marquardt method, so that the next parameter value is `-pinv(H+d*I)*grad` from the previous one, where H is the Hessian, I is the identity matrix, and *grad* is the gradient. d is a number that is increased until a lower value of the criterion is found.
- `'lsqnonlin'` — Uses `lsqnonlin` optimizer from Optimization Toolbox software. This search method can only handle the Trace criterion.
- `'grad'` — The steepest descent gradient search method.
- `'auto'` — A choice among the preceding options is made in the algorithm. The descent direction is calculated using `'gn'`, `'gna'`, `'lm'`, and `'grad'` successively, at each iteration until a sufficient reduction in error is achieved.

`'SearchOption'` — Option set for the search algorithm

search option set

Option set for the search algorithm with fields that depend on the value of `SearchMethod`.

SearchOption structure when SearchMethod is specified as 'gn', 'gna', 'lm', 'grad', or 'auto'

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td><code>Toleran</code></td><td>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 <code>Tolerance</code>, the iterations stop. The estimate of the expected loss-function improvement at the next iteration is based on the Gauss-Newton vector computed for the current parameter value.

Default: <code>0.01</code></td></tr>
<tr><td><code>MaxIter</code></td><td>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>Tolerance</code>.

Setting <code>MaxIter = 0</code> returns the result of the start-up procedure.

Use <code>sys.Report.Termination.Iterations</code> to get the actual number of iterations during an estimation, where <i>sys</i> is an <code>idtf</code> model.

Default: <code>20</code></td></tr>
<tr><td><code>Advance</code></td><td>Advanced search settings.

Specified as a structure with the following fields:

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td><code>GnPinvCon</code></td><td>Singular values of the Jacobian matrix that are smaller than <code>GnPinvConst*max(size(J)*norm(J)*eps)</code> are discarded when computing the search direction. Applicable when <code>SearchMethod</code> is <code>'gn'</code>.

<code>GnPinvConst</code> must be a positive, real value.

Default: <code>10000</code></td></tr>
<tr><td><code>InitGnaTo</code></td><td>Initial value of <i>gamma</i>. Applicable when <code>SearchMethod</code> is <code>'gna'</code>.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>Default: `0.0001`</td></tr>
<tr><td>`LMStartVa`</td><td>Starting value of search-direction length <i>d</i> in the Levenberg-Marquardt method. Applicable when `SearchMethod` is `'lm'`.
Default: `0.001`</td></tr>
<tr><td>`LMStep`</td><td>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'`.
Default: `2`</td></tr>
<tr><td>`MaxBisect`</td><td>Maximum number of bisections used by the line search along the search direction.
Default: `25`</td></tr>
<tr><td>`MaxFunEva`</td><td>Iterations stop if the number of calls to the model file exceeds this value.
`MaxFunEvals` must be a positive, integer value.
Default: `Inf`</td></tr>
<tr><td>`MinParCha`</td><td>Smallest parameter update allowed per iteration.
`MinParChange` must be a positive, real value.
Default: `0`</td></tr>
<tr><td>`RelImprov`</td><td>Iterations stop if the relative improvement of the criterion function is less than `RelImprovement`.
`RelImprovement` must be a positive, integer value.
Default: `0`</td></tr>
<tr><td>`StepReduc`</td><td>Suggested parameter update is reduced by the factor `StepReduction` after each try. This reduction continues until</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field Name</th><th>Description</th></tr>
<tr><td></td><td>either <code>MaxBisections</code> tries are completed or a lower value of the criterion function is obtained.

<code>StepReduction</code> must be a positive, real value that is greater than 1.

Default: <code>2</code></td></tr>
</table>
</td></tr>
</table>

SearchOption structure when SearchMethod is specified as 'lsqnonlin'

Field Name	Description
`TolFun`	Termination tolerance on the loss function that the software minimizes to determine the estimated parameter values. The value of `TolFun` is the same as that of `opt.SearchOption.Advanced.TolFun`. **Default:** `1e-5`
`TolX`	Termination tolerance on the estimated parameter values. The value of `TolX` is the same as that of `opt.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 `opt.SearchOption.Advanced.MaxIter`. **Default:** `20`
`Advance`	Options set for `lsqnonlin`.

Field Name	Description
	For more information, see the Optimization Options table in "Optimization Options". Use `optimset('lsqnonlin')` to create an options set for `lsqnonlin`, and then modify it to specify its various options.

`'Advanced'` — Additional advanced options

structure

Additional advanced options, specified as 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 loss function. The standard deviation is estimated robustly as the median of the absolute deviations from the median of the prediction errors, divided by `0.7`. For more information on robust norm choices, see section 15.2 of [1].

  `ErrorThreshold = 0` disables robustification and leads to a purely quadratic loss function. When estimating with frequency-domain data, the software sets `ErrorThreshold` to zero. For time-domain data that contains outliers, try setting `ErrorThreshold` to `1.6`.

  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 value.

  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.

  The initial condition is estimated when

  $$\frac{\left\| y_{p,z} - y_{meas} \right\|}{\left\| y_{p,e} - y_{meas} \right\|} > \text{AutoInitThreshold}$$

  - y_{meas} is the measured output.
  - $y_{p,z}$ is the predicted output of a model estimated using zero initial states.
  - $y_{p,e}$ is the predicted output of a model estimated using estimated initial states.

  Applicable when `InitialCondition` is `'auto'`.

  Default: `1.05`

Output Arguments

`opt` — Option set for `tfest`

`tfestOptions` option set

Option set for `tfest`, returned as an `tfestOptions` option set.

Examples

Create Default Options Set for Transfer Function Estimation

```
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';
```

More About

- “Loss Function and Model Quality Metrics”

References

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

[4] Garnier, H., M. Mensler, and A. Richard. “Continuous-time Model Identification From Sampled Data: Implementation Issues and Performance Evaluation” *International Journal of Control,* 2003, Vol. 76, Issue 13, pp 1337–1357.

[5] Ljung, L. “Experiments With Identification of Continuous-Time Models.” *Proceedings of the 15th IFAC Symposium on System Identification.* 2009.

[6] Jansson, M. “Subspace identification and ARX modeling.” *13th IFAC Symposium on System Identification* , Rotterdam, The Netherlands, 2003.

See Also

`tfest`

Introduced in R2012b

timeoptions

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 and System Identification 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 values: `'off'` | `'on'` **Default**: `'off'`
GridColor	Color of the grid lines, specified as one of the following: Vector of RGB values in the range `[0,1]`| character vector of color name |`'none'`. For example, for yellow color, specify as one of the following: `[1 1 0]`, `'yellow'`, or `'y'`. **Default**: `[0.15,0.15,0.15]`
XlimMode, YlimMode	Limit modes
Xlim, Ylim	Axes limits

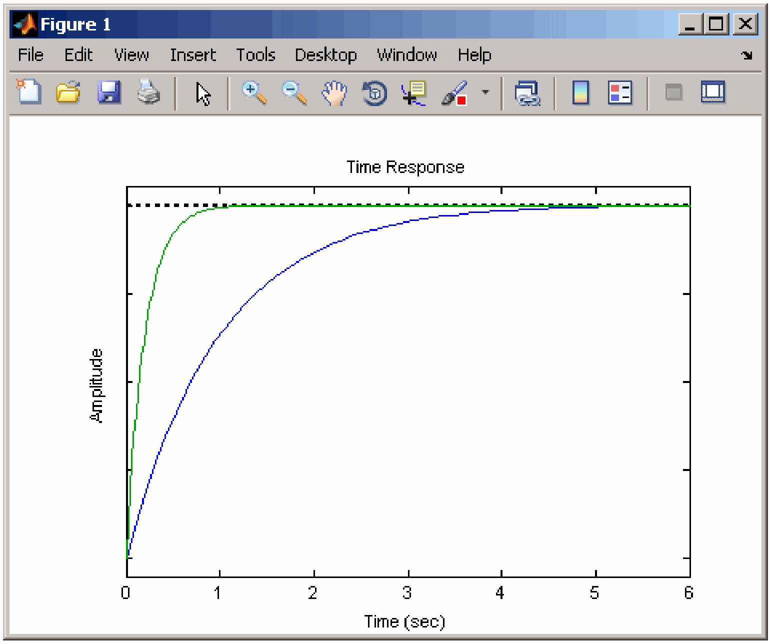
Option	Description
IOGrouping	Grouping of input-output pairs, specified as one of the following values: `'none'` |`'inputs'`|`'outputs'`|`'all'` **Default**: `'none'`
InputLabels, OutputLabels	Input and output label styles
InputVisible, OutputVisible	Visibility of input and output channels
Normalize	Normalize responses, specified as one of the following values: `'on'` |`'off'` **Default**: `'off'`
SettleTimeThreshold	Settling time threshold
RiseTimeLimits	Rise time limits
TimeUnits	Time units, specified as one of the following values: • `'nanoseconds'` • `'microseconds'` • `'milliseconds'` • `'seconds'` • `'minutes'` • `'hours'` • `'days'` • `'weeks'` • `'months'` • `'years'` **Default**: `'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` | `impulseplot` | `initialplot` | `lsimplot` | `setoptions` | `stepplot`

Introduced in R2012a

totaldelay

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`

Introduced in R2012a

translatecov

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.

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.

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)
```

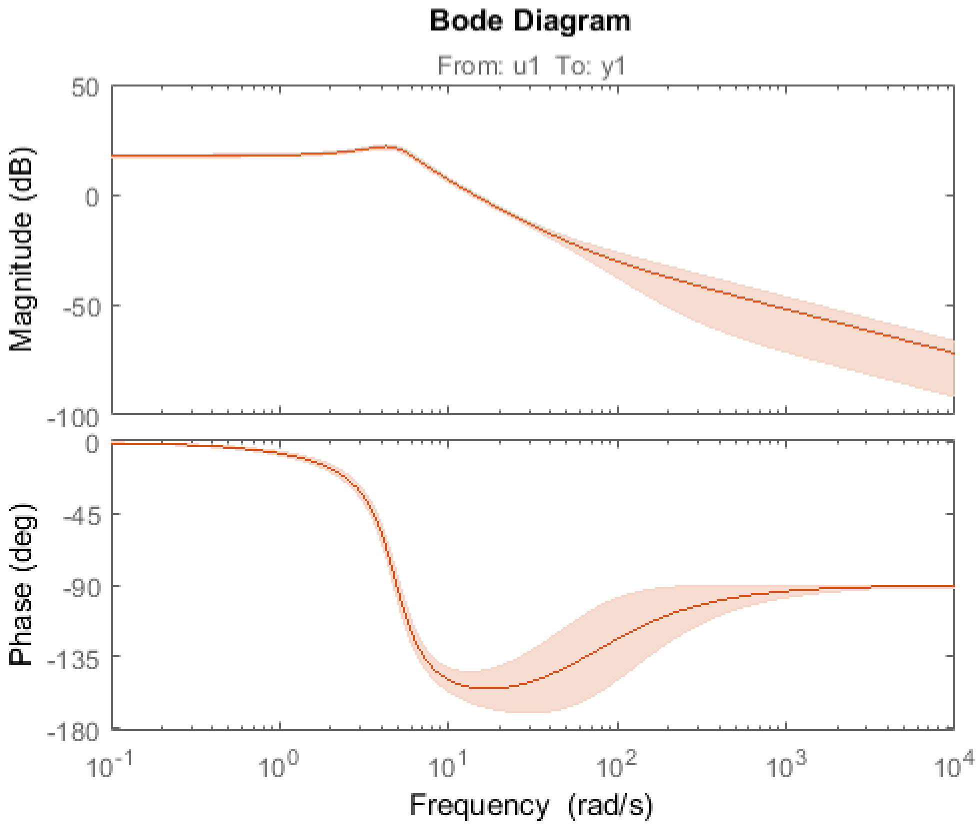
```
ans =

     []
```

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', 'none');
m2 = ssest(data(:,2,:),2,'feedthrough',true(1,2), 'DisturbanceModel', 'none');
m3 = ssest(data(:,3,:),2,'feedthrough',true(1,2), 'DisturbanceModel', 'none');
```

Combine the estimated models while also translating the covariance information.

```
f = @(x,y,z)[x;y;z];
M2 = translatecov(f, m1, m2, m3);
```

The parameter covariance is not empty.

```
getcov(M2, 'factors')
```

```
ans = 

  struct with fields:
```

```
       R: [36×36 double]
       T: [24×36 double]
    Free: [90×1 logical]
```

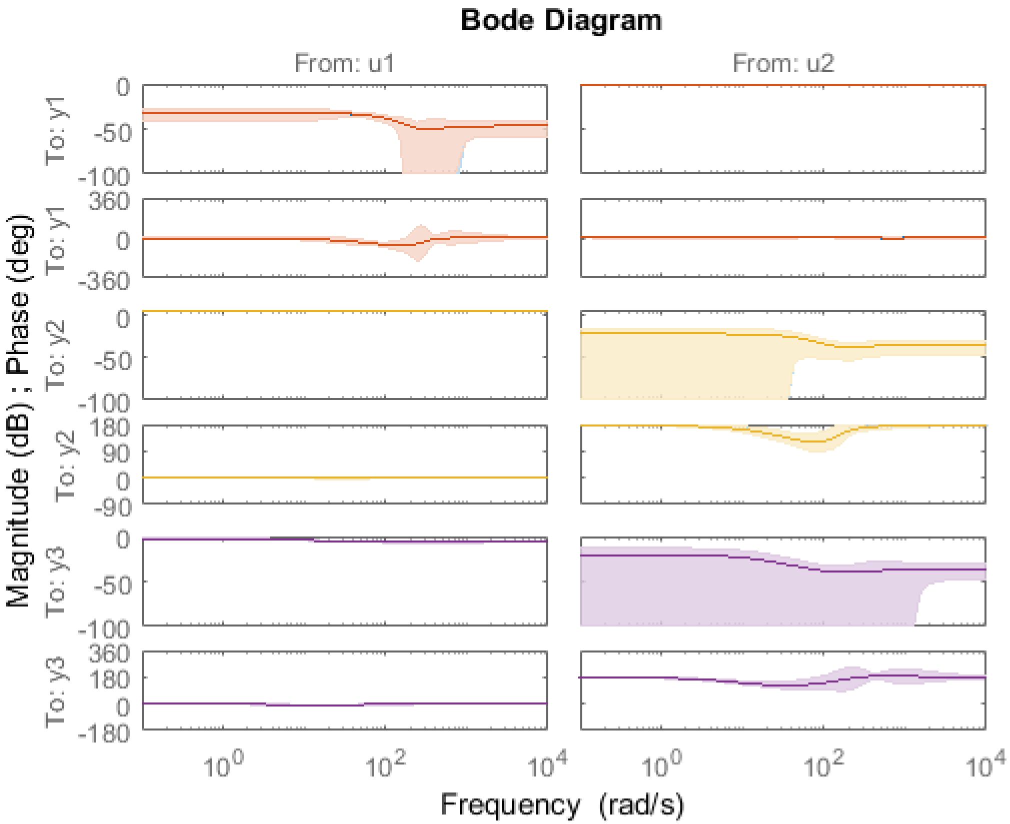
If you combine the estimated models into one 3-output model directly, the covariance information is lost (the output of `getcov` is empty).

```
M1 = [m1;m2;m3];
getcov(M1)
```

```
ans =

     []
```

Compare the confidence bounds.

```
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.

More About

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.

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 $cov\_new = J \times \mathrm{cov} \times J^T$ to translate the covariance, where `J` is the Jacobian matrix. This operation can be slow for models containing a large number of free parameters.

- “What Is Model Covariance?”
- “Types of Model Uncertainty Information”

See Also

`getcov` | `getpvec` | `rsample` | `setcov`

Introduced in R2012b

treepartition

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 1-1570. 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`	Integer specifies the number of nodes in the tree. Default=`'auto'` selects the number of nodes from the data using the pruning algorithm. When you estimate a model containing a `treepartition` nonlinearity, the value of `NumberOfUnits` is automatically changed to show the actual number of leaves used—which is the largest integer of the form `2^n-1` and less than or equal to `N` (the integer value of units you specify). For example: `treepartition('NumberOfUnits',5)`
`Parameters`	Structure containing the following fields: • `RegressorMean`: 1-by-`m` vector containing the means of `x` in estimation data, `r`. • `RegressorMinMax`: `m`-by-2 matrix containing the maximum and minimum estimation-data regressor values. • `OutputOffset`: scalar `d`. • `LinearCoef`: `m`-*by*-1 vector *L*. • `SampleLength`: Length of estimation data. • `NoiseVariance`: Estimated variance of the noise in estimation data. • `Tree`: A structure containing the following tree parameters:

Property Name	Description
	- `TreeLevelPntr`: `N`-*by*-1 vector containing the levels `j` of each node. - `AncestorDescendantPntr`: `N`-*by*-3 matrix, such that the entry `(k,1)` is the ancestor of node `k`, and entries `(k,2)` and `(k,3)` are the left and right descendants, respectively. - `LocalizingVectors`: `N`-*by*-`(m+1)` matrix, such that the `r`th row is `B_r`. - `LocalParVector`: `N`-*by*-`(m+1)` matrix, such that the `k`th row is `C_k`. - `LocalCovMatrix`: `N`-*by*-`((m+1)m/2)` matrix such that the `k`th row is the covariance matrix of `C_k`. `C_k` is reshaped as a row vector.
`Options`	Structure containing the following fields that affect the initial model: - `FinestCell`: Integer or character vector specifying the minimum number of data points in the smallest partition. Default: `'auto'`, which computes the value from the data. - `Threshold`: Threshold parameter used by the adaptive pruning algorithm. Smaller threshold value corresponds to a shorter branch that is terminated by the active partition `D_a`. Higher threshold value results in a longer branch. Default: `1.0`. - `Stabilizer`: Penalty parameter of the penalized least-squares algorithm used to compute local parameter vectors `C_k`. Higher stabilizer value improves stability, but may deteriorate the accuracy of the least-square estimate. Default: `1e-6`.

Examples

Use `treepartition` to specify the nonlinear estimator in nonlinear ARX models. For example:

```
m=nlarx(Data,Orders,treepartition('num',5));
```

The following commands provide an example 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);
```

More About

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 <=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*.

When the `Focus` option in `nlarxOptions` 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 estimators is not differentiable. Minimization of simulation error requires differentiable nonlinear functions.

See Also

`nlarx`

Introduced in R2007a

TrendInfo

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.

Property Name	Default	Description
`DataName`	`''`	Name of the `iddata` object from which trend information is derived (if any)
`InputOffset`	`zeros(1,nu)`, where `nu` is the number of inputs	• For transient data, the physical equilibrium offset you specify for each input signal. • For steady-state data, the mean of input values. Computed automatically when detrending the data. • If removing a linear trend from the input-output data, the value of the line at `t0`, where `t0` is the start time. For multiple experiment data, this is a cell array of size equal to the number of experiments in the data set.
`InputSlope`	`zeros(1,nu)`, where `nu` is the number of inputs	Slope of linear trend in input data, computed automatically when using the `detrend` command to remove the linear trend in the data. For multiple experiment data, this is a cell array of size equal to the number of experiments in the data set.
`OutputOffset`	`zeros(1,ny)`, where `ny` is the number of outputs	• For transient data, the physical equilibrium offset you specify for each output signal • For steady-state data, the mean of output values. Computed automatically when detrending the data. • If removing a linear trend from the input-output data, the value of the line at `t0`, where `t0` is the start time. For multiple experiment data, this is a cell array of size equal to the number of experiments in the data set.
`OutputSlope`	`zeros(1,ny)`, where `ny` is the number of outputs	Slope of linear trend in output data, computed automatically when using the `detrend` command to remove the linear trend in the data. For multiple experiment data, this is a cell array of size equal to the number of experiments in the data set.

Examples

Remove Offsets From Data

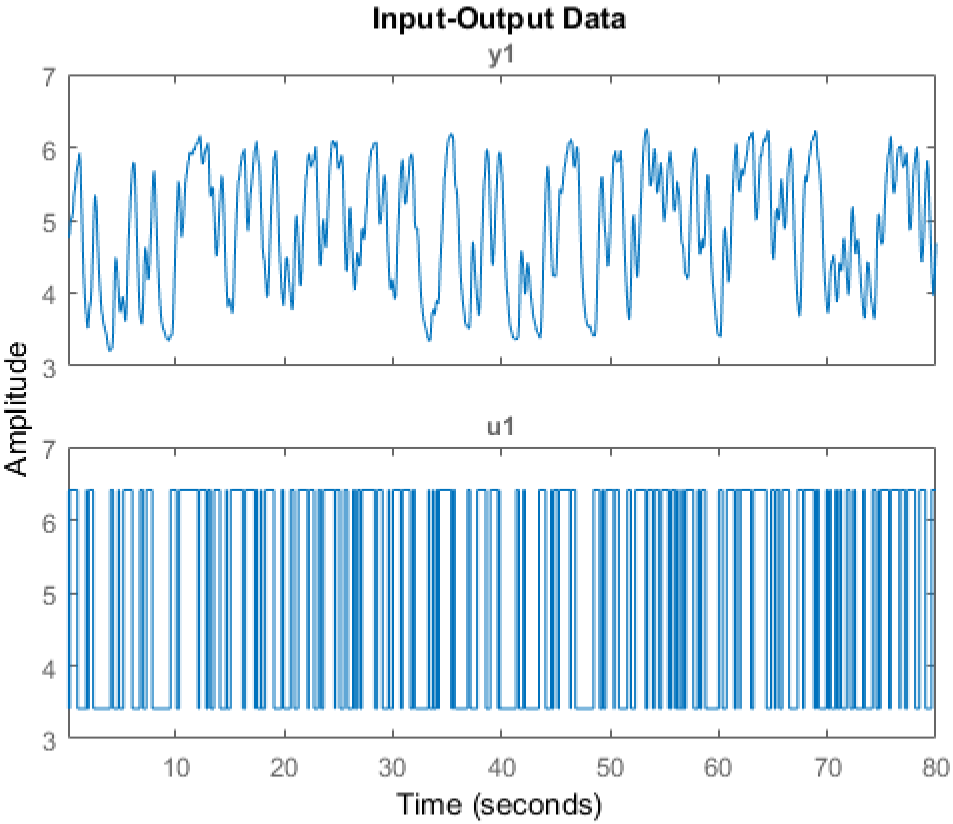
Remove specified offset from input and output signals.

Load SISO data containing vectors `u2` and `y2`.

```
load dryer2
```

Create a data object with sample time of 0.08 seconds and plot it.

```
data = iddata(y2,u2,0.08);
plot(data)
```



The data has a nonzero mean value.

Store the data offset and trend information in a `TrendInfo` object.

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

Assign offset values to the `TrendInfo` object.

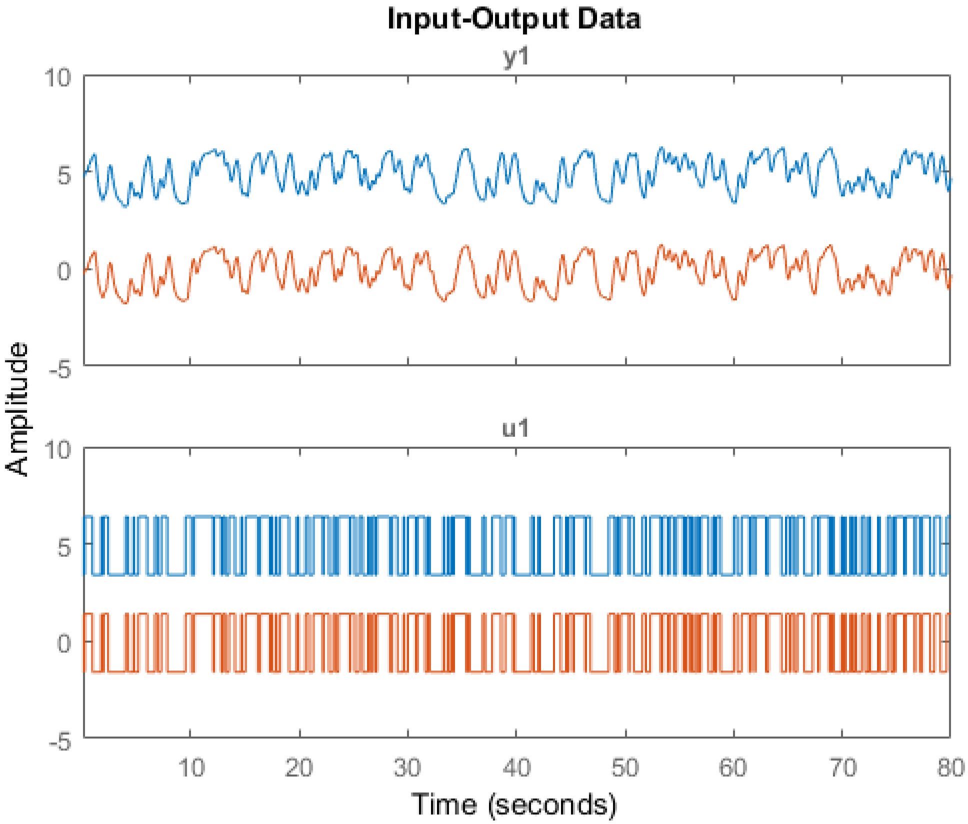
```
T.InputOffset = 5;
T.OutputOffset = 5;
```

Subtract offset from the data.

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

Plot the detrended data on the same plot.

```
hold on
plot(data_d)
```



View the mean value removed from the data.

```
get(T)
```

```
ans = 

  struct with fields:

        DataName: 'data'
     InputOffset: 5
    OutputOffset: 5
      InputSlope: 0
     OutputSlope: 0
```

Store Trend Information

Construct the `TrendInfo` object that stores trend information as part of data detrending.

Load SISO data containing vectors `u2` and `y2`.

```
load dryer2
```

Create data object with sample time of 0.08 seconds.

```
data = iddata(y2,u2,0.08);
```

Detrend the mean from the data and store the mean as a `TrendInfo` object `T`.

```
[data_d,T] = detrend(data,0);
```

View the mean value removed from the data.

```
get(T)
```

```
ans = 

  struct with fields:

        DataName: 'data'
     InputOffset: 5.0000
    OutputOffset: 4.8901
      InputSlope: 0
     OutputSlope: 0
```

More About

- "Handling Offsets and Trends in Data"

See Also

`detrend` | `getTrend` | `retrend`

Introduced in R2009a

unitgain

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.

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 model with a sigmoid nonlinearity at the output and a unit gain at the input, use the following command:

```
m = nlhw(Data,Orders,[],'sigmoid');
```

More About

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 $F(x)=x$.

See Also

`deadzone` | `nlhw` | `saturation` | `sigmoidnet`

Introduced in R2007a

unscentedKalmanFilter

Create unscented Kalman filter object for online state estimation

Syntax

```
obj = unscentedKalmanFilter(StateTransitionFcn,MeasurementFcn,
InitialState)
obj = unscentedKalmanFilter(StateTransitionFcn,MeasurementFcn,
InitialState,Name,Value)
obj = unscentedKalmanFilter(Name,Value)
```

Description

`obj = unscentedKalmanFilter(StateTransitionFcn,MeasurementFcn, InitialState)` creates an unscented Kalman filter object for online state estimation of a discrete-time nonlinear system. `StateTransitionFcn` is a function that calculates the state of the system at time *k*, given the state vector at time *k*-1. `MeasurementFcn` is a function that calculates the output measurement of the system at time *k*, given the state at time *k*. `InitialState` specifies the initial value of the state estimates.

After creating the object, use the `correct` and `predict` commands to update state estimates and state estimation error covariance values using a discrete-time unscented Kalman filter algorithm and real-time data.

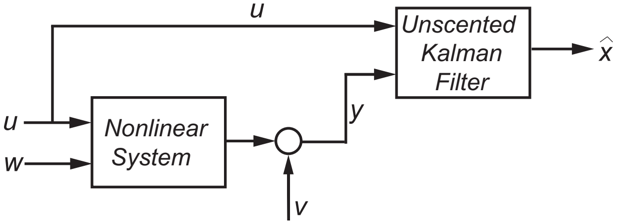
`obj = unscentedKalmanFilter(StateTransitionFcn,MeasurementFcn, InitialState,Name,Value)` specifies additional attributes of the unscented Kalman filter object using one or more `Name,Value` pair arguments.

`obj = unscentedKalmanFilter(Name,Value)` creates an unscented Kalman filter object with properties specified using one or more `Name,Value` pair arguments. The properties that you do not specify retain their default value.

Object Description

`unscentedKalmanFilter` creates an object for online state estimation of a discrete-time nonlinear system using the discrete-time unscented Kalman filter algorithm.

Consider a plant with states x, input u, output y, process noise w, and measurement noise v. Assume that you can represent the plant as a nonlinear system.



The algorithm computes the state estimates $\hat{x}$ of the nonlinear system using state transition and measurement functions specified by you. The software lets you specify the noise in these functions as additive or nonadditive:

- **Additive Noise Terms** — The state transition and measurements equations have the following form:

  $$x[k] = f(x[k-1], u_s[k-1]) + w[k-1]$$
  $$y[k] = h(x[k], u_m[k]) + v[k]$$

  Here f is a nonlinear state transition function that describes the evolution of states `x` from one time step to the next. The nonlinear measurement function h relates `x` to the measurements `y` at time step `k`. `w` and `v` are the zero-mean, uncorrelated process and measurement noises, respectively. These functions can also have additional input arguments that are denoted by u_s and u_m in the equations. For example, the additional arguments could be time step `k` or the inputs `u` to the nonlinear system. There can be multiple such arguments.

  Note that the noise terms in both equations are additive. That is, `x(k)` is linearly related to the process noise `w(k-1)`, and `y(k)` is linearly related to the measurement noise `v(k)`.

- **Nonadditive Noise Terms** — The software also supports more complex state transition and measurement functions where the state $x[k]$ and measurement $y[k]$ are nonlinear functions of the process noise and measurement noise, respectively. When the noise terms are nonadditive, the state transition and measurements equation have the following form:

$$x[k] = f(x[k-1], w[k-1], u_s[k-1])$$
$$y[k] = h(x[k], v[k], u_m[k])$$

When you perform online state estimation, you first create the nonlinear state transition function *f* and measurement function *h*. You then construct the `unscentedKalmanFilter` object using these nonlinear functions, and specify whether the noise terms are additive or nonadditive. After you create the object, you use the `predict` command to predict state estimates at the next time step, and `correct` to correct state estimates using the unscented Kalman filter algorithm and real-time data. For information about the algorithm, see "Extended and Unscented Kalman Filter Algorithms for Online State Estimation".

You can use the following commands with `unscentedKalmanFilter` objects:

Command	Description
`correct`	Correct the state and state estimation error covariance at time step *k* using measured data at time step *k*.
`predict`	Predict the state and state estimation error covariance at time the next time step.
`clone`	Create another object with the same object property values. Do not create additional objects using syntax `obj2 = obj`. Any changes made to the properties of the new object created in this way (`obj2`) also change the properties of the original object (`obj`).

For `unscentedKalmanFilter` object properties, see "Properties" on page 1-1591.

Examples

Create Unscented Kalman Filter Object for Online State Estimation

To define an unscented Kalman filter object for estimating the states of your system, you write and save the state transition function and measurement function for the system.

In this example, use the previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`. These functions describe a discrete-approximation to van der Pol oscillator with nonlinearity parameter, mu, equal to 1. The oscillator has two states.

Specify an initial guess for the two states. You specify the initial state guess as an `M`-element row or column vector, where `M` is the number of states.

```
initialStateGuess = [1;0];
```

Create the unscented Kalman filter object. Use function handles to provide the state transition and measurement functions to the object.

```
obj = unscentedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,initialStateGuess);
```

The object has a default structure where the process and measurement noise are additive.

To estimate the states and state estimation error covariance from the constructed object, use the `correct` and `predict` commands and real-time data.

Specify Process and Measurement Noise Covariances in Unscented Kalman Filter Object

Create an unscented Kalman filter object for a van der Pol oscillator with two states and one output. Use the previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`. These functions are written for additive process and measurement noise terms. Specify the initial state values for the two states as [2;0].

Since the system has two states and the process noise is additive, the process noise is a 2-element vector and the process noise covariance is a 2-by-2 matrix. Assume there is no cross-correlation between process noise terms, and both the terms have the same variance 0.01. You can specify the process noise covariance as a scalar. The software uses the scalar value to create a 2-by-2 diagonal matrix with 0.01 on the diagonals.

Specify the process noise covariance during object construction.

```
obj = unscentedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,[2;0],...
    'ProcessNoise',0.01);
```

Alternatively, you can specify noise covariances after object construction using dot notation. For example, specify the measurement noise covariance as 0.2.

```
obj.MeasurementNoise = 0.2;
```

Since the system has only one output, the measurement noise is a 1-element vector and the `MeasurementNoise` property denotes the variance of the measurement noise.

Specify Nonadditive Measurement Noise in Unscented Kalman Filter Object

Create an unscented Kalman filter object for a van der Pol oscillator with two states and one output. Assume that the process noise terms in the state transition function are additive. That is, there is a linear relation between the state and process noise. Also assume that the measurement noise terms are nonadditive. That is, there is a nonlinear relation between the measurement and measurement noise.

```
obj = unscentedKalmanFilter('HasAdditiveMeasurementNoise',false);
```

Specify the state transition function and measurement functions. Use the previously written and saved functions, `vdpStateFcn.m` and `vdpMeasurementNonAdditiveNoiseFcn.m`.

The state transition function is written assuming the process noise is additive. The measurement function is written assuming the measurement noise is nonadditive.

```
obj.StateTransitionFcn = @vdpStateFcn;
obj.StateTransitionFcn = @vdpMeasurementNonAdditiveNoiseFcn;
```

Specify the initial state values for the two states as [2;0].

```
obj.State = [2;0];
```

You can now use the `correct` and `predict` commands to estimate the state and state estimation error covariance values from the constructed object.

Specify Additional Inputs in State Transition and Measurement Functions

Consider a nonlinear system with input `u` whose state `x` and measurement `y` evolve according to the folowing state transition and measurement equations:

$$x[k] = \sqrt{x[k-1] + u[k-1]} + w[k-1]$$

$$y[k] = x[k] + 2 * u[k] + v[k]^2$$

The process noise `w` of the system is additive while the measurement noise `v` is nonadditive.

Create the state transition function and measurement function for the system. Specify the functions with an additional input `u`.

```
f = @(x,u)(sqrt(x+u));
h = @(x,v,u)(x+2*u+v^2);
```

`f` and `h` are function handles to the anonymous functions that store the state transition and measurement functions, respectively. In the measurement function, because the measurement noise is nonadditive, `v` is also specified as an input. Note that `v` is specified as an input before the additional input `u`.

Create an unscented Kalman filter object for estimating the state of the nonlinear system using the specified functions. Specify the initial value of the state as 1, and the measurement noise as nonadditive.

```
obj = unscentedKalmanFilter(f,h,1,'HasAdditiveMeasurementNoise',false);
```

Specify the measurement noise covariance.

```
obj.MeasurementNoise = 0.01;
```

You can now estimate the state of the system using the `predict` and `correct` commands. You pass the values of `u` to `predict` and `correct`, which in turn pass them to the state transition and measurement functions, respectively.

Correct the state estimate with measurement y[k]=0.8 and input u[k]=0.2 at time step k.

```
correct(obj,0.8,0.2)
```

Predict the state at next time step, given u[k]=0.2.

```
predict(obj,0.2)
```

- “Nonlinear State Estimation Using Unscented Kalman Filter”
- “Fault Detection Using an Extended Kalman Filter”
- “Generate Code for Online State Estimation in MATLAB”

Input Arguments

`StateTransitionFcn` — State transition function

function handle

State transition function *f*, specified as a function handle. The function calculates the *M*-element state vector of the system at time step *k*, given the state vector at time step *k*-1. *M* is the number of states of the nonlinear system.

You write and save the state transition function for your nonlinear system, and use it to construct the object. For example, if `vdpStateFcn.m` is the state transition function, specify `StateTransitionFcn` as `@vdpStateFcn`. You can also specify `StateTransitionFcn` as a function handle to an anonymous function.

The inputs to the function you write depend on whether you specify the process noise as additive or nonadditive in the `HasAdditiveProcessNoise` property of the object:

- `HasAdditiveProcessNoise` is true — The process noise `w` is additive, and the state transition function specifies how the states evolve as a function of state values at the previous time step:

  ```
  x(k) = f(x(k-1),Us1,...,Usn)
  ```

  Where `x(k)` is the estimated state at time `k`, and `Us1,...,Usn` are any additional input arguments required by your state transition function, such as system inputs or the sample time. During estimation, you pass these additional arguments to the `predict` command, which in turn passes them to the state transition function.
- `HasAdditiveProcessNoise` is false — The process noise is nonadditive, and the state transition function also specifies how the states evolve as a function of the process noise:

  ```
  x(k) = f(x(k-1),w(k-1),Us1,...,Usn)
  ```

To see an example of a state transition function with additive process noise, type `edit vdpStateFcn` at the command line.

`MeasurementFcn` — Measurement function

function handle

Measurement function *h*, specified as a function handle. The function calculates the *N*-element output measurement vector of the nonlinear system at time step *k*, given the state vector at time step *k*. *N* is the number of measurements of the system. You write and save the measurement function, and use it to construct the object. For example, if `vdpMeasurementFcn.m` is the measurement function, specify `MeasurementFcn` as `@vdpMeasurementFcn`. You can also specify `MeasurementFcn` as a function handle to an anonymous function.

The inputs to the function depend on whether you specify the measurement noise as additive or nonadditive in the `HasAdditiveMeasurementNoise` property of the object:

- `HasAdditiveMeasurementNoise` is true — The measurement noise `v` is additive, and the measurement function specifies how the measurements evolve as a function of state values:

  ```
  y(k) = h(x(k),Um1,...,Umn)
  ```

  Where `y(k)` and `x(k)` are the estimated output and estimated state at time `k`, and `Um1,...,Umn` are any optional input arguments required by your measurement function. For example, if you are using multiple sensors for tracking an object, an additional input could be the sensor position. During estimation, you pass these additional arguments to the `correct` command, which in turn passes them to the measurement function.
- `HasAdditiveMeasurementNoise` is false — The measurement noise is nonadditive, and the measurement function also specifies how the output measurement evolves as a function of the measurement noise:

  ```
  y(k) = h(x(k),v(k),Um1,...,Umn)
  ```

To see an example of a measurement function with additive process noise, type `edit vdpMeasurementFcn` at the command line. To see an example of a measurement function with nonadditive process noise, type `edit vdpMeasurementNonAdditiveNoiseFcn`.

`InitialState` — Initial state estimates

vector

Initial state estimates, specified as an *M*-element vector, where *M* is the number of states in the system. Specify the initial state values based on your knowledge of the system.

The specified value is stored in the `State` property of the object. If you specify `InitialState` as a column vector then `State` is also a column vector, and `predict` and `correct` commands return state estimates as a column vector. Otherwise, a row vector is returned.

If you want a filter with single-precision floating-point variables, specify `InitialState` as a single-precision vector variable. For example, for a two-state system with state transition and measurement functions `vdpStateFcn.m` and `vdpMeasurementFcn.m`, create the unscented Kalman filter object with initial states `[1;2]` as follows:

```
obj = unscentedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,single([1;2]))
```

Data Types: `double` | `single`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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 properties of `unscentedKalmanFilter` object during object creation. For example, to create an unscented Kalman filter object and specify the process noise covariance as 0.01:

```
obj = unscentedKalmanFilter(StateTransitionFcn,MeasurementFcn,InitialState,'ProcessNoi
```

Properties

`unscentedKalmanFilter` object properties are of three types:

- Tunable properties that you can specify multiple times, either during object construction using `Name,Value` arguments, or any time afterwards during state estimation. After object creation, use dot notation to modify the tunable properties.

  ```
  obj = unscentedKalmanFilter(StateTransitionFcn,MeasurementFcn,InitialState);
  obj.ProcessNoise = 0.01;
  ```

  The tunable properties are `State`, `StateCovariance`, `ProcessNoise`, `MeasurementNoise`, `Alpha`, `Beta`, and `Kappa`.
- Nontunable properties that you can specify once, either during object construction or afterward using dot notion. Specify these properties before state estimation using `correct` and `predict`. The `StateTransitionFcn` and `MeasurementFcn` properties belong to this category.
- Nontunable properties that you must specify during object construction. The `HasAdditiveProcessNoise` and `HasAdditiveMeasurementNoise` properties belong to this category.

`Alpha` — Spread of sigma points

`1e-3` (default) | scalar value between 0 and 1

Spread of sigma points around mean state value, specified as a scalar value between 0 and 1 (`0` < `Alpha` <= `1`).

The unscented Kalman filter algorithm treats the state of the system as a random variable with mean value `State` and variance `StateCovariance`. To compute the state and its statistical properties at the next time step, the algorithm first generates a set of state values distributed around the mean `State` value by using the unscented transformation. These generated state values are called sigma points. The algorithm uses each of the sigma points as an input to the state transition and measurement functions to get a new set of transformed state points and measurements. The transformed points are used to compute the state and state estimation error covariance value at the next time step.

The spread of the sigma points around the mean state value is controlled by two parameters `Alpha` and `Kappa`. A third parameter, `Beta`, impacts the weights of the transformed points during state and measurement covariance calculations:

- `Alpha` — Determines the spread of the sigma points around the mean state value. It is usually a small positive value. The spread of sigma points is proportional to `Alpha`. Smaller values correspond to sigma points closer to the mean state.
- `Kappa` — A second scaling parameter that is usually set to 0. Smaller values correspond to sigma points closer to the mean state. The spread is proportional to the square-root of `Kappa`.
- `Beta` — Incorporates prior knowledge of the distribution of the state. For Gaussian distributions, `Beta` = 2 is optimal.

If you know the distribution of state and state covariance, you can adjust these parameters to capture the transformation of higher-order moments of the distribution. The algorithm can track only a single peak in the probability distribution of the state. If there are multiple peaks in the state distribution of your system, you can adjust these parameters so that the sigma points stay around a single peak. For example, choose a small `Alpha` to generate sigma points close to the mean state value.

For more information, see “Unscented Kalman Filter Algorithm”.

`Alpha` is a tunable property. You can change it using dot notation.

`Beta` — Characterization of state distribution

2 (default) | scalar value greater than or equal to 0

Characterization of the state distribution that is used to adjust weights of transformed sigma points, specified as a scalar value greater than or equal to 0. For Gaussian distributions, `Beta` = 2 is an optimal choice.

For more information, see the `Alpha` property description.

`Beta` is a tunable property. You can change it using dot notation.

`HasAdditiveMeasurementNoise` — Measurement noise characteristics

`true` (default) | `false`

Measurement noise characteristics, specified as one of the following values:

- `true` — Measurement noise `v` is additive. The measurement function *h* that is specified in `MeasurementFcn` has the following form:

  ```
  y(k) = h(x(k),Um1,...,Umn)
  ```

  Where `y(k)` and `x(k)` are the estimated output and estimated state at time `k`, and `Um1,...,Umn` are any optional input arguments required by your measurement function.
- `false` — Measurement noise is nonadditive. The measurement function specifies how the output measurement evolves as a function of the state *and* measurement noise:

  ```
  y(k) = h(x(k),v(k),Um1,...,Umn)
  ```

`HasAdditiveMeasurementNoise` is a nontunable property, and you can specify it only during object construction. You cannot change it using dot notation.

`HasAdditiveProcessNoise` — Process noise characteristics

`true` (default) | `false`

Process noise characteristics, specified as one of the following values:

- `true` — Process noise `w` is additive. The state transition function *f* specified in `StateTransitionFcn` has the following form:

  ```
  x(k) = f(x(k-1),Us1,...,Usn)
  ```

  Where `x(k)` is the estimated state at time `k`, and `Us1,...,Usn` are any additional input arguments required by your state transition function.
- `false` — Process noise is nonadditive. The state transition function specifies how the states evolve as a function of the state *and* process noise at the previous time step:

```
x(k) = f(x(k-1),w(k-1),Us1,...,Usn)
```

`HasAdditiveProcessNoise` is a nontunable property, and you can specify it only during object construction. You cannot change it using dot notation.

`Kappa` — Spread of sigma points

`0` (default) | scalar value between 0 and 3

Spread of sigma points around mean state value, specified as a scalar value between 0 and 3 (`0` <= `Kappa` <= `3`). `Kappa` is typically specified as `0`. Smaller values correspond to sigma points closer to the mean state. The spread is proportional to the square-root of `Kappa`. For more information, see the `Alpha` property description.

`Kappa` is a tunable property. You can change it using dot notation.

`MeasurementFcn` — Measurement function

`[ ]` (default) | function handle

Measurement function *h*, specified as a function handle. The function calculates the *N*-element output measurement vector of the nonlinear system at time step *k*, given the state vector at time step *k*. *N* is the number of measurements of the system. You write and save the measurement function and use it to construct the object. For example, if `vdpMeasurementFcn.m` is the measurement function, specify `MeasurementFcn` as `@vdpMeasurementFcn`. You can also specify `MeasurementFcn` as a function handle to an anonymous function.

The inputs to the function depend on whether you specify the measurement noise as additive or nonadditive in the `HasAdditiveMeasurementNoise` property of the object:

- `HasAdditiveMeasurementNoise` is true — The measurement noise `v` is additive, and the measurement function specifies how the measurements evolve as a function of state values:

  ```
  y(k) = h(x(k),Um1,...,Umn)
  ```

  Where `y(k)` and `x(k)` are the estimated output and estimated state at time `k`, and `Um1,...,Umn` are any optional input arguments required by your measurement function. For example, if you are using multiple sensors for tracking an object, an additional input could be the sensor position. During estimation, you pass these additional arguments to the `correct` command which in turn passes them to the measurement function.

- `HasAdditiveMeasurementNoise` is false — The measurement noise is nonadditive, and the measurement function also specifies how the output measurement evolves as a function of the measurement noise:

  ```
  y(k) = h(x(k),v(k),Um1,...,Umn)
  ```

To see an example of a measurement function with additive process noise, type `edit vdpMeasurementFcn` at the command line. To see an example of a measurement function with nonadditive process noise, type `edit vdpMeasurementNonAdditiveNoiseFcn`.

`MeasurementFcn` is a nontunable property. You can specify it once before using the `correct` command either during object construction or using dot notation after object construction. You cannot change it after using the `correct` command.

`MeasurementNoise` — Measurement noise covariance

1 (default) | scalar | matrix

Measurement noise covariance, specified as a scalar or matrix depending on the value of the `HasAdditiveMeasurementNoise` property:

- `HasAdditiveMeasurementNoise` is true — Specify the covariance as a scalar or an *N*-by-*N* matrix, where *N* is the number of measurements of the system. Specify a scalar if there is no cross-correlation between measurement noise terms and all the terms have the same variance. The software uses the scalar value to create an *N*-by-*N* diagonal matrix.
- `HasAdditiveMeasurementNoise` is false — Specify the covariance as a *V*-by-*V* matrix, where *V* is the number of measurement noise terms. `MeasurementNoise` must be specified before using `correct`. After you specify `MeasurementNoise` as a matrix for the first time, to then change `MeasurementNoise` you can also specify it as a scalar. Specify as a scalar if there is no cross-correlation between the measurement noise terms and all the terms have the same variance. The software extends the scalar to a *V*-by-*V* diagonal matrix with the scalar on the diagonals.

`MeasurementNoise` is a tunable property. You can change it using dot notation.

`ProcessNoise` — Process noise covariance

1 (default) | scalar | matrix

Process noise covariance, specified as a scalar or matrix depending on the value of the `HasAdditiveProcessNoise` property:

- `HasAdditiveProcessNoise` is true — Specify the covariance as a scalar or an *M*-by-*M* matrix, where *M* is the number of states of the system. Specify a scalar if there is no cross-correlation between process noise terms, and all the terms have the same variance. The software uses the scalar value to create an *M*-by-*M* diagonal matrix.
- `HasAdditiveProcessNoise` is false — Specify the covariance as a *W*-by-*W* matrix, where *W* is the number of process noise terms. `ProcessNoise` must be specified before using `predict`. After you specify `ProcessNoise` as a matrix for the first time, to then change `ProcessNoise` you can also specify it as a scalar. Specify as a scalar if there is no cross-correlation between the process noise terms and all the terms have the same variance. The software extends the scalar to a *W*-by-*W* diagonal matrix.

`ProcessNoise` is a tunable property. You can change it using dot notation.

`State` — State of nonlinear system

[ ] (default) | vector

State of the nonlinear system, specified as a vector of size *M*, where *M* is the number of states of the system.

When you use the `predict` command, `State` is updated with the predicted value at time step *k* using the state value at time step *k*–1. When you use the `correct` command, `State` is updated with the estimated value at time step *k* using measured data at time step *k*.

The initial value of `State` is the value you specify in the `InitialState` input argument during object creation. If you specify `InitialState` as a column vector, then `State` is also a column vector, and the `predict` and `correct` commands return state estimates as a column vector. Otherwise, a row vector is returned. If you want a filter with single-precision floating-point variables, you must specify `State` as a single-precision variable during object construction using the `InitialState` input argument.

`State` is a tunable property. You can change it using dot notation.

`StateCovariance` — State estimation error covariance

1 (default) | scalar | matrix

State estimation error covariance, specified as a scalar or an *M*-by-*M* matrix, where *M* is the number of states of the system. If you specify a scalar, the software uses the scalar value to create an *M*-by-*M* diagonal matrix.

Specify a high value for the covariance when you do not have confidence in the initial state values that you specify in the `InitialState` input argument.

When you use the `predict` command, `StateCovariance` is updated with the predicted value at time step *k* using the state value at time step *k*–1. When you use the `correct` command, `StateCovariance` is updated with the estimated value at time step *k* using measured data at time step *k*.

`StateCovariance` is a tunable property. You can change it using dot notation after using the `correct` or `predict` commands.

`StateTransitionFcn` — State transition function

[ ] (default) | function handle

State transition function *f*, specified as a function handle. The function calculates the *M*-element state vector of the system at time step *k*, given the state vector at time step *k*-1. *M* is the number of states of the nonlinear system.

You write and save the state transition function for your nonlinear system and use it to construct the object. For example, if `vdpStateFcn.m` is the state transition function, specify `StateTransitionFcn` as `@vdpStateFcn`. You can also specify `StateTransitionFcn` as a function handle to an anonymous function.

The inputs to the function you write depend on whether you specify the process noise as additive or nonadditive in the `HasAdditiveProcessNoise` property of the object:

- `HasAdditiveProcessNoise` is true — The process noise `w` is additive, and the state transition function specifies how the states evolve as a function of state values at previous time step:

  ```
  x(k) = f(x(k-1),Us1,...,Usn)
  ```

  Where `x(k)` is the estimated state at time `k`, and `Us1,...,Usn` are any additional input arguments required by your state transition function, such as system inputs or the sample time. During estimation, you pass these additional arguments to the `predict` command, which in turn passes them to the state transition function.
- `HasAdditiveProcessNoise` is false — The process noise is nonadditive, and the state transition function also specifies how the states evolve as a function of the process noise:

  ```
  x(k) = f(x(k-1),w(k-1),Us1,...,Usn)
  ```

To see an example of a state transition function with additive process noise, type `edit vdpStateFcn` at the command line.

`StateTransitionFcn` is a nontunable property. You can specify it once before using the `predict` command either during object construction or using dot notation after object construction. You cannot change it after using the `predict` command.

Output Arguments

`obj` — unscented Kalman filter object for online state estimation
`unscentedKalmanFilter` object

Unscented Kalman filter object for online state estimation, returned as an `unscentedKalmanFilter` object. This object is created using the specified properties. Use the `correct` and `predict` commands to estimate the state and state estimation error covariance using the unscented Kalman filter algorithm.

When you use `predict`, `obj.State` and `obj.StateCovariance` are updated with the predicted value at time step *k* using the state value at time step *k*–1. When you use `correct`, `obj.State` and `obj.StateCovariance` are updated with the estimated values at time step *k* using measured data at time step *k*.

More About

- "What Is Online Estimation?"
- "Extended and Unscented Kalman Filter Algorithms for Online State Estimation"
- "Validate Online State Estimation at the Command Line"
- "Troubleshoot Online State Estimation"

See Also

Functions
`clone` | `correct` | `extendedKalmanFilter` | `predict`

Blocks
`Kalman Filter`

Introduced in R2016b

wavenet

Create a wavelet network nonlinearity estimator object

Syntax

```
NL = wavenet
NL = wavenet(Name,Value)
```

Description

`NL = wavenet` creates a default wavelet network nonlinearity estimator object for estimating nonlinear ARX and Hammerstein-Wiener models. Use dot notation to customize the object properties, if needed.

`NL = wavenet(Name,Value)` creates a wavelet network nonlinearity estimator object with properties specified by one or more `Name,Value` pair arguments. The properties that you do not specify retain their default value.

Object Description

`wavenet` is an object that stores the wavelet network nonlinearity estimator for estimating nonlinear ARX and Hammerstein-Wiener models.

Use `wavenet` to define a nonlinear function $y = F(x,\theta)$, where y is scalar, x is an `m`-dimensional row vector of regressors, and θ represent the parameters in wavelet expansion. The wavelet network function is based on the following function expansion:

$$
\begin{aligned}
F(x,\theta) = (x-r)PL + a_{s\_1} f\left(b_{s\_1}\left((x-r)Q - c_{s\_1}\right)\right) + \ldots \\
+ a_{s\_ns} f\left(b_{s\_ns}\left((x-r)Q - c_{s\_ns}\right)\right) \\
+ a_{w\_1} g\left(b_{w\_1}\left((x-r)Q - c_{w\_1}\right)\right) + \ldots \\
+ a_{w\_nw} g\left(b_{w\_nw}\left((x-r)Q - c_{w\_nw}\right)\right) + d
\end{aligned}
$$

$$f(z) = e^{-0.5zz^T}$$

$$g(z) = (m - zz^T)e^{-0.5zz^T}$$

Here,

- *f(z)* is a radial function called the scaling function, and z is the input to the scaling function. z is a 1-by-q row vector. q is the number of components of `x` used in the scaling and wavelet functions.
- *g(z)* is a radial function called the wavelet function, and z is the input to the wavelet function.
- θ represents the following parameters of the nonlinearity estimator:
  - P and Q — Projection matrices of size `m`-by-`p` and `m`-by-`q`, respectively.

    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 is `q`. `q` is 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 – Mean value of the regressor vector computed from estimation data, specified as a 1-by-`m` vector.
  - a_s, b_s, a_w, and b_w — Scaling and wavelet parameters, specified as scalars. Parameters with the s subscript are scaling parameters, and parameters with the w subscript are wavelet parameters.
  - L — Specified as a `p`-by-1 vector.
  - c_s and c_w – Specified as a 1-by-`q` vectors.
  - d — Output offset, specified as a scalar.

The value `F(x)` is computed by `evaluate(NL,x)`, where `NL` is the wavenet object.

For `wavenet` object properties, see “Properties” on page 1-1604.

Examples

Create a Default Wavelet Nonlinearity Estimator

```
NL = wavenet;
```

Exclude the linear term from the wavelet expansion.

```
NL.LinearTerm = 'off';
```

Estimate Nonlinear ARX Model with Specific Nonlinearity

Load the estimation data.

```
load twotankdata;
```

Create an `iddata` object from the estimation data.

```
z = iddata(y,u,0.2);
```

Create a wavelet network nonlinearity estimator with 5 units.

```
NL = wavenet('NumberOfUnits',5);
```

Estimate the nonlinear ARX model.

```
sys = nlarx(z,[4 4 1],NL);
```

Estimate MIMO Nonlinear ARX Model

Load the estimation data.

```
load motorizedcamera;
```

Create an `iddata` object.

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

`z` is an `iddata` object with 6 inputs and 2 outputs.

Specify the model orders.

```
Orders = [ones(2,2),2*ones(2,6),ones(2,6)];
```

Specify different nonlinearity estimators for each output channel.

```
NL = [wavenet('NumberOfUnits',2),linear];
```

Estimate the nonlinear ARX model.

```
sys = nlarx(z,Orders,NL);
```

Estimate MIMO Hammerstein-Wiener Model

Load the estimation data.

```
load motorizedcamera;
```

Create an `iddata` object.

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

`z` is an `iddata` object with 6 inputs and 2 outputs.

Specify the model orders and delays.

```
Orders = [ones(2,6),ones(2,6),ones(2,6)];
```

Specify the same nonlinearity estimator for each input channel.

```
InputNL = saturation;
```

Specify different nonlinearity estimators for each output channel.

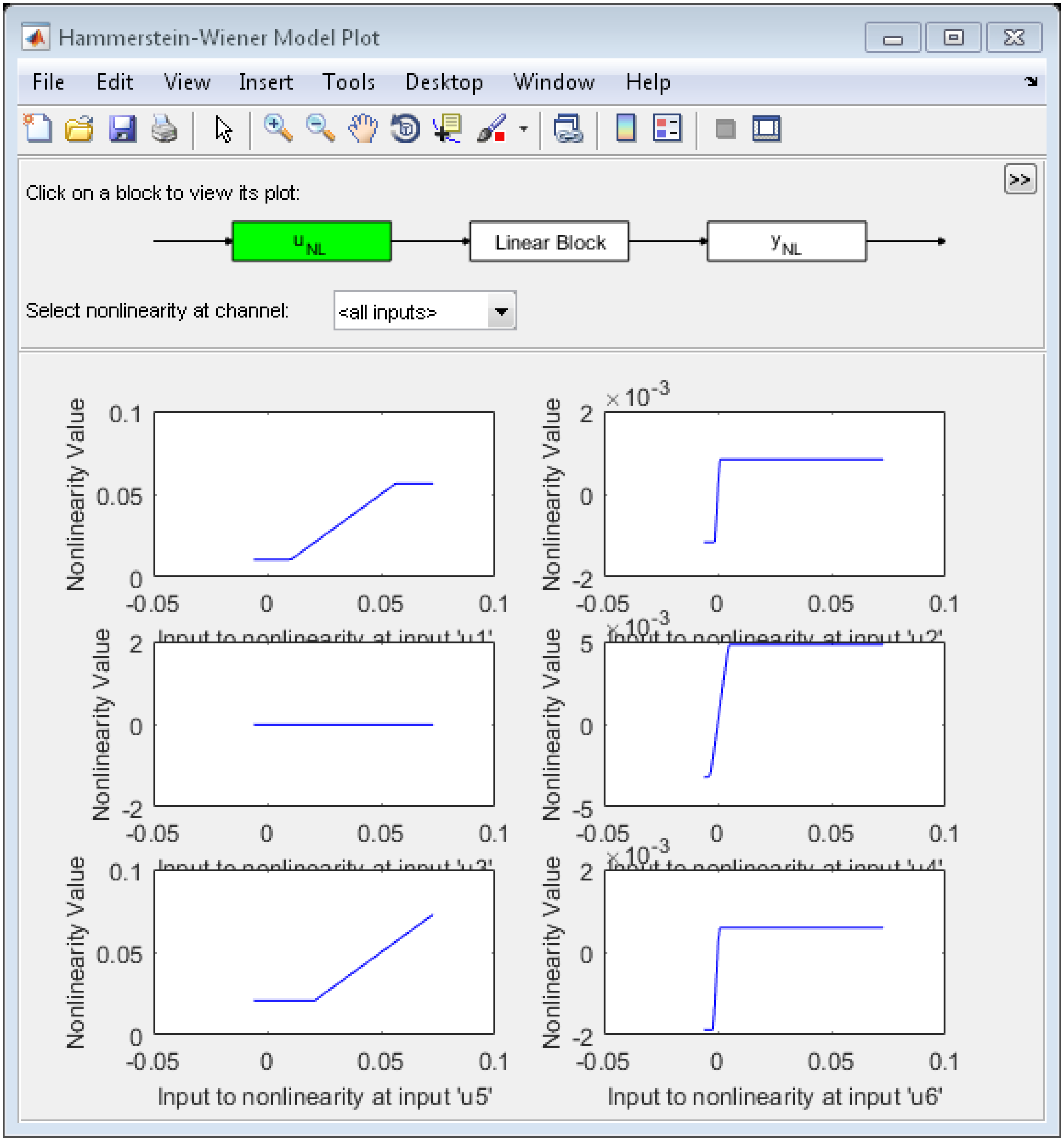
```
 OutputNL = [deadzone,wavenet];
```

Estimate the Hammerstein-Wiener model.

```
sys = nlhw(z,Orders,InputNL,OutputNL);
```

To see the shape of the estimated input and output nonlinearities, plot the nonlinearities.

```
plot(sys)
```


Hammerstein-Wiener Model Plot
File
Edit
View
Insert
Tools
Desktop
Window
Help
>>
Click on a block to view its plot:
u_NL
Linear Block
y_NL
Select nonlinearity at channel:
<all inputs>
Nonlinearity Value
Input to nonlinearity at input 'u1'
Input to nonlinearity at input 'u2'
Input to nonlinearity at input 'u3'
Input to nonlinearity at input 'u4'
Input to nonlinearity at input 'u5'
Input to nonlinearity at input 'u6'
× 10^-3
0.1
0.05
0
2
-2
5
-5
-0.05
0
0.05
0.1

Click on the input and output nonlinearity blocks on the top of the plot to see the nonlinearites.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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 `wavenet` nonlinearity. For example, `NL= wavenet('NumberofUnits',5)` creates a wavelet nonlinearity estimator object with five nonlinearity units in wavelet expansion.

Properties

`NumberofUnits`

Number of nonlinearity units in wavelet expansion, specified as a positive integer or one of the following values:

- `'Auto'` — The number of units are automatically determined from estimation data.
- `'Interactive'` — The number of units are determined during model estimation.

Default: `'Auto'`

`LinearTerm`

Inclusion of linear term, specified as one of the following values:

- `'on'` — The linear term, $(x-r)PL$, is included in the wavelet expansion.
- `'off'` — The linear term in not included in the wavelet expansion.

Default: `'on'`

`Parameters`

Parameters in wavelet expansion, specified as a structure with the following fields:

Field Name	Description	Default
`RegressorMe`	Means of the regressors, r, specified as a 1-by-m vector. m is the number of regressors, x. For Hammerstein-Wiener models, $m = 1$.	`[]`
`NonLinearSu`	Projection matrix, Q, specified as an m-by-q matrix.	`[]`
`LinearSubsp`	Projection matrix, P, specified as an m-by-p matrix.	`[]`
`LinearCoef`	Linear coefficients, L, specified as a p-by-1 vector.	`[]`
`ScalingDila`	Scaling function dilation, b_{s_ns}, specified as an ns-by-1 matrix.	`[]`
`WaveletDila`	Wavelet function dilation, b_{w_nw}, specified as an nw-by-1 matrix.	`[]`
`ScalingTran`	Scaling function translation, c_{s_ns}, specified as an ns-by-q matrix.	`[]`
`WaveletTran`	Wavelet function translation, c_{w_nw}, specified as an nw-by-q matrix.	`[]`
`ScalingCoef`	Scaling function coefficients, a_{s_ns}, specified as an ns-by-1 vector.	`[]`
`WaveletCoef`	Wavelet function coefficients, a_{w_nw}, specified as an nw-by-1 vector.	`[]`
`OutputOffse`	Output offset, `d`, specified as a scalar.	`[]`

The parameters are typically not assigned directly. They are estimated by the identification algorithm (`nlarx` or `nlhw`) when `wavenet` is used in a Nonlinear ARX (`idnlarx`) or Hammerstein-Wiener (`idnlhw`) model.

`Options`

Options specifying the initial wavelet nonlinearity structure, specified as a structure with the following fields:

Field Name	Description	Default
`FinestCell`	Minimum number of data points in the smallest cell, specified as an integer or character vector. A cell is the area covered by the significantly nonzero portion of a wavelet.	`'auto'` — Compute the value from the data.

Field Name	Description	Default
`MinCells`	Minimum number of cells in the partition, specified as an integer.	`16`
`MaxCells`	Maximum number of cells in the partition, specified as an integer.	`128`
`MaxLevels`	Maximum number of wavelet levels, specified as an integer.	`10`
`DilationSte`	Dilation step size, specified as real scalar.	`2`
`Translation`	Translation step size, specified as real scalar.	`1`

Output Arguments

`NL` — Wavelet nonlinearity estimator object

`wavenet` object

Wavelet nonlinearity estimator object, returned as a `wavenet` object.

More About

Algorithms

`wavenet` can be used in both Nonlinear ARX and Hammerstein-Wiener models.

- When used in a Nonlinear ARX model:
  - If the `Focus` estimation option (see, `nlarxOptions`) is `'prediction'`, `wavenet` uses a fast, noniterative technique for estimating parameters [1]. Successive refinements after the first estimation use an iterative algorithm.
  - If `Focus` is `'simulation'`, `wavenet` uses an iterative technique for estimating parameters.

  To always use a noniterative or iterative algorithm, specify the `IterWavenet` option of `nlarxOptions`.
- When used in a Hammerstein-Wiener model, `wavenet` parameters are determined by iterative minimization.

References

[1] Zhang, Q. “Using wavelet network in nonparametric estimation.” *IEEE Trans. on Neural Networks*, Vol. 8, Number 2, March 1997, pp. 227-236.

See Also

`nlarx` | `nlhw`

Introduced in R2007a

xperm

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`

Introduced in R2008b

zero

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 Zero Locations and Gain of Transfer Function

Create the following tranfer 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]);
```

Calculate the zero locations and overall gain of the transfer function.

```
[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.

```
H = chgTimeUnit(H,'minutes');
```

`zero` returns locations relative to the new unit.

```
[z,gain] = zero(H)
```

```
z =

   -4.3581
    0.7867


gain =

    4.2000
```

See Also

`pole` | `pzmap` | `tzero`

Introduced in R2012a

zgrid

Generate z-plane grid of constant damping factors and natural frequencies

Syntax

```
zgrid
zgrid(z,wn)
zgrid([],[])
```

Description

`zgrid` 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 π in steps of π/10, and plots the grid over the current axis. If the current axis contains a discrete *z*-plane root locus diagram or pole-zero map, `zgrid` draws the grid over the plot without altering the current axis limits.

`zgrid(z,wn)` plots a grid of constant damping factor and natural frequency lines for the damping factors and normalized natural frequencies in the vectors `z` and `wn`, respectively. If the current axis contains a discrete *z*-plane root locus diagram or pole-zero map, `zgrid(z,wn)` draws the grid over the plot. The frequency lines for unnormalized (true) frequencies can be plotted using

```
zgrid(z,wn/Ts)
```

where `Ts` is the sample time.

`zgrid([],[])` draws the unit circle.

Alternatively, you can select **Grid** from the right-click menu to generate the same z-plane grid.

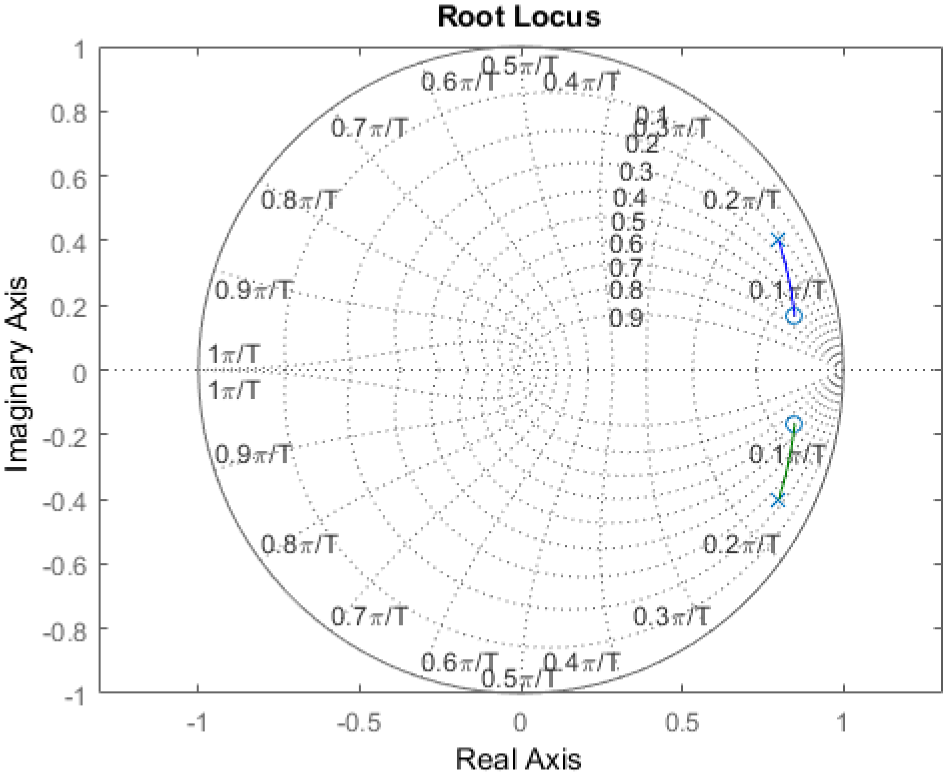
Examples

Plot z-plane grid lines on the root locus

To see the z-plane grid on the root locus plot, type

```
H = tf([2 -3.4 1.5],[1 -1.6 0.8],-1)
rlocus(H)
zgrid
axis equal
```

```
H =

  2 z^2 - 3.4 z + 1.5
  -------------------
   z^2 - 1.6 z + 0.8

Sample time: unspecified
Discrete-time transfer function.
```



See Also

`pzmap` | `rlocus` | `sgrid`

Introduced in R2012a

zpkdata

Access zero-pole-gain data

Syntax

```
[z,p,k] = zpkdata(sys)
[z,p,k,Ts] = zpkdata(sys)
[z,p,k,Ts,covz,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] = zpkdata(sys)` also returns the sample time `Ts`.

`[z,p,k,Ts,covz,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-Nz, where `Nz` 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)

Sample 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`

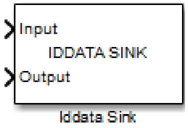
Introduced before R2006a

2

Blocks — Alphabetical List

IDDATA Sink

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.

Parameters

IDDATA Name

Name of the `iddata` object in the MATLAB workspace.

Sample Time (sec.)

Time interval (in seconds) between successive data samples.

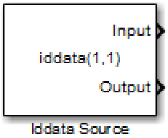
See Also

`IDDATA Source`

Introduced in R2008a

IDDATA Source

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.

Parameters

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`.

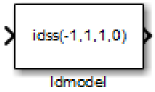
See Also

`IDDATA Sink`

Introduced in R2008a

IDMODEL Model

Simulate identified linear model 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.

Parameters

Identified model

Name of an estimated linear model in the MATLAB workspace. The model must be an `idss`, `idgrey`, `idpoly`, `idtf`, or `idproc` object.

The block supports continuous-time or discrete-time models with or without input-output delays.

Initial states (state space only: idss, idgrey)

Initial states for state-space (`idss`) and grey-box (`idgrey`) models. Initial states must be a vector of length equal to the order of the model.

For models other than `idss` and `idgrey`, initial conditions are zero.

In some situations, you may want to reproduce the results in the Model Output plot window in the System Identification app or those of the `compare` plot. To do so:

1. Convert the identified model into state-space form (`idss` model), and use the state-space model in the block.
2. Compute the initial state values that produce the best fit between the model output and the measured output signal using `findstates`.
3. Specify the same input signal for simulation that you used as the validation data in the app or in the `compare` plot.

   For example:

   ```
   % Convert to state-space model
   mss = idss(m);
   % Estimate initial states from data
   X0 = findstates(mss,z);
   ```

   `z` is the data set you used for validating the model `m`. Use the model `mss` and initial states `X0` in the Idmodel block to perform the simulation.

Add noise

Select to add noise. When selected, Simulink derives the noise amplitude from 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 is infinite. In reality, you see a band-limited noise that takes into account the time constants of the system. You can interpret the resulting simulated output as filtered using a lowpass filter with a passband that does not distort the dynamics from the input.

Noise seed(s)

Seed, specified as an integer, that forces the simulation to add the same noise to the output every time you simulate the model. Applies only when you select the **Add noise** check box. For more information about using seeds, see `rand` in the MATLAB documentation.

For multi-output models, you can use independent noise realizations that generate the outputs with additive noise. Enter a vector of $\mathtt{N_y}$ entries, where $\mathtt{N_y}$ is the number of output channels.

For random restarts that vary from one simulation to another, leave the field empty.

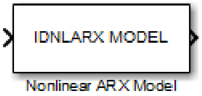
See Also

`findstates` | `idpoly` | `idproc` | `idss` | `idtf` | `sim`

Introduced in R2008a

IDNLARX Model

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.

For multi-input models, specify the input as an `Nu`-element vector, where `Nu` is the number of inputs. For example, you can use a `Vector Concatenate` block to concatenate scalar signals into a vector signal.

Note: Do not use a `Bus Creator` or `Mux` block to produce the vector signal.

Output

Simulated output from the model.

Parameters

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` 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` command.

    For example, to compute initial states such that the response of the model `M` matches the output data in the data set `z`, you can enter `X0`, such that:

    ```
    X0 = findstates(M,z,[],'sim')
    ```
  - To continue a simulation from a previous run, use the simulated input-output values from the previous simulation to compute the initial states `X0` for the current simulation.

For example, suppose that `firstSimData` is a variable that stores the input and output values from a previous simulation. For a model `M`, you can enter `X0`, such that:

```
X0 = data2state(M,firstSimData)
```

Examples

Simulate Nonlinear ARX Model in Simulink®

Load the sample data.

```
load twotankdata
```

Create a data object from sample data.

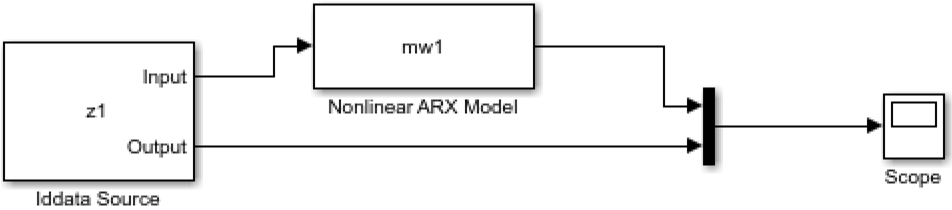
```
z = iddata(y,u,0.2,'Tstart',0,'Name','Two tank system');
z1 = z(1:1000);
```

Estimate a nonlinear ARX model.

```
mw1 = nlarx(z1,[5 1 3],wavenet('NumberOfUnits',8));
```

Open a preconfigured Simulink model.

```
model = fullfile(matlabroot,'examples','ident','ex_idnlarx_block');
open_system(model);
```



The model uses the Iddata Source, Nonlinear ARX Model, and Scope blocks. The following block parameters have been preconfigured to specify the estimation data, estimated model, and input and output levels:

1. Block parameters of Iddata Source block:

- **IDDATA Object** - `z1`

2. Block parameters of Nonlinear ARX Model block:

- **Model** - `mw1`
- **Initial conditions** - Input and output values (default)
- **Input level** - 10
- **Output level** - 0.1

Run the simulation.

Click the Scope block to view the difference between measured output and model output.

To reduce the difference between the measured and simulated responses, estimate an initial state vector for the model from the estimation data, `z1`.

```
x0 = findstates(mw1,z1,[],'simulation');
```

Set the **Initial Conditions** block parameter value of the Nonlinear ARX Model block to `State Values`.

Specify initial states as `x0`.

Run the simulation, and view the difference between measured output and model output.

```
bdclose(model)
```

See Also

Related Commands

`idnlarx/findop`

`findstates`

`idnlarx`

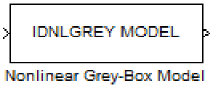
Topics in the System Identification Toolbox User's Guide

"Identifying Nonlinear ARX Models"

Introduced in R2008a

IDNLGREY Model

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.

Parameters

IDNLGREY model

Name of `idnlgrey` variable in the MATLAB workspace.

Initial state

Specify the initial states as one of the following:

- `'z'`: Specifies zero, which corresponds to a system starting from rest.
- `'m'`: Specifies the internal initial states of the model.

- Vector of size equal to the number of states in the `idnlgrey` object.
- An initial state structure array. For information about creating this structure, type `help idnlgrey/sim` in the MATLAB Command Window.

See Also

Related Commands

`idnlgrey`

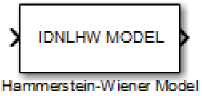
Topics in the System Identification Toolbox User's Guide

"Estimate Nonlinear Grey-Box Models"

Introduced in R2008a

IDNLHW Model

Simulate Hammerstein-Wiener model in Simulink software



Library

System Identification Toolbox

Description

The IDNLHW Model block simulates a Hammerstein-Wiener (`idnlhw`) model for time-domain input and output data.

Input

Input signal to the model.

For multi-input models, specify the input as an `Nu`-element vector, where `Nu` is the number of inputs. For example, you can use a `Vector Concatenate` block to concatenate scalar signals into a vector signal.

Note: Do not use a `Bus Creator` or `Mux` block to produce the vector signal.

Output

Simulated output from the model.

Parameters

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` 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`.

    For example, for the data set `z` and model `m`, you can enter `X0`, such that:

    ```
    X0 = findstates(m,z)
    ```

Examples

Example 1

In this example, you estimate a Hammerstein-Wiener model from data and compare the model output of the model to the measured output of the system.

1. Load the sample data.

   ```
   load twotankdata
   ```
2. Create a data object from sample data.

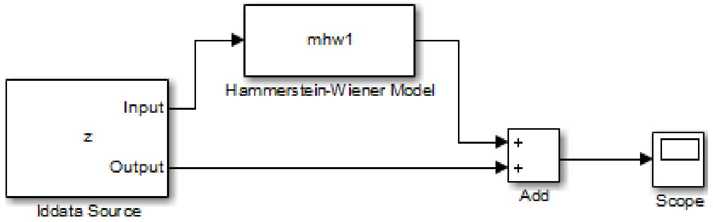
   ```
   z = iddata(y,u,0.2, ...
             'Name','Two tank system',...
   ```

```
              'Tstart',0);
```

3 Estimate a Hammerstein-Wiener model.

```
mhw1 = nlhw(z,[1 5 3],pwlinear,pwlinear);
```

4 Build the following Simulink model using the IDDATA Source, IDNLHW Model, and Scope blocks.



5 Double-click the IDDATA Source block and enter the following into the block parameter dialog box:

- **IDDATA Object**: z

Click **OK**.

6 Double-click the IDNLHW Model block and enter the following into the block parameter dialog box:

- **Model**: mhw1
- **Initial Conditions**: Zero

7 Run the simulation.

Click the Scope block to view the difference between the measured output and the model output. Use the **Autoscale** toolbar button to scale the axes.

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

`idnlhw/findop`

`findstates`

`idnlhw`

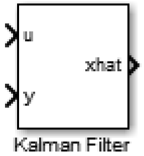
Topics in the System Identification Toolbox User's Guide

"Identifying Hammerstein-Wiener Models"

Introduced in R2008a

Kalman Filter

Estimate states of discrete-time or continuous-time linear system



Library

Estimators

Description

Use the Kalman Filter block to estimate states of a state-space plant model given process and measurement noise covariance data. The state-space model can be time-varying. A steady-state Kalman filter implementation is used if the state-space model and the noise covariance matrices are all time-invariant. A time-varying Kalman filter is used otherwise.

Kalman filter provides the optimal solution to the following continuous or discrete estimation problems:

Continuous-Time Estimation

Given the continuous plant

$$\begin{aligned}\dot{x}(t) &= A(t)x(t) + B(t)u(t) + G(t)w(t) \quad \text{(state equation)}\\ y(t) &= C(t)x(t) + D(t)u(t) + H(t)w(t) + v(t) \quad \text{(measurement equation)}\end{aligned}$$

with known inputs u, white process noise w, and white measurement noise v satisfying:

$$
\begin{aligned}
&E\left[w(t)\right] = E\left[v(t)\right] = 0\\
&E[w(t)w^T(t)] = Q(t)\\
&E[w(t)v^T(t)] = N(t)\\
&E[v(t)v(t)^T] = R(t)
\end{aligned}
$$

construct a state estimate $\hat{x}$ that minimizes the state estimation error covariance $P(t) = E[(x-x)(x-x)^T]$.

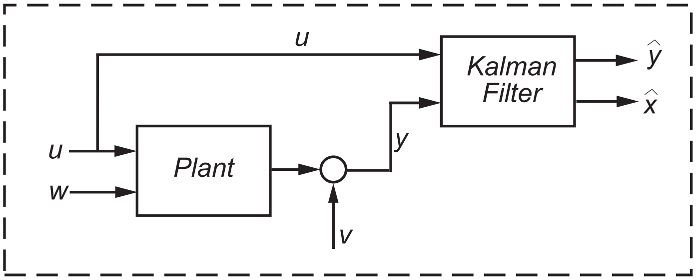
The optimal solution is the Kalman filter with equations

$$
\begin{aligned}
&L(t) = (P(t)C^T(t) + \bar{N}),\\
&\dot{P}(t) = A(t)P(t) + P(t)A^T(t) + \bar{Q}(t) - L(t)\bar{R}(t)L^T(t),\\
&\dot{x}(t) = A(t)x(t) + B(t)u(t) + L(t)(y(t) - C(t)x(t) - D(t)u(t)),
\end{aligned}
$$

where

$$
\begin{aligned}
&\bar{Q}(t) = G(t)Q(t)G^T(t),\\
&\bar{R}(t) = R(t) + H(t)N(t) + N^T(t)H^T(t) + H(t)Q(t)H^T(t),\\
&\bar{N}(t) = G(t)(Q(t)H^T(t) + N(t)).
\end{aligned}
$$

The Kalman filter uses the known inputs u and the measurements y to generate the state estimates $\hat{x}$. If you want, the block can also output the estimates of the true plant output $\hat{y}$.



Kalman Estimator

The block implements the steady-state Kalman filter when the system matrices (`A(t)`, `B(t)`, `C(t)`, `D(t)`, `G(t)`, `H(t)`) and noise covariance matrices (`Q(t)`, `R(t)`, `N(t)`) are constant (specified in the Block Parameters dialog box). The steady-state Kalman filter uses a constant matrix `P` that minimizes the steady-state estimation error covariance and solves the associated continuous-time algebraic Riccati equation:

$$P = \lim_{t \to \infty} E[(x - x)(x - x)^T].$$

Discrete-Time Estimation

Given the discrete plant

$$\begin{aligned} x[n+1] &= A[n]\, x[n] + B[n]\, u[n] + G[n]\, w[n], \\ y[n] &= C[n]\, x[n] + D[n]\, u[n] + H[n]\, w[n] + v[n], \end{aligned}$$

with known inputs u, white process noise w and white measurement noise v satisfying

$$\begin{aligned} &E[w[n]] = E[v[n]] = 0, \\ &E[w[n]w^T[n]] = Q[n], \\ &E[v[n]v^T[n]] = R[n], \\ &E[w[n]v^T[n]] = N[n]. \end{aligned}$$

The estimator has the following state equation

$$\hat{x}[n+1 \mid n] = A[n]\hat{x}[n \mid n-1] + B[n]u[n] + L[n](y[n] - C[n]\hat{x}[n \mid n-1] - D[n]u[n]),$$

where the gain `L[n]` is calculated through the discrete Riccati equation:

$$\begin{aligned} &L[n] = (A[n]P[n]C^T[n] + \bar{N}[n])(C[n]P[n]C^T[n] + \bar{R}[n])^{-1}, \\ &M[n] = P[n]C^T[n](C[n]P[n]C^T[n] + \bar{R}[n])^{-1}, \\ &Z[n] = (I - M[n]C[n])P[n](I - M[n]C[n])^T + M[n]\bar{R}[n]M^T[n], \\ &P[n+1] = (A[n] - \bar{N}[n]\bar{R}^{-1}[n]C[n])Z(A[n] - \bar{N}[n]\bar{R}^{-1}[n]C[n])^T + \bar{Q}[n] - N[n]\bar{R}^{-1}[n]N^T[n], \end{aligned}$$

where I is the identity matrix of appropriate size and

$$
\begin{aligned}
&\bar{Q}[\mathrm{n}] = G[\mathrm{n}]Q[\mathrm{n}]G^T[\mathrm{n}],\\
&\bar{R}[\mathrm{n}] = R[\mathrm{n}] + H[\mathrm{n}]N[\mathrm{n}] + N^T[\mathrm{n}]H^T[\mathrm{n}] + H[\mathrm{n}]Q[\mathrm{n}]H^T[\mathrm{n}],\\
&\bar{N}[\mathrm{n}] = G[\mathrm{n}](Q[\mathrm{n}]H^T[\mathrm{n}] + N[\mathrm{n}]),\\
&\text{and}\\
&P[\mathrm{n}] = E[(x - x[n \mid n-1])(x - \hat{x}[n \mid n-1])^T],\\
&Z[\mathrm{n}] = E[(x - \hat{x}[n \mid n])(x - \hat{x}[n \mid n])^T],
\end{aligned}
$$

The steady-state Kalman filter uses a constant matrix `P` that minimizes the steady-state estimation error covariance and solves the associated discrete-time algebraic Riccati equation.

There are two variants of discrete-time Kalman filters:

- The current estimator generates the state estimates $\hat{x}[n \mid n]$ using all measurement available, including $y[n]$. The filter updates $\hat{x}[n \mid n-1]$ with $y[n]$ and outputs:

  $$
  \begin{aligned}
  &\hat{x}[n \mid n] = \hat{x}[n \mid n-1] + M[n](y[n] - C[n]\hat{x}[n \mid n-1] - D[n]u[n]),\\
  &\hat{y}[n \mid n] = C[n]\hat{x}[n \mid n] + D[n]u[n].
  \end{aligned}
  $$

- The delayed estimator generates the state estimates $\hat{x}[n \mid n-1]$ using measurements up to $y[n-1]$. The filter outputs $\hat{x}[n \mid n-1]$ as defined previously, along with the optional output $\hat{y}[n \mid n-1]$

  $$\hat{y}[n \mid n-1] = C[n]\hat{x}[n \mid n-1] + D[n]u[n]$$

The current estimator has better estimation accuracy compared to the delayed estimator, which is important for slow sample times. However, it has higher computational cost, making it harder to implement inside control loops. More specifically, it has direct feedthrough. This leads to an algebraic loop if the Kalman filter is used in a feedback loop that does not contain any delays (the feedback loop itself also has direct feedthrough). The algebraic loop can impact the speed of simulation. You cannot generate code if your model contains algebraic loops.

The Kalman Filter block differs from the `kalman` command in the following ways:

- When calling `kalman(sys,...)`, `sys` includes the `G` and `H` matrices. Specifically, `sys.B` has `[B G]` and `sys.D` has `[D H]`. When you provide a LTI variable to the Kalman Filter block, it does not assume that the LTI variable provided contains `G` and `H`. They are optional and separate.
- The `kalman` command outputs `[yhat;xhat]` by default. The block only outputs `xhat` by default.

Parameters

The following table summarizes the Kalman Filter block parameters, accessible via the Block Parameter dialog box.

Task	Parameters
Specify filter settings	• **Time domain** • **Use the current measurement y[n] to improve xhat[n]**
Specify the system model	**Model source** in **Model Parameters** tab
Specify initial state estimates	**Source** in **Model Parameters** tab
Specify noise characteristics	In **Model Parameters** tab: • **Use G and H matrices (default G=I and H=0)** • **Q**, **Time-invariant Q** • **R**, **Time-invariant R** • **N**, **Time-invariant N**
Specify additional inports	In **Options** tab: • **Add input port u** • **Add input port Enable to control measurement updates** • **External reset**
Specify additional outports	In **Options** tab: • **Output estimated model output y**

Task	Parameters
	- **Output state estimation error covariance Z**

Time domain

Specify whether to estimate continuous-time or discrete-time states:

- `Discrete-Time` (**Default**) — Block estimates discrete-time states
- `Continuous-Time` — Block estimates continuous-time states

  When the Kalman Filter block is in a model with synchronous state control (see the `State Control` block), you cannot select `Continuous-time`.

Use the current measurement y[n] to improve xhat[n]

Use the current estimator variant of the discrete-time Kalman filter. When not selected, the delayed estimator (variant) is used.

This option is available only when **Time Domain** is `Discrete-Time`.

Model source

Specify how the A, B, C, D matrices are provided to the block. Must be one of the following:

- `Dialog: LTI State-Space Variable` — Use the values specified in the LTI state-space variable. You must also specify the variable name in **Variable**. The sample time of the model must match the setting in the **Time domain** option, i.e. the model must be discrete-time if the **Time domain** is discrete-time.
- `Dialog: Individual A, B, C, D matrices` — Specify values in the following block parameters:
  - **A** — Specify the A matrix. It must be real and square.
  - **B** — Specify the B matrix. It must be real and have as many rows as the A matrix. This option is available only when **Add input port u** is selected in the **Options** tab.
  - **C** — Specify the C matrix. It must be real and have as many columns as the A matrix.

  - **D** — Specify the D matrix. It must be real. It must have as many rows as the C matrix and as many columns as the B matrix. This option is available only when **Add input port u** is selected in the **Options** tab.
- `External` — Specify the A, B, C, D matrices as input signals to the Kalman Filter block. If you select this option, the block includes additional input ports A, B, C and D. You must also specify the following in the block parameters:
  - `Number of states` — Number of states to be estimated, specified as a positive integer. The default value is 2.
  - `Number of inputs` — Number of known inputs in the model, specified as a positive integer. The default value is 2. This option is only available when **Add input port u** is selected.
  - `Number of outputs` — Number of measured outputs in the model, specified as a positive integer. The default value is 2.

Sample Time

Block sample time, specified as -1 or a positive scalar.

This option is available only when **Time Domain** is `Discrete Time` and **Model Source** is `Dialog: Individual A, B, C, D matrices` or `External`. The sample time is obtained from the LTI state-space variable if the Model Source is `Dialog: LTI State-Space Variable`.

The default value is -1, which implies that the block inherits its sample time based on the context of the block within the model. All block input ports must have the same sample time.

Source

Specify how to enter the initial state estimates and initial state estimation error covariance:

- `Dialog` — Specify the values directly in the dialog box. You must also specify the following parameters:
  - **Initial states x[0]** — Specify the initial state estimate as a real scalar or vector. If you specify a scalar, all initial state estimates are set to this scalar. If you specify a vector, the length of the vector must match with the number of states in the model.

- **State estimation error covariance P[0]** (only when time-varying Kalman filter is used) — Specify the initial state estimation error covariance P[0] for discrete-time Kalman filter or P(0) for continuous-time Kalman filter. Must be specified as one of the following:

  - Real nonnegative scalar. P is an Ns-by-Ns diagonal matrix with the scalar on the diagonals. Ns is the number of states in the model.
  - Vector of real nonnegative scalars. P is an Ns-by-Ns diagonal matrix with the elements of the vector on the diagonals of P.
  - Ns-by-Ns positive semi-definite matrix.
- `External` — Inherit the values from input ports. The block includes an additional input port X0. A second additional input port P0 is added when time-varying Kalman filter is used. X0 and P0 must satisfy the same conditions described previously when you specify them in the dialog box.

Use the Kalman Gain K from the model variable

Specify whether to use the pre-identified Kalman Gain contained in the state-space plant model. This option is available only when:

- **Model Source** is `Dialog: LTI State-Space Variable` and **Variable** is an identified state-space model (`idss`) with a nonzero K matrix.
- **Time Invariant Q**, **Time Invariant R** and **Time Invariant N** options are selected.

If the **Use G and H matrices (default G=I and H=0)** option is selected, **Time Invariant G** and **Time Invariant H** options must also be selected.

Use G and H matrices (default G=I and H=0)

Specify whether to use non-default values for the G and H matrices. If you select this option, you must specify:

- **G** — Specify the G matrix. It must be a real matrix with as many rows as the A matrix. The default value is 1.
- **Time-invariant G** — Specify if the G matrix is time invariant. If you unselect this option, the block includes an additional input port G.
- **H** — Specify the H matrix. It must be a real matrix with as many rows as the C matrix and as many columns as the G matrix. The default value is 0.

- **Time-invariant H** — Specify if the H matrix is time invariant. If you unselect this option, the block includes an additional input port G.
- **Number of process noise inputs** — Specify the number of process noise inputs in the model. The default value is 1.

  This option is available only when **Time-invariant G** and **Time-invariant H** are unselected. Otherwise, this information is inferred from the G or H matrix.

Q

Process noise covariance matrix, specified as one of the following:

- Real nonnegative scalar. Q is an Nw-by-Nw diagonal matrix with the scalar on the diagonals. Nw is the number of process noise inputs in the model.
- Vector of real nonnegative scalars. Q is an Nw-by-Nw diagonal matrix with the elements of the vector on the diagonals of Q.
- Nw-by-Nw positive semi-definite matrix.

Time Invariant Q

Specify if the Q matrix is time invariant. If you unselect this option, the block includes an additional input port Q.

R

Measurement noise covariance matrix, specified as one of the following:

- Real positive scalar. R is an Ny-by-Ny diagonal matrix with the scalar on the diagonals. Ny is the number of measured outputs in the model.
- Vector of real positive scalars. R is an Ny-by-Ny diagonal matrix with the elements of the vector on the diagonals of R.
- Ny-by-Ny positive-definite matrix.

Time Invariant R

Specify if the R matrix is time invariant. If you unselect this option, the block includes an additional input port R.

N

Process and measurement noise cross-covariance matrix. Specify it as a Nw-by-Ny matrix. The matrix [Q N; N^T R] must be positive definite.

Time Invariant N

Specify if the N matrix is time invariant. If you unselect this option, the block includes an additional input port N.

Add input port u

Select this option if your model contains known inputs `u(t)` or `u[k]`. The option is selected by default. Unselecting this option removes the input port `u` from the block and removes the **B**, **D** and **Number of inputs** parameters from the block dialog box.

Add input port Enable to control measurement updates

Select this option if you want to control the measurement updates. The block includes an additional inport `Enable`. The `Enable` input port takes a scalar signal. This option is unselected by default.

By default the block does measurement updates at each time step to improve the state and output estimates $\hat{x}$ and $\hat{y}$ based on measured outputs. The measurement update is skipped for the current sample time when the signal in the Enable port is 0. Concretely, the equation for state estimates become $\dot{\hat{x}}(t) = A(t)\hat{x}(t) + B(t)u(t)$ for continuous-time Kalman filter and $\hat{x}[n+1 \mid n] = A[n]\hat{x}[n \mid n-1] + B[n]u[n]$ for discrete-time.

External Reset

Option to reset estimated states and parameter covariance matrix using specified initial values.

Suppose you reset the block at a time step, `t`. If the block is enabled at `t`, the software uses the initial parameter values specified either in the block dialog or the input ports P0 and X0 to estimate the states. In other words, at `t`, the block performs a time update and if it is enabled, a measurement update after the reset. The block outputs these updated estimates.

Specify one of the following:

- `None` (Default) — Estimated states $\hat{x}$ and state estimation error covariance matrix P values are not reset.
- `Rising` — Triggers a reset when the control signal rises from a negative or zero value to a positive value. If the initial value is negative, rising to zero triggers a reset.
- `Falling` — Triggers a reset when the control signal falls from a positive or a zero value to a negative value. If the initial value is positive, falling to zero triggers a reset.
- `Either` — Triggers a reset when the control signal is either rising or falling.
- `Level` — Triggers a reset in either of these cases:
  - The control signal is nonzero at the current time step.
  - The control signal changes from nonzero at the previous time step to zero at the current time step.
- `Level hold` — Triggers reset when the control signal is nonzero at the current time step.

When you choose an option other than `None`, a `Reset` input port is added to the block to provide the reset control input signal.

Output estimated model output y

Add $\hat{y}$ output port to the block to output the estimated model outputs. The option is unselected by default.

Output estimated model output P or Z

Add P output port or Z output port to the block. The Z matrix is provided only when **Time Domain** is `Discrete Time` and the **Use the current measurement y[n] to improve xhat[n]** is selected. Otherwise, the P matrix, as described in the “Description” on page 2-20 section previously, is provided.

The option is unselected by default.

Ports

Port Name	Port Type (In/ Out)	Description
u (Optional)	In	Known inputs, specified as a real scalar or vector.
y	In	Measured outputs, specified as a real scalar or vector.
xhat	Out	Estimated states, returned as a real scalar or vector.
yhat (Optional)	Out	Estimated outputs, returned as a real scalar or vector.
P or Z (Optional)	Out	State estimation error covariance, returned as a matrix.
A (Optional)	In	A matrix, specified as a real matrix.
B (Optional)	In	B matrix, specified as a real matrix.
C (Optional)	In	C matrix, specified as a real matrix.
D (Optional)	In	D matrix, specified as a real matrix.
G (Optional)	In	G matrix, specified as a real matrix.
H (Optional)	In	H matrix, specified as a real matrix.
Q (Optional)	In	Q matrix, specified as a real scalar, vector or matrix.
R (Optional)	In	R matrix, specified as a real scalar, vector or matrix.
N (Optional)	In	N matrix, specified as a real matrix.
P0 (Optional)	In	P matrix at initial time, specified as a real scalar, vector, or matrix.
X0 (Optional)	In	Initial state estimates, specified as a real scalar or vector.
Enable (Optional)	In	Control signal to enable measurement updates, specified as a real scalar.
Reset (Optional)	In	Control signal to reset state estimates, specified as a real scalar.

Supported Data Types

- Double-precision floating point
- Single-precision floating point (for discrete-time Kalman filter only)

Note:

- All input ports except **Enable** and **Reset** must have the same data type (single or double).

- **Enable** and **Reset** ports support `single`, `double`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, and boolean data types.

Limitations

- The plant and noise data must satisfy:
  - (C,A) detectable
  - $\bar{R} > 0$ and $\bar{Q} - \bar{N}\bar{R}^{-1}\bar{N}^T \geq 0$
  - $(A - \bar{N}\bar{R}^{-1}C, \bar{Q} - \bar{N}\bar{R}^{-1}\bar{N}^T)$ has no uncontrollable mode on the imaginary axis (or unit circle in discrete time) with the notation

$$\bar{Q} = GQG^T$$
$$\bar{R} = R + HN + N^T H^T + HQH^T$$
$$\bar{N} = G(QH^T + N)$$

- The continuous-time Kalman filter cannot be used in Function-Call Subsystems or Triggered Subsystems.

References

[1] Franklin, G.F., J.D. Powell, and M.L. Workman, *Digital Control of Dynamic Systems*, Second Edition, Addison-Wesley, 1990.

[2] Lewis, F., *Optimal Estimation*, John Wiley & Sons, Inc, 1986.

See Also

`kalman` | `Recursive Least Squares Estimator` | `Recursive Polynomial Model Estimator`

Related Examples

- “State Estimation Using Time-Varying Kalman Filter”
- “Preprocess Online Parameter Estimation Data in Simulink”
- “Validate Online Parameter Estimation Results in Simulink”
- “Generate Online Estimation Code in Simulink”

Introduced in R2014b

Model Type Converter

Convert polynomial model coefficients to state-space model matrices

Library

Estimators

Description

Use the Model Type Converter block to convert the ARX, ARMAX, OE, or BJ model coefficients into state-space model matrices.

The block inport, u, requires a bus. The number of elements depends on the input polynomial model type:

- ARX — A, B
- ARMAX — A, B, C
- OE — B, F
- BJ — B,C, D, F

These bus elements must contain row vectors of the estimated coefficient values as outputted by the `Recursive Polynomial Model Estimator` block. For MISO data, specify B polynomial coefficients as a matrix where the i-th row parameters correspond to the i-th input. The coefficient values can vary with time. The Model Type Converter block converts these coefficients into the A, B, C, and D matrices of a discrete-time state-space model. The Model Type Converter block outport, y, returns a bus with elements that correspond to the A, B, C, and D matrices of the state-space model. If the signals in u are time-varying, then the state-space matrices are time-varying too.

You can also estimate a state-space model online by using the Recursive Polynomial Model Estimator and Model Type Converter blocks together. Connect the outport of the

Recursive Polynomial Model Estimator block to the inport of the Model Type Converter block to obtain online values of the state-space matrices. The conversion ignores the noise component of the models. In other words, the state-space matrices only capture the y(t)/u(t) relationship.

Parameters

Input model type

Specify the model type coefficients to convert to state-space model matrices. Specify one of the following model types:

- `ARX`
- `ARMAX`
- `OE`
- `BJ`

Ports

Port	Port Type (In/Out)	Description
u	In	Estimated A, B, C, D and F polynomial coefficients, specified as a bus with elements: A, B, C, D and F.
y	Out	State-space model, returned as a bus with elements that correspond to the A, B, C, and D matrices.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

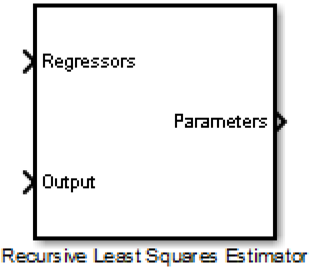
See Also

`Recursive Polynomial Model Estimator`

Introduced in R2014a

Recursive Least Squares Estimator

Estimate model coefficients using recursive least squares (RLS) algorithm



Library

Estimators

Description

Use the Recursive Least Squares Estimator block to estimate the parameters of a system that is linear in the parameters. Such a system has the following form:

$$y(t) = H(t)\theta(t)$$

y and H are known quantities that you provide to the block to estimate θ. The block estimates θ using the recursive least squares algorithm (see [1]).

For a given time step, t, $y(t)$ and $H(t)$ correspond to the Output and Regressors inports of the Recursive Least Squares Estimator block, respectively. $\theta(t)$ corresponds to the Parameters outport. $y(t)$, must be a real-valued scalar. H and θ are real-valued vectors of length N, where N is the number of parameters to be estimated.

For example, suppose you want to estimate a scalar gain, θ, in the system $y = h^2\theta$. Here, y is linear with respect to θ. You can use the Recursive Least Squares Estimator block to estimate θ. Specify y and h^2 as inputs to the Output and Regressor inports.

Parameters

Model Parameters

Initial Estimate

Initial guess of the values of the parameters to be estimated, specified as one of the following options (each option can change the block dialog):

- **None**

  (Default) No initial guess of parameter values is specified. The software uses 1 as the initial guess for each parameter value.

  Specify the following:

  Number of Parameters

  Number of parameters to be estimated, specified as a positive integer.

  The default value is 2.

  Parameter Covariance Matrix

  Initial covariance of parameters, specified as one of the following:

  - Real positive scalar, a — Covariance matrix is an N-by-N diagonal matrix, with a as the diagonal elements.
  - Vector of real positive scalars, $[a_1,...,a_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[a_1,...,a_N]$ as the diagonal elements.
  - N-by-N symmetric positive-definite matrix.

  N is the number of parameters to be estimated.

  This option is applicable only when **Estimation Method** is either `Forgetting Factor` or `Kalman Filter`.

  The default value is 1e4.

- **Internal**

  Specify the initial guess of parameter values in the block dialog. Specify the following:

Initial Parameter Values

Initial guess of the values of the parameters to be estimated, specified as a scalar or vector of length *N*. *N* is the number of parameters to be estimated.

The default value is `[1 1]`.

Parameter Covariance Matrix

Initial covariance of parameters, specified as one of the following:

- Real positive scalar, *a* — Covariance matrix is an *N*-by-*N* diagonal matrix, with *a* as the diagonal elements.
- Vector of real positive scalars, $[a_1,...,a_N]$ — Covariance matrix is an *N*-by-*N* diagonal matrix, with $[a_1,...,a_N]$ as the diagonal elements.
- *N*-by-*N* symmetric positive-definite matrix.

N is the number of parameters to be estimated.

This option is applicable only when **Estimation Method** is either `Forgetting Factor` or `Kalman Filter`.

The default value is 1e4.

- **External**

Specify the initial guess of parameter values as an input signal to the Recursive Least Squares Estimator block. If you select this option, the block includes additional inports:

- InitialParameters — Initial guess of the values of the parameters to be estimated, specified as a scalar or vector of length *N*. *N* is the number of parameters to be estimated.
- InitialCovariance — Initial covariance of parameters, specified as one of the following:
  - Real positive scalar, *a* — Covariance matrix is an *N*-by-*N* diagonal matrix, with *a* as the diagonal elements.
  - Vector of real positive scalars, $[a_1,...,a_N]$ — Covariance matrix is an *N*-by-*N* diagonal matrix, with $[a_1,...,a_N]$ as the diagonal elements.
  - *N*-by-*N* symmetric positive-definite matrix.

N is the number of parameters to be estimated.

The InitialCovariance inport is included only when **Estimation Method** is either `Forgetting Factor` or `Kalman Filter`.

You must also specify the following in the block dialog:

Number of Parameters

Number of parameters to be estimated, specified as a positive integer.

Suppose you specify a vector as an input to the InitialParameters inport. The length of this vector must match the value specified in **Number of Parameters**.

The default value is 2.

Sample Time

Block sample time, specified as -1 or a positive scalar.

The default value is -1. The block inherits its sample time based on the context of the block within the model.

Algorithm and Block Options

Algorithm Options

Estimation Method

Recursive estimation algorithm, specified as one of the following (each option can change the block dialog):

- `Forgetting Factor` — (Default) Forgetting factor algorithm

  If you select this option, you must specify the **Forgetting Factor**, λ, as a scalar in the (0 1] range. λ specifies the measurement window relevant for parameter estimation. Suppose the system remains approximately constant over T_0 samples. You can choose λ such that:

$$T_0 = \frac{1}{1-\lambda}$$

Setting $\lambda = 1$ corresponds to “no forgetting” and estimating constant coefficients. Setting $\lambda < 1$ implies that past measurements are less significant for parameter estimation and can be “forgotten.” Set $\lambda < 1$ to estimate time-varying coefficients. Typical choices of λ are in the [0.98 0.995] range.

The default value is 1.

- `Kalman Filter` — Kalman filter algorithm

  If you select this option, you must specify the **Noise Covariance Matrix** as one of the following:

  - Real nonnegative scalar, a — Covariance matrix is an N-by-N diagonal matrix, with a as the diagonal elements.
  - Vector of real nonnegative scalars, $[a_1,...,a_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[a_1,...,a_N]$ as the diagonal elements.
  - N-by-N symmetric positive semidefinite matrix.

  N is the number of parameters to be estimated.

  0 values in the noise covariance matrix correspond to estimating constant coefficients. Values larger than 0 correspond to time-varying parameters. Large values correspond to rapidly changing parameters.

  The default value is 1.

- `Normalized Gradient` — Normalized gradient adaptation algorithm

  If you select this option, you must specify the following:

  - **Adaptation Gain** — Adaptation gain, γ, specified as a real positive scalar. γ is directly proportional to the relative information content in the measurements. That is, when your measurements are trustworthy, specify a large value for γ.

    The default value is 1.

  - **Normalization Bias** — Bias in adaptation gain scaling, *Bias*, specified as a real nonnegative scalar. The normalized gradient algorithm scales the adaptation gain at each step by the square of the two-norm of the gradient vector. If the gradient is close to zero, this can cause jumps in the estimated parameters. *Bias* is the term introduced in the denominator to prevent these jumps. Increase *Bias* if you observe jumps in estimated parameters.

The default value is `eps`.

- `Gradient` — Unnormalized gradient adaptation algorithm

  If you select this option, you must specify the **Adaptation Gain**, γ, as a real, positive scalar. γ is directly proportional to the relative information content in the measurements. That is, when your measurements are trustworthy, specify a large value for γ, and vice versa.

  The default value is 1.

For more information about these algorithms, see “Recursive Algorithms for Online Parameter Estimation”.

Block Options

Output estimation error

Add Error outport to the block. Use this signal to validate the estimation.

For a given time step, t, the estimation error is calculated as:

$$e(t) = y(t) - yest(t).$$

☑ On

Add Error outport.

☐ Off

(Default) Do not add Error outport.

Output parameter covariance matrix

Add Covariance outport to the block. Use this signal to examine parameter estimation uncertainty.

Parameter covariance is computed assuming that the residuals, $e(t)$, are white noise, and the variance of these residuals is 1.

This option is not available when **Estimation Method** is either `Normalized Gradient` or `Gradient`.

☑ On

Add Covariance outport.

☐ Off

(Default) Do not add Covariance outport.

Add enable port

Add Enable inport to the block. Use this input signal to specify a control signal that enables or disables parameter estimation. The block estimates the parameter values for each time step that parameter estimation is enabled. If you disable parameter estimation at a given step, *t*, then the software does not update the parameters for that time step. Instead, the block outputs the last estimated parameter values. Use this option, for example, to disable parameter estimation when the system enters a mode where the parameter values do not vary with time.

☑ On

Add Enable inport.

☐ Off

(Default) Do not add Enable inport.

External reset

Option to reset estimated parameters and parameter covariance matrix using specified initial values.

Suppose you reset the block at a time step, *t*. If the block is enabled at *t*, the software uses the initial parameter values specified in **Initial Estimate** to estimate the parameter values. In other words, at *t*, the block performs a parameter update using the initial estimate and the current values of the inports. The block outputs these updated parameter value estimates using the Parameters outport.

If the block is disabled at *t* and you reset the block, the block outputs the values specified in **Initial Estimate**.

Use this option, for example, when you reset the input because it did not excite the system as needed, resulting in poor estimation results.

Specify this option as one of the following:

- `None` — (Default) Estimated parameters and covariance matrix values are not reset.

- `Rising` — Triggers reset when the control signal rises from a negative or zero value to a positive value. If the initial value is negative, rising to zero triggers reset.
- `Falling` — Triggers reset when the control signal falls from a positive or a zero value to a negative value. If the initial value is positive, falling to zero triggers reset.
- `Either` — Triggers reset when the control signal is either rising or falling.
- `Level` — Triggers reset in either of these cases:
  - Control signal is nonzero at the current time step
  - Control signal changes from nonzero at the previous time step to zero at the current time step
- `Level hold` — Triggers reset when the control signal is nonzero at the current time step.

When you choose any option other than `None`, the software adds a Reset inport to the block. You provide the reset control input signal to this inport.

Ports

Port Name	Port Type	Description
Regressors	In	$H(t)$, specified as a real scalar or vector.
Output	In	$y(t)$, specified as a real scalar signal.
Parameters	Out	$\theta(t)$, returned as a vector of the same length as $H(t)$.
Enable (Optional)	In	Control signal to enable parameter estimation, specified as a scalar.
Reset (Optional)	In	Control signal to reset parameter estimation, specified as a scalar.
InitialParamete (Optional)	In	Initial guess of the values of the parameters to be estimated, specified as a scalar or vector.
InitialCovarian (Optional)	In	Initial covariance of parameters, specified as a real positive scalar, vector of real positive scalars, or symmetric positive-definite matrix.

Port Name	Port Type	Description
Error (Optional)	Out	Estimation error, returned as a scalar.
Covariance (Optional)	Out	Covariance of estimated parameters, returned as an N-by-N matrix. N is the number of parameters.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

Note: The Regressors and Output inports must have matching data types.

References

[1] Ljung, L. *System Identification: Theory for the User*. Upper Saddle River, NJ: Prentice-Hall PTR, 1999, pp. 363–369.

See Also

`Kalman Filter` | `Recursive Polynomial Model Estimator`

Related Examples

- “Online Recursive Least Squares Estimation”
- “Preprocess Online Parameter Estimation Data in Simulink”
- “Validate Online Parameter Estimation Results in Simulink”
- “Generate Online Estimation Code in Simulink”

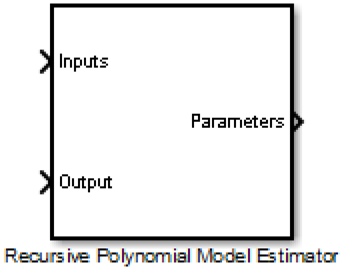
More About

- “Recursive Algorithms for Online Parameter Estimation”

Introduced in R2014a

Recursive Polynomial Model Estimator

Estimate input-output and time-series polynomial model coefficients



Library

Estimators

Description

Use the Recursive Polynomial Model Estimator block to estimate discrete-time input-output polynomial and time-series models.

These model structures are:

- AR — $A(q)y(t) = e(t)$
- ARMA — $A(q)y(t) = C(q)e(t)$
- ARX — $A(q)y(t) = B(q)u(t{-}nk)+e(t)$
- ARMAX — $A(q)y(t) = B(q)u(t{-}nk)+C(q)e(t)$
- OE — $y(t) = \dfrac{B(q)}{F(q)}u(t-n_k) + e(t)$
- BJ — $y(t) = \dfrac{B(q)}{F(q)}u(t-n_k) + \dfrac{C(q)}{D(q)}e(t)$

q is the time-shift operator and nk is the delay. $u(t)$ is the input, $y(t)$ is the output, and $e(t)$ is the error. For MISO models, there are as many $B(q)$ polynomials as the number of inputs. The orders of these models are:

- na — $1+a_1q^{-1}+a_2q^{-2}+...+a_{na}q^{-na}$
- nb — $b_1+b_2q^{-1}+b_3q^{-2}+...+b_{nb}q^{-(nb-1)}$
- nc — $1+c_1q^{-1}+c_2q^{-2}+...+c_{nc}q^{-nc}$
- nd — $1+d_1q^{-1}+d_2q^{-2}+...+d_{nd}q^{-nd}$
- nf — $1+f_1q^{-1}+f_2q^{-2}+...+f_{nf}q^{-nf}$

The orders na, nb, nc, nd, nf and delay, nk, are known a priori and fixed. These are provided through the **Model Parameters** tab of the block dialog. $u(t)$ and $y(t)$ are provided through the Inputs and Outputs inports, respectively. The block estimates the A(q), B(q), C(q), D(q) and F(q) coefficients and outputs them in the Parameters outport. This outport provides a bus signal with the following elements:

- A — Vector containing [1 $a_1(t)$ ... $a_{na}(t)$].
- B — Vector containing [`zeros(1,nk)` $b_1(t)$ ... $b_{nb}(t)$]. For MISO data, B is a matrix where the i-th row parameters correspond to the i-th input.
- C — Vector containing [1 $c_1(t)$ ... $c_{nc}(t)$].
- D — Vector containing [1 $d_1(t)$ ... $d_{nd}(t)$].
- F — Vector containing [1 $f_1(t)$ ... $f_{nf}(t)$].

For example, suppose you want to estimate the coefficients for the following SISO ARMAX model:

$$y(t)+a_1y(t-1)+...+a_{na}y(t-na) = b_1u(t-nk)+...+b_{nb}u(t-nb-nk+1)+c_1e(t-1)+...+c_{nc}e(t-nc)$$

y, u, na, nb, nc, nd, nf, and nk are known quantities that you provide to the block. The block estimates the A, B, C, D, and F parameter values. Estimated C, D, and F polynomials are enforced to be stable, that is, having roots in the unit disk. Estimated A and B polynomials can be unstable.

For a given time step, t, specify y and u as inputs to the Output and Inputs inports, respectively. Specify the na, nb, nc, and nk values in the **Model Parameters** tab of the block dialog. The block estimates the A, B, C, D and F parameter values and outputs these estimated values using the Parameters outport.

Parameters

Model Structure

Estimated model structure, specified as one of the following:

- `ARX` — SISO or MISO ARX model.
- `ARMAX` — SISO ARMAX model.
- `OE` — SISO OE model.
- `BJ` — SISO BJ model.
- `AR` — Time-series AR model.
- `ARMA` — Time-series ARMA model.

Model Parameters

Initial Estimate

Initial guess of the values of the parameters to be estimated, specified as one of the following options:

- **None**

  Number of Parameters in A(q) (na)

  Number of estimated parameters in the $A(q)$ polynomial, specified as a positive scalar integer.

  The default value is `1`.

  Number of Parameters in B(q) (nb)

  Number of estimated parameters in the $B(q)$ polynomial, specified as a vector of positive integers.

  For MISO systems, specify $B(q)$ as a vector with elements specifying the order for each input. (Applicable for only ARX models.)

  The default value is `1`.

  Number of Parameters in C(q) (nc)

  Number of estimated parameters in the $C(q)$ polynomial, specified as a positive scalar integer.

This option is applicable for ARMA, ARMAX and BJ models only.

The default value is `1`.

Number of Parameters in D(q) (nd)

Number of estimated parameters in the $D(q)$ polynomial, specified as a positive scalar integer.

This option is applicable for BJ models only.

The default value is `1`.

Number of Parameters in F(q) (nf)

Number of estimated parameters in the $F(q)$ polynomial, specified as a positive scalar integer.

This option is applicable for OE and BJ models only.

The default value is `1`.

Input Delay (nk)

Input-output delay, expressed as fixed leading zeros of the B(q) polynomial, specified as a vector of positive integers.

For MISO systems, specify nk as a vector with elements specifying the delay for each input. (Applicable for only ARX models.)

This vector is of length Nu, where Nu is the number of inputs.

The default value is `1`.

Parameter Covariance Matrix

Initial covariance of parameters, specified as one of the following:

- Real positive scalar, a — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements.
- Vector of real positive scalars, [$a(a)$,$a(b)$,$a(c$, $a(d)$, $a(f)$] — Covariance matrix is an N-by-N diagonal matrix, with [$a(a)$,$a(b)$,$a(c)$,$a(d)$, $a(f)$] as the diagonal elements. $a(a)$ is a vector of the covariance for each coefficient of the A polynomial. Similarly, $a(b)$, $a(c)$, $a(d)$ and $a(f)$ are vectors containing

the covariance of the coefficients of the B, C, D and F polynomials, respectively.

- N-by-N symmetric positive-definite matrix.

  N can be one of the following:

  - AR — $N = na$
  - ARX — $N = na + \sum_{i=1}^{N_u} nb_i$
  - ARMA — $N = na + nc$
  - ARMAX — $N = na + nb + nc$
  - OE — $N = nb + nf$
  - BJ — $N = nb + nc + nd + nf$

This option is applicable only when **Estimation Method** is either `Forgetting Factor` or `Kalman Filter`.

The default value is `1e4`.

- **Internal**

Specify the initial guess of the parameter values in the block dialog. Specify the following:

Initial A(q)

Initial guess of the values of the parameters to be estimated for the $A(q)$ polynomial (applies only to AR, ARX, ARMA and ARMAX models). Must be a row vector of length $na + 1$. The leading coefficient of A must be 1.

The default value is `[1 eps]`.

Initial B(q)

Initial guess of the values of the parameters to be estimated for the $B(q)$ polynomial (applies only to ARX, ARMAX, OE, BJ models). Must be a row vector of length $nb + nk$.

The leading zeros in *B*(*q*) are counted and interpreted as *nk*. Those zeros are fixed throughout the estimation. *nb* is the number of elements after the first nonzero element in *B*(*q*). The block estimates the value of these elements.

For example:

- `[0 eps]` corresponds to *nk*=1 and *nb*=1
- `[0 0 eps]` corresponds to *nk*=2 and *nb*=1
- `[0 0 eps 0 eps]` corresponds to *nk*=2 and *nb*=3

The default value is `[0 eps]`.

Initial C(q)

Initial guess of the values of the parameters to be estimated for the *C*(*q*) polynomial, specified as a row vector of length *nc* + 1. The leading coefficient of *C*(*q*) must be 1.

The coefficients must define a stable discrete-time polynomial, that is, have all polynomial roots within the unit circle.

This option is applicable for ARMAX, ARMA and BJ models.

The default value is `[1 eps]`.

Initial D(q)

Initial guess of the values of the parameters to be estimated for the *D*(*q*) polynomial, specified as a row vector of length *nd* + 1. The leading coefficient of *D*(*q*) must be 1.

The coefficients must define a stable discrete-time polynomial, that is, have all polynomial roots within the unit circle.

This option is applicable for BJ models.

The default value is `[1 eps]`.

Initial F(q)

Initial guess of the values of the parameters to be estimated for the *F*(*q*) polynomial, specified as a row vector of length *nf* + 1. The leading coefficient of *F*(*q*) must be 1.

The coefficients must define a stable discrete-time polynomial, that is, have all polynomial roots within the unit circle.

This option is applicable for OE and BJ models.

The default value is `[1 eps]`.

Parameter Covariance Matrix

Initial covariance of parameters, specified as one of the following:

- Real positive scalar, a — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements.
- Vector of real positive scalars, [$a(a)$,$a(b)$,$a(c$, $a(d)$, $a(f)$] — Covariance matrix is an N-by-N diagonal matrix, with [$a(a)$,$a(b)$,$a(c)$,$a(d)$, $a(f)$] as the diagonal elements. $a(a)$ is a vector of the covariance for each coefficient of the A polynomial. Similarly, $a(b)$, $a(c)$, $a(d)$ and $a(f)$ are vectors containing the covariance of the coefficients of the B, C, D and F polynomials, respectively.
- N-by-N symmetric positive-definite matrix.

  N can be one of the following:

  - AR — $N = na$
  - ARX — $N = na + \sum_{i=1}^{N_u} nb_i$
  - ARMA — $N = na + nc$
  - ARMAX — $N = na + nb + nc$
  - OE — $N = nb + nf$
  - BJ — $N = nb + nc + nd + nf$

This option is applicable only when **Estimation Method** is either `Forgetting Factor` or `Kalman Filter`.

The default value is `1e4`.

- **External**

  Specify the initial guess of parameter values as an input signal to the Recursive Polynomial Model Estimator block. If you select this option, the block includes additional inports:

  - InitialParameters — Initial guess of the values of the parameters to be estimated, specified as a bus with elements A, B, C, D and F as required by the model type. For a description of these signals, see the **Internal** option description.
  - Initial Covariance — Initial covariance of parameters, specified as one of the following:

    - Real positive scalar, a — Covariance matrix is an N-by-N diagonal matrix, with α as the diagonal elements.
    - Vector of real positive scalars, $[a(a),a(b),a(c, a(d), a(f)]$ — Covariance matrix is an N-by-N diagonal matrix, with $[a(a),a(b),a(c),a(d), a(f)]$ as the diagonal elements. $a(a)$ is a vector of the covariance for each coefficient of the A polynomial. Similarly, $a(b)$, $a(c)$, $a(d)$ and $a(f)$ are vectors containing the covariance of the coefficients of the B, C, D and F polynomials, respectively.
    - N-by-N symmetric positive-definite matrix.

      N can be one of the following:

      - AR — $N = na$
      - ARX — $N = na + \sum_{i=1}^{N_u} nb_i$
      - ARMA — $N = na + nc$
      - ARMAX — $N = na + nb + nc$
      - OE — $N = nb + nf$
      - BJ — $N = nb + nc + nd + nf$

    The InitialCovariance inport is included only when **Estimation Method** is either `Forgetting Factor` or `Kalman Filter`.

  You must also specify the following in the block dialog:

Number of Parameters in A(q) (na)

Number of parameters to be estimated for the $A(q)$ polynomial, specified as a positive integer.

The default value is `1`.

Number of Parameters in B(q) (nb)

Number of parameters to be estimated for the $B(q)$ polynomial, specified as a positive integer.

For MISO systems, specify $B(q)$ as a vector with elements specifying the order for each input. (Applicable for only ARX models.)

The default value is `1`.

Number of Parameters in C(q) (nc)

Number of parameters to be estimated for the $C(q)$ polynomial, specified as a positive integer.

This option is applicable for ARMA, ARMAX and BJ models only.

The default value is `1`.

Number of Parameters in D(q) (nd)

Number of parameters to be estimated in the $D(q)$ polynomial, specified as a positive scalar integer.

This option is applicable for BJ models only.

The default value is `1`.

Number of Parameters in F(q) (nf)

Number of parameters to be estimated in the $F(q)$ polynomial, specified as a positive scalar integer.

This option is applicable for OE and BJ models only.

The default value is `1`.

Input Delay (nk)

Input-output delay, expressed as fixed leading zeros of the B(q) polynomial, specified as a vector of positive integers.

For MISO systems, specify *nk* as a vector with elements specifying the delay for each input. (Applicable for only ARX models.)

This vector is of length *Nu*, where *Nu* is the number of inputs.

The default value is `1`.

Default: `'None'`

Sample Time

Block sample time, specified as -1 or a positive scalar.

The default value is -1. The block inherits its sample time based on the context of the block within the model.

Algorithm and Block Options

Algorithm Options

Estimation Method

Recursive estimation algorithm, specified as one of the following (each option can change the block dialog):

- `Forgetting Factor` — (Default) Forgetting factor algorithm

  If you select this option, you must specify the **Forgetting Factor**, λ, as a scalar in the (0 1] range. λ specifies the measurement window relevant for parameter estimation. Suppose the system remains approximately constant over T_0 samples. You can choose λ such that:

$$T_0 = \frac{1}{1-\lambda}$$

  Setting $\lambda = 1$ corresponds to “no forgetting” and estimating constant coefficients. Setting $\lambda < 1$ implies that past measurements are less significant for parameter estimation and can be “forgotten.” Set $\lambda < 1$ to estimate time-varying coefficients. Typical choices of λ are in the [0.98 0.995] range.

  The default value is 1.

- `Kalman Filter` — Kalman filter algorithm

  If you select this option, you must specify the **Noise Covariance Matrix** as one of the following:

  - Real nonnegative scalar, a — Covariance matrix is an N-by-N diagonal matrix, with a as the diagonal elements.
  - Vector of real nonnegative scalars, $[a_1,...,a_N]$ — Covariance matrix is an N-by-N diagonal matrix, with $[a_1,...,a_N]$ as the diagonal elements.
  - N-by-N symmetric positive semidefinite matrix.

  N is the number of parameters to be estimated.

  0 values in the noise covariance matrix correspond to estimating constant coefficients. Values larger than 0 correspond to time-varying parameters. Large values correspond to rapidly changing parameters.

  The default value is 1.
- `Normalized Gradient` — Normalized gradient adaptation algorithm

  If you select this option, you must specify the following:

  - **Adaptation Gain** — Adaptation gain, γ, specified as a real positive scalar. γ is directly proportional to the relative information content in the measurements. That is, when your measurements are trustworthy, specify a large value for γ.

    The default value is 1.
  - **Normalization Bias** — Bias in adaptation gain scaling, *Bias*, specified as a real nonnegative scalar. The normalized gradient algorithm scales the adaptation gain at each step by the square of the two-norm of the gradient vector. If the gradient is close to zero, this can cause jumps in the estimated parameters. *Bias* is the term introduced in the denominator to prevent these jumps. Increase *Bias* if you observe jumps in estimated parameters.

    The default value is `eps`.
- `Gradient` — Unnormalized gradient adaptation algorithm

  If you select this option, you must specify the **Adaptation Gain**, γ, as a real, positive scalar. γ is directly proportional to the relative information content in the measurements. That is, when your measurements are trustworthy, specify a large value for γ, and vice versa.

The default value is 1.

For more information about these algorithms, see "Recursive Algorithms for Online Parameter Estimation".

Block Options

Output estimation error

Add Error outport to the block. Use this signal to validate the estimation.

For a given time step, t, the estimation error is calculated as:

$$e(t) = y(t) - yest(t).$$

☑ On

Add Error outport.

☐ Off

(Default) Do not add Error outport.

Output parameter covariance matrix

Add Covariance outport to the block. Use this signal to examine parameter estimation uncertainty.

Parameter covariance is computed assuming that the residuals, $e(t)$, are white noise, and the variance of these residuals is 1.

This option is not available when **Estimation Method** is either `Normalized Gradient` or `Gradient`.

☑ On

Add Covariance outport.

☐ Off

(Default) Do not add Covariance outport.

Add enable port

Add Enable inport to the block. Use this input signal to specify a control signal that enables or disables parameter estimation. The block estimates the parameter values for each time step that parameter estimation is enabled. If you disable parameter estimation at a given step, *t*, then the software does not update the parameters for that time step. Instead, the block outputs the last estimated parameter values. Use this option, for example, to disable parameter estimation when the system enters a mode where the parameter values do not vary with time.

☑ On

Add Enable inport.

☐ Off

(Default) Do not add Enable inport.

External reset

Option to reset estimated parameters and parameter covariance matrix using specified initial values.

Suppose you reset the block at a time step, *t*. If the block is enabled at *t*, the software uses the initial parameter values specified in **Initial Estimate** to estimate the parameter values. In other words, at *t*, the block performs a parameter update using the initial estimate and the current values of the inports. The block outputs these updated parameter value estimates using the Parameters outport.

If the block is disabled at *t* and you reset the block, the block outputs the values specified in **Initial Estimate**.

Use this option, for example, when you reset the input because it did not excite the system as needed, resulting in poor estimation results.

Specify this option as one of the following:

- `None` — (Default) Estimated parameters and covariance matrix values are not reset.
- `Rising` — Triggers reset when the control signal rises from a negative or zero value to a positive value. If the initial value is negative, rising to zero triggers reset.
- `Falling` — Triggers reset when the control signal falls from a positive or a zero value to a negative value. If the initial value is positive, falling to zero triggers reset.

- `Either` — Triggers reset when the control signal is either rising or falling.
- `Level` — Triggers reset in either of these cases:
    - Control signal is nonzero at the current time step
    - Control signal changes from nonzero at the previous time step to zero at the current time step
- `Level hold` — Triggers reset when the control signal is nonzero at the current time step.

When you choose any option other than `None`, the software adds a Reset inport to the block. You provide the reset control input signal to this inport.

Ports

Port Name	**Port Type (In/ Out)**	**Description**
Inputs	In	$u(t)$, specified as a real scalar or vector. The port is available when the **Model Structure** is `ARX`, `ARMAX`, `BJ` or `OE`.
Output	In	$y(t)$, specified as a real scalar signal.
Parameters	Out	Estimated polynomial coefficients, returned as a bus. The bus contains an element each for the A, B, C, D and F polynomials. The following signals are available for the corresponding model: • `AR` • `ARX` • `ARMA` • `ARMAX` • `OE` • `BJ` Each bus element contains a vector of the associated polynomial coefficients. For example, the A element contains [1 $a_1(t)$ ... $a_{na}(t)$].

Port Name	Port Type (In/ Out)	Description
		Estimated C, D, and F polynomials are enforced to be stable, that is, have all roots in the unit disk. Estimated A and B polynomials can be unstable.
Enable (Optional)	In	Control signal to enable parameter estimation, specified as a scalar.
Reset (Optional)	In	Control signal to reset parameter estimation, specified as a scalar.
InitialParamete (Optional)	In	Initial guess of the values of the parameters to be estimated, specified as a bus.
InitialCovarian (Optional)	In	Initial covariance of parameters, specified as a real nonnegative scalar, vector of real nonnegative scalars, or positive semi-definite matrix.
Error (Optional)	Out	Estimation error, returned as a scalar.
Covariance (Optional)	Out	Covariance of estimated parameters, returned as an N-by-N matrix. N is the number of parameters.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

Note: The Inputs and Output inports must have matching data types.

References

[1] Ljung, L. *System Identification: Theory for the User*. Upper Saddle River, NJ: Prentice-Hall PTR, 1999, pp. 363–369.

See Also

`Kalman Filter` | `Recursive Least Squares Estimator`

Related Examples

- "Online ARMAX Polynomial Model Estimation"
- "Preprocess Online Parameter Estimation Data in Simulink"
- "Validate Online Parameter Estimation Results in Simulink"
- "Generate Online Estimation Code in Simulink"

More About

- "What Are Polynomial Models?"
- "Recursive Algorithms for Online Parameter Estimation"

Introduced in R2014a